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

A PHYSICALLY-ADDRESSED L4 KERNEL

Abi Nourai

Bachelor of Engineering (Computer Engineering)

March 2005

Supervisor: Gernot Heiser
Assessor: Kevin Elphinstone

Abstract

All current implementations of the L4 microkernel map thread control blocks (TCBs) into a linear array in virtual memory, a decision that was originally made almost entirely for the performance advantages it offers on the Intel 486 platform. The drawback of this design choice is that page faults generated within L4 complicate the kernel and in particular its verification by formal methods.

An alternative exists however on architectures where physical addressing, or at least a loose equivalent in superpages, is available. On such architectures, TCBs may be addressed physically via indirection provided by an auxiliary lookup table. Addressing TCBs in this manners leads to a completely physically-addressed L4 kernel that offers advantages in simplicity, but has a non-obvious effect on the cache footprint of the performance-critical IPC path that warrants examination.

This thesis endeavours to provide a thorough investigation into the performance trade-offs involved in making the L4Ka::Pistachio implementation of the L4 Version 4 API completely physically addressed. We stress architectural issues that effect the outcome of these trade-offs and explore various implementational design choices that aim to weaken the performance penalties a physically-addressed kernel may suffer. We conclude by running Linux on top of the L4 microkernel to obtain a concise set of benchmarks that prove, at least for the MIPS64 architecture, that the simplicity of a completely physically-addressed L4 kernel may be enjoyed without any notable performance degradation.

Acknowledgements

I would like to thank all those members of the DiSy group at the University of New South Wales who freely volunteered their guidance and wisdom to me over the course of my thesis. It was a pleasure to work with such a diverse but tightly-knit group of people.

I would especially like to thank Carl van Schaik and Matthew Chapman for allowing me to invade their cubicles unannounced as often as I pleased. The many hours of help they provided was far beyond the call of duty, and proved invaluable to my work.

My supervisor, Gernot Heiser, also deserves special mention for providing me with the opportunity to work on such an interesting and challenging project. His encouragement, feedback and standards of excellence made the last 12 months the most rewarding year of my engineering degree.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

L4 is a second-generation microkernel based on the axioms of minimality, extensibility and flexibility. It offers all the advantages of the classical microkernel approach without suffering the performance degradation exhibited by earlier microkernels. The L4 philosophy is founded on the principle that providing only a minimal set of abstractions leads to improved efficiency and flexibility. The documented success of L4 compared to the so-called first generation microkernels validates this philosophy.

The approach adopted by L4 in minimising what is contained in the kernel not only improves performance, but also offers advantages in simplicity. The L4Ka::Pistachio implementation of the L4 API contains in the order of 10,000 lines of C++ and assembly code — an order of magnitude smaller than the first-generation Mach microkernel and two orders of magnitude smaller than the Linux kernel. A microkernel of this size becomes a strong candidate for formal verification, a process by which the kernel source code can be shown to meet correctness and security constraints by mathematical proof. Formally verifying a microkernel is highly desirable because its correctness, reliability and robustness are a prerequisite for the correct and safe execution of any user-level application.

It is a characteristic of software development however, that striving for improved performance often runs counter to simplicity. In particular this is true for the manner in which current implementations of the L4 microkernel choose to address a critical class of kernel-maintained data structures called thread control blocks (TCBs). All current L4 implementations map TCBs into a linear array in virtual memory, a decision that was originally made almost entirely for the performance advantages it offers on the Intel 486 platform. Unfortunately this design choice implies that execution of kernel code cannot be assumed to be sequential, because the occurrence of page faults and, depending on the architecture, TLB-miss exceptions, may interrupt the kernel's smooth operation. This impedes the formal verification process by complicating the development of an accurate formal model representative of the kernel's behaviour.

The alternative to addressing TCBs as a virtual linear array is to address TCBs physically, but indirectly through a lookup table. The motivation for the latter approach is to obtain a completely physically-addressed kernel that is simpler because it is void of the possibility of page faults being generated from within the kernel. On the surface it would appear that this goal is only achievable on architectures that support physical addressing of memory. However we will show that architectures supporting superpages are equally capable of eliminating page faults from within the kernel. In particular this goal is attainable for every architecture for which there currently exists an implementation of L4Ka::Pistachio.

For the Intel 486 platform where L4 was originally implemented, it is not difficult to show that the current method of addressing TCBs offers significant performance advantages over addressing TCBs through a lookup table. However for computer architectures that permit physical addressing, or at least a loose equivalent in superpages, the performance implications associated with physically addressing TCBs are ambiguous and need

to be thoroughly evaluated. To understand why, one must first appreciate what is required of a microkernel to achieve high levels of performance.

Comparisons between L4 and first-generation microkernels such as Mach have highlighted that efficient inter-process communication (IPC) provided by the microkernel is a prerequisite for overall system performance to be fast. In particular, studies have shown that the cache footprint of a microkernel's IPC mechanism must be small if system performance is not to suffer. The underlying thesis that any microkernel implementation must respect is that IPC may be invoked sufficiently frequently on a microkernel-based system that user-level servers and applications must perpetually compete for any portion of the system's caches consumed by IPC primitives.

It has been well established that current implementations of L4 offer highly efficient IPC that is an order of magnitude faster than that found in first-generation microkernels. A small cache footprint largely contributes to the performance achievements of L4. It turns out that choosing to physically address TCB structures impacts on the nature of this cache footprint. On one hand, table lookups required to perform TCB physical addressing potentially increase the usage of a system's memory caches. On the other hand, dispensing with virtual addressing relieves pressure on the translate lookaside buffer (TLB). Although the performance trade-offs associated with physically addressing TCBs are more intricate than this alone, the non-trivial impact a physically-addressed L4 microkernel will have on the performance-critical IPC path is enough to warrant a thorough performance analysis of such a microkernel. The overriding goal of our work is to provide that analysis.

In this thesis we modify the L4Ka::Pistachio implementation of the L4 API so as to obtain a physically-addressed L4 kernel that can be subjected to scrutiny via benchmarking. In doing so, a number of implementational design choices are explored, each seeking to weaken any performance penalties a physically-addressed kernel may endure. The investigation itself is largely conducted in the context of the MIPS R4700 64-bit processor. Despite this, we ensure to highlight any architectural properties that have a particularly strong influence on our analysis and results so that we can anticipate a physically-addressed kernel's performance impact on other hardware platforms.

Our evaluation is performed in two stages. In the first stage we perform a series of microbenchmarks that quantify the precise impact a physically-addressed kernel has on the performance-critical IPC path. Both cycle counts and cache footprint are considered. In the second stage we run Linux on top of the L4 microkernel so that we can investigate the overall effect a physically-addressed kernel has on a non-trivial L4-based system. We use the AIM7 multiuser benchmark suite in conjunction with traditional kernel compiles to produce a concise set of results demonstrating that, at least for the MIPS R4700 platform, the simplicity gained from a completely physically-addressed L4 kernel may be enjoyed without any notable performance degradation.

Thesis Outline

Chapter Two: Background

Chapter two presents the background material required for the remaining chapters in this thesis. It begins by describing the memory-management subsystem on modern architectures — an understanding of which is required for appreciating the subtleties of the trade-offs associated with a physically-addressed kernel. The chapter concludes with an introduction to the L4 microkernel, describing its key concepts and abstractions, and highlighting any implementational properties of L4Ka::Pistachio that prove relevant to our work.

Chapter Three: Physically-Addressed L4 Kernel

Before investigating the trade-offs involved with making the L4Ka::Pistachio implementation of L4 completely physically addressed, this thesis first establishes a foundation by not only describing, but also rationalising, the

current state of addressing in Pistachio. It turns out that addressing thread control blocks (TCBs) within the kernel is the major impediment to obtaining a physically-addressed kernel.

We ensure to explain why current implementations of L4 all choose to address TCBs virtually, and then proceed to motivate the quest for removing virtual addressing from within Pistachio. In particular we explain why not having to handle page faults within the kernel leads to simplicity, and why this simplicity is worth striving for. The third chapter concludes with a qualitative architecture-independent exposition of both the performance and non-performance trade-offs associated with physically addressing TCB structures.

Chapter Four: Design & Implementation

In Chapter four, this thesis turns to addressing the performance impact of a physically-addressed Pistachio kernel. We conduct our analysis in the context of the MIPS64 architecture and focus particularly on the effect on the performance-critical IPC path. A number of implementational design choices are proposed that aim to reduce performance penalties a physically-addressed kernel may suffer from. Chapter four concludes with a presentation of a number of concrete physically-addressed kernel implementations whose impact on the performance-critical IPC path is carefully analysed.

Chapter Five: Evaluation

Chapter five evaluates the performance of the concrete physically-addressed kernel implementations introduced in the preceding chapter. By running Linux on top of the L4 microkernel, traditional and standardised benchmarks are used to quantify the overall effect these kernels have on an L4-based system. Throughout this chapter, a strong emphasis is placed on appreciating the influence architectural properties and implementational design choices have on benchmark results.

Chapter Six: Epilogue

The concluding chapter summarises the achievements of this thesis and then proposes a course for future work that addresses any shortcomings.

Conventions and Caveats

- When referring to the L4 microkernel unqualified, we are specifically referring to L4 as defined by the *eXperimental* Version X.2 API. When referring to the L4Ka::Pistachio implementation of the L4 API, we specifically refer to the Version 0.4 release of L4Ka::Pistachio made by the L4Ka team in June 2004.

- New keywords are generally introduced *emphasised*. Code snippets are typeset using `typewriter` font. Acronyms are capitalised. L4 system calls are typeset in SMALL CAPS. For example, IPC should be treated as the acronym for inter-process communication, but IPC refers specifically to the L4 system call that performs inter-process communication.

- It is somewhat unfortunate that the TCB acronym is overloaded in the literature. It is used as an acronym for both *trusted computing base* and for the *thread control block* data structure. Throughout this document, the TCB acronym is always used to mean the latter. When we need to refer to the former, we use the unabbreviated form to avoid confusion.

  Furthermore, when referring to L4 TCBs unqualified, we specifically refer to the kernel-maintained TCBs (KTCBs) and not the user-accessible TCBs (UTCBs). When there is scope for confusion, we shall explicitly use the KTCB and UTCB acronyms.

Chapter 2

Background

This thesis is concerned with investigating the trade-offs associated with obtaining a completely physically-addressed L4 kernel. Understanding the subtlety of these trade-offs requires an understanding of the memory-management subsystem of modern architectures. Hence this thesis begins with an introduction to two of the most important elements of modern memory-management units — virtual memory and caches. It concludes with an introduction to the L4 microkernel.

2.1 Virtual Memory

2.1.1 Address translation

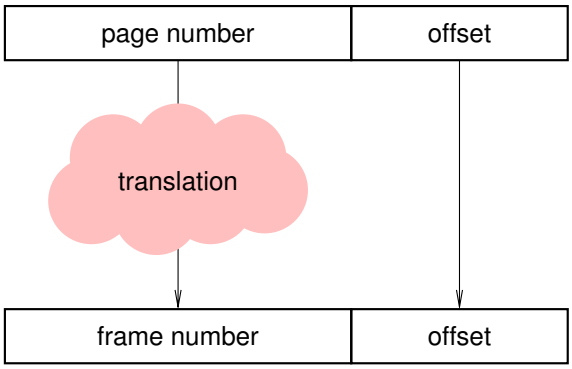

Figure 2.1: Address translation.

The usefulness of a *microprocessor* resides in its ability to execute programs. Doing so requires data (instructions and operands) to be fetched from and stored to *physical memory*. The location of such data in physical memory is called its *physical address*. Permitting programs to access physical memory directly violates the principle of *independence* in that programs may access and modify other programs' data without restriction. It is a principal task of the operating-system kernel to provide this independence.

For the above reason, amongst others, modern-day operating systems and hardware work together to create a *virtual memory* for programs to execute in. Virtual memory is an abstraction of a system's storage (which includes physical memory). When a program executes in virtual memory, it addresses memory by issuing *virtual addresses* instead of physical addresses. A hardware device called the memory-management unit (MMU) lies

between the CPU and memory and has the responsibility of translating these virtual addresses into physical addresses. A *virtual address space* is a set of such virtual to physical mappings.

It is the operating system's responsibility to actually define the virtual to physical translations. These translations occur at the granularity of *pages*. A page is a block of contiguous virtual addresses and is mapped to an equally-sized contiguous block of physical addresses called a *frame*. Depending on the given architecture and operating system, pages may be of fixed or varying size. On architectures that support multiple page sizes, the larger page sizes are often called *superpages*.

Each page in virtual memory is numbered consecutively by a *virtual page number*. Frames in physical memory are similarly numbered by a *physical frame number*. The operating system manages virtual memory by maintaining a data structure called a *page table*. The implementation of page tables varies across operating systems and often depends on the underlying architecture [22]. Despite the variance in implementation, each page-table entry (PTE) typically not only maps a virtual page number to a frame number, but also maintains meta-data such as access rights for that page.

2.1.2 The translation lookaside buffer

A fundamental operation in the virtual memory abstraction is the translation of virtual to physical addresses whenever virtual memory is addressed. A naive implementation of this translation would require a consultation of the page table on every such access. Since the page table itself is contained in memory, this would at the least double the number of memory accesses performed by a process and hence lead to potentially intolerable performance degradation.

To alleviate the overheard in performing address translation, most systems provide a hardware device called the *translation lookaside buffer* (TLB) to hold a subset of address-space mappings. The TLB is a hardware cache of page-table entries, and the operating system's page tables serve as a backing store for that cache.

Upon addressing virtual memory, the MMU first examines the TLB with the appropriate virtual page number in hope of finding a valid mapping to a physical frame number. If found, the MMU hardware validates the access (using protection attributes stored with the TLB translation) and locates the data in physical memory. If the TLB does not contain a valid mapping, a *TLB miss* occurs and the page table must be searched for the correct entry so that it can be placed into the TLB. This operation is referred to as a *TLB refill*.

Depending on the underlying architecture, a TLB refill may be performed either by hardware or software (operating system). Hardware-walked page tables offer faster refill times at the expense of flexibility in page-table structure and simpler hardware design. If the TLB cannot be refilled because the page table contains no valid mapping, a *page fault* is said to have occurred. Page faults are handled by the operating system.

A TLB is said to *cover* a set of virtual addresses if it contains valid mappings for those virtual addresses. The total set of virtual addresses covered by a TLB is referred to as the *TLB's coverage*. Since TLBs only cache a small number of PTEs (typically 32–256 entries) and pages are (typically) only a few kilobytes in size, a TLBs coverage is only a small fraction of the entire virtual address space. Hence to be effective, the TLB's coverage must encompass the working set of currently executing programs so that the costs saved by avoiding memory access (significantly) outbalance the costs incurred by performing TLB refills.

2.1.3 Benefits of virtual memory

Since a primary objective of this thesis is to implement a completely physically-addressed L4 kernel, it is appropriate to describe the benefits of virtual addressing so that later we can later ascertain what impact (if any) the forfeiting of these benefits has for the kernel.

- Virtual memory provides a basis with which an operating system can prevent processes executing in different address spaces from interfering with each other. That is, operating systems may use virtual address spaces to provide independence and security to user-level programs.

- The access right attributes associated with each mapping of a virtual page to a physical frame permits the operating system to protect memory access at page granularity and have this protection enforced by hardware. This makes it possible, for example, to protect programs from modifying their own text segments (by marking them read-only).
- Virtual memory permits user processes to address their data without knowing (or assuming) its actual location in physical memory.
- The operating system may provide shared memory access for processes executing in different address spaces by mapping virtual pages in their respective address spaces to identical physical frames.
- Virtual memory permits the kernel to implement *demand paging* transparently to user processes.
- Virtual memory allows programs to access data in an address space larger than that provided by physical memory, and in fact provides a uniform view of all storage devices in a system that the operating system chooses to map virtual memory to.

2.2 Memory Caches

It turns out that our physically-addressed kernel increases the pressure exerted on a performance-critical component of modern architectures called the memory cache. Much of this thesis is devoted to understanding and evaluating the performance impact this increased pressure implies. To do so, however, we must first understand cache architecture and cache operation on modern architectures.

2.2.1 The memory hierarchy and locality

There is an abundance of devices that a microprocessor can use for storing and retrieving data. These devices can be characterised by their capacity, speed and cost. In an ideal world, storage would be as large, cheap and fast as one desired. Unfortunately in the real world, the aforementioned storage characteristics are diametrically opposed to each other— devices with fast access times, for example, are generally smaller and more expensive than those with slower access times.

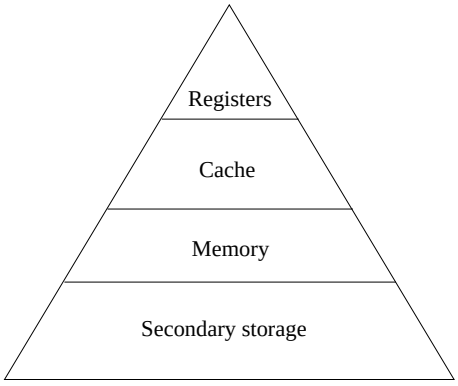

Figure 2.2: A typical memory hierarchy.

The trade-offs amongst the three key characteristics of storage devices presents a dilemma that is solved by organising these storage devices into what is called a *memory hierarchy*. As one ascends the memory hierarchy, access times decrease at the expense of size and cost. The memory hierarchy endeavours to provide a large

storage capacity characteristic of the device at the lowest level of the hierarchy, but at a speed only attainable by devices at the higher levels of the hierarchy. It is the *principle of locality* that makes this goal realisable.

Modern applications exhibit two types of locality: *temporal* locality and *spatial* locality. A program exhibits temporal locality when recently-referenced memory locations are likely to be referenced again in the near future. Spatial locality implies that data nearby recently-referenced memory locations are likely to be soon referenced themselves. When locality is strong, storage devices at the upper levels of the memory hierarchy are more frequently accessed than devices at the lower levels. Because of this, the memory hierarchy is able to create the illusion of a low latency but high capacity storage device being available to computer programs.

2.2.2 The memory cache

Modern computer architectures place a small but high-speed memory called a *cache* between the system's physical memory and the processor. Such a cache forms a critical component of modern memory hierarchies, typically appearing between a processor's register set and main memory. When a processor references main memory, it first checks for the presence of the appropriate data in the memory cache. If the data is found in the cache a *cache hit* is said to have occurred and main memory need not be consulted. Otherwise a *cache miss* has taken place and the cache is (usually) refilled with the appropriate data from main memory.

Cache operation

Caches are organised as a fixed number of *cache sets*. Each cache set contains an identical number of entries called *cache lines*. The structure of cache lines varies between architectures, but at a minimum each cache line contains a fixed amount of cached data, a cache tag and some status bits. The cache tag, together with the encompassing cache set, uniquely identifies the location of the cached data in memory. The status bits typically include a valid bit and a dirty bit. The valid bit indicates whether the cache line holds valid data that may be used by the processor. When a cache line is marked as valid, the dirty bit indicates whether the cached data is consistent with the corresponding data held in main memory.

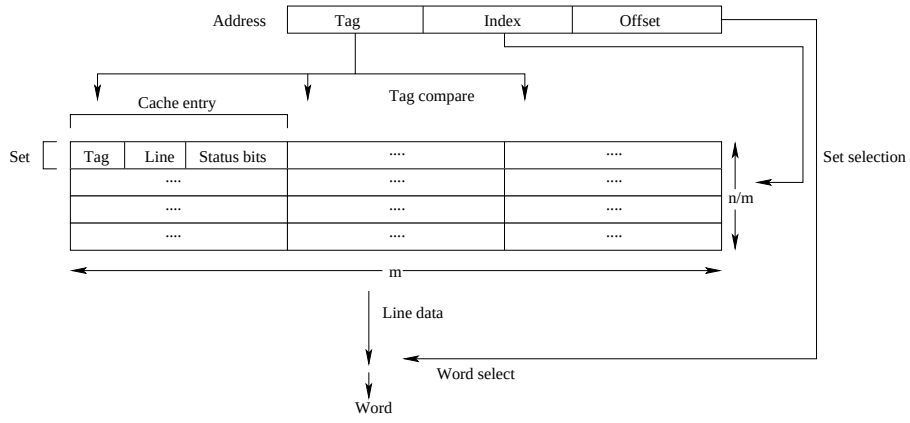

Figure 2.3: The cache-lookup operation.

The process by which data is fetched from a cache is called a *cache lookup* and is depicted in Figure 2.3. The address of the main-memory location being referenced serves as the input to this process. The index bits of the input address index the memory cache to select a cache set. A cache hit occurs precisely when the tag bits of the input address matches against the cache tag of any one the cache lines contained in the selected set, provided that cache line is also marked as valid.

In the special case where a cache contains exactly one cache set, the set-indexing operation becomes trivial and hence the cache-lookup process derives to the task of comparing the tag bits of the input address with the cache tag of every entry in the cache. Such a cache is described as a *fully-associative* cache. At the other extreme, a cache whose cache sets all contain exactly one cache line is known as a *direct-mapped* cache. More generally, an m-way *set-associative* cache is a memory cache where each cache set contains exactly m cache lines. Hence an m-way set-associative cache containing a total of n cache lines is a direct-mapped cache precisely when $m = 1$ and fully associative when $m = n$.

Cache addressing

The address of the main-memory location being referenced serves as the input to the cache-lookup process. The index bits in this address are used to select a cache set, and the tag bits are used to select a cache line from within that set. Thus far we have neglected to specify whether the input address is virtual or physical. In fact not only are both types of address possible, but the index bits and tag bits used in the cache lookup need not both be derived from a physical address, nor need they be both derived from a virtual address. This gives rise to four distinct possibilities — physically-indexed physically-tagged caches, virtually-indexed physically-tagged caches, physically-indexed virtually-tagged caches, and virtually-indexed physically-virtually caches.

The trade-offs associated with each of the four cache-addressing methods are intricate, and for the most part beyond the scope of this thesis. We shall however pay particular attention to virtually-indexed virtually-tagged caches because they share an important property with TLBs. The interested reader can find a more thorough comparison of cache-addressing methods in *UNIX Systems for Modern Architectures* [40].

Virtually-addressed caches

For each valid cache line there is a corresponding memory location whose contents are cached. When a cache is virtually tagged these memory locations are necessarily virtual addresses. In particular, this is true for virtually-indexed virtually-tagged caches — more succinctly referred to simply as *virtually-addressed* caches. Because virtual addresses are always tied to a particular address space, this implies that the contents of a virtually-addressed cache are tied to a particular addressing context.

Virtually-addressed caches are therefore faced with a dilemma when the current addressing context changes (that is, when an address-space switch occurs). The simplest solution to this problem is to perform a *cache flush* on every address-space switch so that the contents of the virtually-addressed cache are written back to main memory and then declared invalid. Cache flushing however is an expensive operation. There are direct costs associated with performing the cache-flush operation itself, and more importantly, potentially crippling indirect costs associated with reestablishing a process's entire cache working set on each address-space switch.

A virtually-addressed cache need not perform a complete flush however if the address space being switched to does not overlap with the address space being switched from. In this special case the contents of the virtually-addressed cache are not made ambiguous on address-space switch and may therefore remain present in the cache provided they are not accessible.

More generally however, a virtually-addressed cache can only avoid a complete cache flush on address-space switch if each valid cache line is identified with the address space it is tied to. For example, if each address space is bestowed a unique numeric *address-space identifier* (ASID) and each cache line is tagged with the ASID of its corresponding address space, then cache entries no longer become ambiguous when the addressing context changes. Hence a virtually-addressed cache tagged with ASIDs need not flush its contents on every address-space switch.

Recall that the translation lookaside buffer introduced in Section 2.1.2 is indexed and tagged by virtual page numbers which are themselves derived from virtual addresses. Hence the cached entries of a translation lookaside buffer are also tied to a particular addressing context. We note that the same mechanisms that may be

used to avoid flushing virtually-addressed caches can, in particular, also be used to avoid flushing the translation lookaside buffer on address-space switch.

Cache misses

Cache misses can be categorised into three different groups [38]:

1. A cache miss is said to be a *compulsory miss* when the cache is not completely full and the new cache entry does not replace an existing valid entry in the cache. Compulsory cache misses, for example, occur immediately after a cache flush has been performed.

2. *Conflict misses* occur when competition between cache lines within a set causes a new cache entry to displace another valid cache line within the same set. The more associative a cache is, the less likely it is for conflict misses to occur. Hence direct-mapped caches exhibit the highest levels of conflict misses.

3. A *capacity miss* is said to have occurred when the sole cause of a cache miss is insufficient cache capacity. Such a cache miss occurs independently of the level of associativity in the cache. In particular it would still occur if the cache were fully associative.

When a cache miss occurs, the cache is generally refilled by obtaining the referenced data from main memory. For a direct-mapped cache there is no dilemma as to which cache line the new cache entry should replace. However caches with higher levels of associativity must choose which cache line in the appropriate cache set to replace. This choice function is governed by the cache's *replacement algorithm* or *replacement policy*. Ideally the replacement algorithm would replace the cache line that is least likely to be used in the near future. Determining such a cache line is in general an impossible task. Nevertheless, feasible algorithms exist that most often perform reasonably effectively. Three common replacement algorithms employed by caches are *first-in first-out* (FIFO), *least-recently-used* (LRU) and *pseudo-random replacement* [43].

Cache write policies

When a processor performs a write to a memory location whose contents are present in the memory cache, it may either write the data into both the cache and physical memory, or it may choose to only write the data into the cache. The former is referred to as a *write-through* cache policy and the latter is referred to as a *write-back* cache policy. When write-back is used, modified cache lines are marked as dirty to identify that their contents are no longer consistent with main memory.

The rationale behind *write-back* is to reduce memory traffic as main memory then only needs to be written to when dirty cache lines are mapped out of the cache. The advantage of write-through, however, is to simplify cache-consistency issues, which is particularly appreciated on multi-processor systems and by I/O modules performing direct memory access [42].

When a processor performs a write to a memory location whose contents are not present in the memory cache, it may choose to either write the data directly to memory but not perform a cache refill, or it may choose to first perform a cache refill and then update the contents of the cache. The former is referred to as a *no-write-allocate* policy and the latter is referred to as a *write-allocate* policy. Write-back caches most often employ a write-allocate policy in hope that subsequent writes will be captured by the cache. Write-through caches however most often employ a no-write-allocate policy since subsequent writes will still need to go to main memory.

2.3 Threads

It turns out that to obtain a physically-addressed L4 microkernel, we must first alter the way in which a class of kernel-maintained data structures called *thread control blocks* are addressed by the kernel. Hence it is appropriate to discuss the use and purpose of these data structures within L4.

2.3.1 Execution abstraction

A *thread* is an abstraction of an execution unit on the CPU. It consists of an *instruction stream* executing within a known *context*. At a minimum a thread's context comprises of the address space in which the thread executes, and the contents of the CPU's registers during its execution. The CPU register set typically consists of an instruction and stack pointer, general purpose registers and system control and status registers.

Most operating-system kernels are responsible for multiplexing the execution of numerous threads on a single CPU. At certain points in time the operating system may perform a *context switch* to change the currently executing thread. A context switch involves the saving of the currently executing thread's context followed by the restoration of a suspended thread's context.

2.3.2 Thread control blocks

An operating-system kernel maintains a per-thread data structure called the *thread control block* (TCB). The thread control block is where a thread's state is saved to or restored from on a context switch. Additionally, the TCB holds thread-specific meta data required for the kernel's management of that thread. Typical TCB fields include:

- a thread identifier that uniquely identifies the thread;
- a characterisation of the thread's current state (e.g. running, suspended);
- execution context (address space and registers);
- scheduling parameters (e.g. thread priority, timeslice, total quantum);
- queues for managing inter-process communication.

Thread control blocks (along with page tables) are perhaps the most important data structures maintained by an operating-system kernel. In fact TCBs implement the thread abstraction for a kernel. Much of this thesis is concerned with investigating and evaluating different methods of addressing TCBs within the context of the L4 microkernel. We have already presented an introduction to addressing and to TCBs. It remains to introduce the L4 microkernel.

2.4 The L4 Microkernel

L4 is a second-generation microkernel based on the principles of minimality, extensibility, and flexibility. It offers all the advantages of the classical microkernel concept without suffering performance degradation symptomatic of the previous generation of microkernels. The L4 project was originally established by Jochen Liedtke in the 1990s and today is actively researched by the L4Ka team at the University of Karlsruhe in collaboration with the DiSy group at the University of New South Wales and the Dresden University of Technology.

Formally, L4 is defined by a platform-independent[1] API and a platform-dependent ABI. Our work is predominantly concerned with L4 as defined by the L4 *eXperimental* Version X.2 kernel API [27] which we hitherto refer to more succinctly as the L4 Version 4 API or the L4 v4 API. L4Ka::Pistachio [26] implements the L4 v4 API and is available on a variety of widely-used architectures including the MIPS64 [14], ARM [24] and IA-32 (Pentium and above) [19] platforms.

In this section we provide a brief introduction to the microkernel concept and identify how L4 distinguishes itself from earlier microkernels. We then proceed to describe the key L4 abstractions and implementational issues that prove relevant to this thesis.

2.4.1 The microkernel concept

Traditionally, the term *kernel* is used to describe the component of the operating system that is either mandatory to all other software, or executes on the underlying architecture at an elevated privileged level [33]. Historically, most operating systems employed a large monolithic kernel. In such systems, all the services provided by the operating system were contained in the kernel itself.

In contrast, the *microkernel* approach aims to minimise the kernel, implementing conventional operating-system services such as file systems, device drivers and even memory management outside the kernel wherever possible. Such services run in user mode and are viewed by the microkernel no differently than any other user-level application.

From the software-engineering point of view, the microkernel concept offers some clear advantages over the classical, large, integrated, monolithic kernel approach. These advantages include (but are not limited to) improved flexibility, extensibility, reliability and security. For an in-depth exposition of the software-technological advantages of microkernels we direct the reader to the literature [33]. More relevant to this thesis are the performance implications of microkernels and the L4 philosophy.

2.4.2 Microkernel performance

Early microkernels such as Amoeba [46], Chorus [39] and Mach [11] were notoriously noted for suffering from excessive performance limitations. For example, Mach on a DEC-Station 5200/200 was found to endure peak degradations of up to 66% when compared to Ultrix running on the same hardware [3], and Mach-based OSF/1 is cited to perform on average at only half the performance level of monolithic OSF/1 [5].

It is well understood that exporting traditional operating-system services as user-level processes running on a microkernel inherently leads to an increased number of user-kernel mode switches and an increased number of address-space switches. Deeper investigation of the performance degradation suffered by early microkernels has, however, highlighted *inter-process communication* (IPC) as the chief cause of their performance overhead [2, 5, 13, 33].

For a microkernel exporting the client-server paradigm, IPC is the fundamental mechanism provided for communicating between different subsystems co-existing on the microkernel. IPC is invoked frequently enough on microkernel-based systems that overall system performance is dependent on IPC performance [29]. Not only

[1] As an exception, we note that the L4 v4 API does distinguish between 32-bit and 64-bit architectures — but never in a processor-specific manner.

must the direct costs of IPC be minimised, but indirect costs in terms of cache footprint must also be considered. Research first conducted by Chen and Bershad [3] and later further analysed by Liedtke [30] demonstrated that up to 73% of the overhead suffered by a user-level Ultrix server running on Mach when compared to native Ultrix was accounted for by IPC-related activities and that 20% of the total system cache misses suffered by user-level Ultrix were caused by user-kernel competition. In particular, the instruction-cache working set of Mach was cited as a principal bottleneck to achieving user-level Ultrix performance comparable to that of monolithic Ultrix.

The response to this performance degradation is what distinguishes the so-called first-generation microkernels from second-generation microkernels. The response adopted by first-generation microkernels such as Mach and Chorus was to reintegrate critical operating-system servers and drivers back into the microkernel [2, 5]. Although this architectural change reduced both IPC overhead and the number of user-level and address-space switches, it sacrificed much of the software-technological benefits promoted by the microkernel movement.

In contrast, second-generation microkernels such as L4 [32], Exokernel [10] and QNX [16] are based on the thesis that efficiency is derived from providing only a minimal set of microkernel abstractions. In particular, the L4 microkernel supplies only three key abstractions in threads, address spaces and IPC and implements a total of only eleven system calls [27]. It offers lean but super-fast IPC that not only provides the foundation for user-level device drivers and memory management, but also performs an order of magnitude faster than earlier counterparts with only 10–20% of the cache footprint [32,34]. Härtig et al. [13] demonstrated that Linux running on top of L4 under AIM multiuser workloads [1] was capable of achieving performance levels within 5–8% of native Linux whereas MkLinux [6] — Linux running on top of a first-generation Mach-derived microkernel — suffered an average performance degradation of 49%.

2.4.3 L4 concepts and abstractions

L4 provides a minimal set of abstractions in threads, address spaces and inter-process communication (IPC) that can be used to construct a wide range of operating-system policies at user level. Because the L4 microkernel serves as a centrepiece for this thesis, we provide a more thorough introduction to these concepts in the following. We pay particular attention to threads, thread identifiers and IPC, as these abstractions especially form the focus of much of our work.

Threads

Threads form the basic execution unit in L4. They are created, manipulated and destroyed via the THREAD-CONTROL system call. Every thread executes within an L4 address space that constitutes its protection domain. A thread may migrate to different address spaces over the course of its life.

Each thread is associated with two special threads, a *pager* and a *scheduler*. A thread's pager is responsible for handling page faults generated by that thread. A thread's scheduler dictates its priority, timeslice length and other scheduling parameters. An optional *exception handler* may be associated with a thread to handle any exceptions it raises.

We note that in L4, hardware interrupts are abstracted as threads to permit the use of user-level device drivers. The occurrence of a hardware interrupt is then represented by an IPC message from the hardware-interrupt thread to a user-level interrupt handler that implements the device driver for that interrupt.

Thread identifiers

Every thread in L4 has both a *global* and *local* thread identifier (thread ID). Global thread IDs are, as the name suggests, unique throughout the entire system. On the other hand, the scope of a local thread ID is limited to that thread's own address space. It is the global thread ID that we will chiefly be interested in. Hence hitherto we shall simply refer to global thread IDs as *thread IDs* unless there is danger of causing confusion with local thread IDs.

A thread ID consists of a single word (hence its size differs on 32-bit and 64-bit architectures), but contains two distinct parts — a thread number and a version number. The upper 18-bits of the thread ID on 32-bit architectures, and the upper 32-bits on 64-bit architectures, encode the thread number. The remaining lower bits of the thread ID word encode the thread's version number.

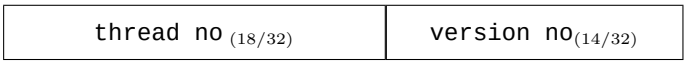

Figure 2.4: A global thread identifier in L4.

At any point in time, at most one thread with a given thread number may exist. The thread version number is assigned by user-level servers (with appropriate privileges) and is not used internally by the L4 kernel except to verify the validity of global thread IDs passed to it from user threads (via system calls). Any implementation of the L4 v4 API may choose an upper limit of thread numbers it supports, provided the limit is one less than a power-of-two and can be encoded into the appropriate upper bits of a thread ID word. The L4 implementation exports this upper limit to user level via the KERNELINTERFACE system call. We call the range of thread numbers made available by an L4 implementation the *thread-number space*.

There are two special thread IDs reserved by the L4 API — `NilThread` and `AnyThread`. The former is guaranteed not to match any thread's identifier and the latter is guaranteed to match every thread's identifier. The `NilThread` is implemented as a word whose bits are all set to zero, and the `AnyThread` identifier is implemented as a word whose bits are all set to one. We will revisit these special thread IDs when discussing L4 IPC in Section 2.4.3.

Address spaces

Because L4 address spaces play a minor role in this thesis, we only provide a brief description of their construction and manipulation. The interested reader can find a more thorough treatment in Liedtke's *On μ-Kernel Construction* [30] and in the L4 *e*X*perimental* Version X.2 reference manual [27] .

L4 provides the mechanism for implementing recursively-defined virtual address spaces that are managed completely at user level. Each address space is defined in terms of at least one parent address space, with the exception of the σ_0 address space that acts as the root address space and represents physical memory. The *map*, *grant* and *unmap* primitives are used to recursively construct L4 address spaces [30]. The map operation maps memory regions called *fpages* [27] from one address space into another. The grant operation is similar but also removes the mapping from the source address space once it has been transferred. The unmap primitive is used to revoke mappings.

Page faults in L4 are abstracted by the microkernel by representing them as special messages delivered via IPC. When a thread generates a page fault, the microkernel fabricates a page-fault IPC message from the faulting thread to its associated pager. The contents of this message include the address of the faulting instruction and the address that was faulted upon. Upon receiving the page-fault message, the pager can respond (via IPC) with an address-space mapping that will permit execution of the faulting thread to continue.

Inter-process communication

Inter-process communication (IPC) is the fundamental mechanism provided by the L4 microkernel for synchronisation and communication between threads. L4 IPC is additionally used to abstract and propagate page faults, hardware interrupts and exception events to user-level servers.

L4 provides message-based, synchronous IPC between threads. Hence an IPC operation in L4 transfers a message from a sending thread to a destination thread if and only if the destination thread has agreed to the exchange by invoking a corresponding IPC operation.

Every thread in L4 owns 64 *virtual message registers*. These registers are mapped to real hardware registers or to memory locations as dictated by the processor-specific L4 ABI [27]. The simplest form of IPC simply transfers a subset of these virtual registers from source to destination. In such a transfer, the data contained in the virtual registers are called *untyped* because the microkernel imposes no semantics on them.

A more complicated form of IPC in L4 transfers *typed* items between threads. Typed items have semantics imposed on them by the microkernel and fall into two distinct categories — those that are used to map and unmap memory regions (fpages) between address spaces, and those that are used to transfer memory buffers. For historic reasons dating back to the Version 2 L4 API (and earlier) [9, 31], IPC involving transfer of memory buffers between address spaces is often referred to as *Long IPC*.

The IPC system-call interface

In theory, synchronous IPC can be provided by a kernel via two system calls — SEND and WAIT — each accepting a thread identifier as the sole parameter. In this case a round-trip send-and-reply IPC message between two threads would require four system calls and in particular four user-kernel mode switches to take place. Instead L4 offers a single IPC system call that accepts two thread identifiers as input — a `to-thread` specifier and a `from-thread` specifier. When a thread invokes this IPC primitive, it sends a message to the thread identified by `to-thread`, and then waits for a message from the thread identified by `from-thread`. In effect, the IPC primitive in L4 combines the logical SEND and WAIT operations into a single IPC system call. This allows a round-trip send-and-reply IPC between two threads to take place with only half the number of user-kernel mode switches.

In the special case where the `to-thread` parameter is `NilThread`, an invocation of the IPC system call only executes the wait phase of the IPC. We term this specific type of IPC a *wait-only* IPC. Likewise, when the `from-thread` specifier is `NilThread`, the IPC system call only executes a send operation, skipping the wait phase. We term these *send-only* IPCs. A thread may also invoke the IPC system call with `AnyThread` as the `from-thread` specifier. This indicates that the thread, once it has completed the send phase of the system call, is willing to accept an IPC message from any L4 thread. We call this special case an *open-wait* IPC and call any other invocation of the IPC system call involving a wait-phase a *closed-wait* IPC[2]. A special case of closed-wait IPCs occurs when the `to-thread` specifier is identical to the `from-thread` specifier — signifying that the source thread expects a reply from the send-phase recipient and is not willing to accept IPC from any other threads. We describe these as *call* IPCs.

We conclude by noting that the L4 IPC system call also accepts a third parameter that can be used to specify (possibly different) timeout constraints for the send and wait phases of an IPC invocation. Two special timeouts are the `Zero` timeout and the `Forever` timeout. As an example of their use, a send-only IPC with a `Zero` timeout will not complete if the destination thread is not already blocked in the wait phase of an IPC invocation. A thread performing a wait-only IPC system call with a `Forever` timeout and the `from-thread` specifier set to its own thread ID will block forever.

User thread control blocks

The L4 API provides a *lazy thread switching* mechanism with which intra-address space IPC may be performed without entering kernel mode [35]. The purpose of this is to avoid the performance penalty of user-kernel mode switches when performing a frequently invoked subset of IPC system calls. This mechanism however requires at a minimum, a thread's user stack pointer and thread status be made visible to other threads executing in its address space. Traditionally, a thread's stack pointer and status are maintained in kernel-protected TCB structures. Since it would be dangerous to make TCBs entirely user accessible, support for lazy switching

[2] In the L4 v4 API, threads are also able to specify that they are only willing to accept IPC messages from threads executing in the same address space as themselves. We choose to ignore this special case as it currently has no implementation in L4Ka::Pistachio.

necessitates dividing each TCB into a *kernel TCB* (KTCB) and a *user TCB* (UTCB). The KTCB is kernel protected whereas the UTCB is made available to user threads (executing in the same address space as the UTCB's owner).

In L4, UTCBs also serve as an efficient way for user-level threads to communicate with the kernel. In particular, a thread's virtual registers that are not mapped to hardware register are actually mapped to memory locations inside its UTCB. Which virtual registers are mapped into the UTCB and at what offset is specified by the L4 processor-specific ABI [27].

It should be mentioned that our work is concerned almost entirely with KTCBs rather than UTCBs. Hence we shall refer to KTCBs simply as TCBs unless there is potential for confusion.

2.4.4 Inside L4Ka::Pistachio

L4Ka::Pistachio is an implementation of the L4 v4 API. Our work is primarily concerned with investigating the trade-offs involved in making the Pistachio kernel completely physically addressed. Recognising and understanding these trade-offs naturally requires an understanding of the current implementation of Pistachio. In this section we highlight a few implementational design choices in Pistachio that prove highly relevant to the remainder of this thesis. An exposition of the current state of addressing in Pistachio is, however, deferred until Chapter 3 where we give it a particularly thorough treatment.

Kernel stacks

A currently-executing user thread may be pre-empted at any time by an interrupt or exception. At this point the microprocessor traps into system code with an elevated privilege called kernel mode. Synchronous events such as system-call invocations may also trap into system code.

For the kernel to execute, it must have its own local stack. In theory only a single kernel stack (per processor) is needed, but for simplicity the Pistachio implementation uses per-thread kernel stacks. Per-thread kernel stacks simplify handling blocking system calls as all the state required to resume a thread's execution is implicitly contained on its own kernel stack. The drawbacks over single kernel stack implementations include increased cache footprint and resource usage.

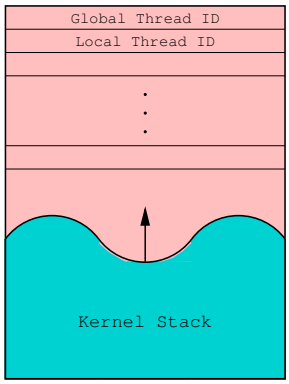

Figure 2.5: Per-thread kernel stacks in L4 reside in each thread's TCB.

In L4Ka::Pistachio, each thread's kernel stack resides entirely in that thread's TCB. This design choice implies that the Pistachio kernel's method of addressing TCBs determines the method used to address the current kernel stack. It also implies that TCBs must be made sufficiently large to hold the entirety of a thread's kernel stack at any point during its execution.

The IPC fastpath

In Section 2.4.2 we noted that for a microkernel, efficient IPC performance is a prerequisite for overall system performance to be fast. For this reason, the Pistachio implementation provides a highly-tuned, processor-specific assembly implementation for a frequently-invoked subset of IPC operations called *fastpath* IPCs. The implementation itself is referred to as the *IPC fastpath*. There are two principal reasons why the existence of an IPC fastpath is crucial for Pistachio:

1. For improved portability, the platform-independent parts of Pistachio are implemented in C++. Even when using the best C++ compilers available, a C++ implementation of the L4 IPC system call cannot compete with a hand-optimised assembler version, both in terms of number of instructions required and in terms of cache pollution.

2. An assembler version of the IPC system call is required to take advantage of a register-only transfer of untyped items (i.e. where the IPC send phase transfers only untyped items contained in virtual registers that are all mapped to processor-specific hardware registers). Assembly is necessary here because register-mapped virtual registers are typically mapped to a processor's general-purpose registers which are prone to being overwritten by instructions generated by high-level languages such as C++. When an immediate context switch to the recipient thread can take place, the assembler fastpath can transfer register-mapped data simply by leaving the registers they are mapped to untouched on context switch. In particular this incurs no copy overhead and is one of the chief factors behind L4's revolutionary IPC performance [34].

As previously mentioned, only a subset of IPC invocations are executed by the fastpath. The remaining IPCs system calls are executed by generic C++ code, aptly called the *IPC slowpath*. The fastpath typically outperforms the slowpath by at a factor of three to five — the precise improvement being highly dependent on the specific platform and the nature of the message transferred.

The criteria used to distinguish between fastpath and slowpath IPCs is also platform dependent. Nevertheless, a primary goal of the IPC fastpath on all architectures is to perform register-only transfer of untyped items without any copying overhead wherever possible. Hence any IPC invocation that involves transfer of typed items or does not involve an immediate context switch to the recipient thread after the send phase has concluded is a candidate for the slowpath.

Long IPC

Long IPC inherently complicates any L4 kernel because the message transfer is performed completely within the execution context of the source thread. If it were otherwise, the kernel would have to copy the message data from the source thread's address space into a temporary internal buffer and then, once switched to the execution context of the destination thread, complete the transfer by copying the data from its internal buffer into the destination thread's address space. Although such an implementation is not overly complicated and even appropriate for asynchronous IPC, the duplicate copying overhead it incurs is needless when IPC in L4 is always performed synchronously.

To avoid buffering long IPC message data, the kernel performs the transfer whilst executing entirely within the source thread's address-space context. To do so, it establishes a temporary mapping from a reserved memory region called the *copy window* in the source thread's address space to the receive buffer in the destination thread's address space. This temporary mapping can be cheaply implemented for example, by copying the appropriate top-level page-table entry in the destination address space's hierarchial page table [8] into the top-level entry in the source address space's page table that corresponds to the copy window. This requires the copy window and hence the maximum buffer size transferred by long IPC to not exceed the size of the virtual memory region mapped by a top-level page-table entry, but has the performance advantage of not requiring a complete page-table traversal.

Chapter 3

Physically-Addressed L4 Kernel

This thesis investigates the trade-offs involved in making the L4Ka::Pistachio implementation of the L4 API completely physically addressed. Ideally, by *completely physically addressed* we would mean that the kernel itself should not use any virtual-memory mechanisms provided by the underlying architecture. However it turns out this definition is too restrictive. In the following section we explain why, and provide a broader alternative.

3.1 Physical Addressing Defined

Physical addressing, in the strict sense, implies by-passing the underlying architecture's virtual-memory mechanisms in order to address a system's main memory directly. In particular it implies by-passing the translation lookaside buffer (TLB). Unfortunately not all architectures reasonably support this style of addressing. For example, although the Intel IA-32 [18] platform provides a status bit in its control register for disabling its virtual memory subsystem, toggling this option incurs a non-negligible performance penalty that cannot be tolerated by a high-performance microkernel on every user-kernel mode switch. A more severe case is the ARM [24] where disabling virtual memory implies forfeiting use of its virtually-addressed memory caches.

Some architectures are kinder. The PowerPC [36] for example automatically disables virtual addressing on user-to-kernel mode switch. A MIPS64 R4x00 [14, 17] processor is more flexible in that it does not disable virtual addressing but instead maps a kernel-reserved 512MB segment of the 64-bit address space it offers directly onto the first 512MB of physical memory.

To ensure the subject matter in this thesis is relevant to a broad spectrum of architectures, we turn to expanding the definition of physical addressing. To this end, throughout this thesis we shall treat virtual addressing in the kernel as physical when there is no possibility of such addressing causing a TLB miss (even on architectures with hardware-walked page tables). In particular this guarantees a page fault or, where appropriate, a TLB-miss exception, cannot occur and hence is consistent with the chief motivating factor in our pursuit of a completely physically-addressed kernel — namely to eliminate such faults from being raised within the kernel.

We now note that, in the sense described above, physical addressing is supported by any architecture where a single sufficiently large superpage that covers all of kernel-accessed memory can be locked permanently into the TLB. In particular, the Intel Itanium [21] and ARM platforms [24] satisfy this condition. Both provide sufficiently large superpages to cover all of kernel space, and both provide support for designating TLB entries as irreplaceable.

The Intel Pentium and its IA-32 successors prove a more difficult case. Although the Pentium supports 4MB superpages, a single 4MB mapping may not be sufficiently large to cover all of kernel-addressed memory on every system. Furthermore there is no mechanism on the Pentium for pinning an entry in the TLB. Nevertheless, because only a small number of 4MB superpages should suffice for all but the largest systems, and because on the Pentium 4MB superpages are guaranteed not to compete for TLB entries with 4KB pages [18], we shall make an exception and deem the Pentium capable of physical addressing.

3.2 Addressing Inside L4Ka::Pistachio

Because obtaining a completely physically-addressed L4 kernel is a principle goal of this thesis, it is natural to start by analysing the current state of addressing in L4. Specifically, we shall describe where physical and virtual addressing are used in the kernel and why this is the case.

3.2.1 Statically allocated data

Physical addressing by a kernel incurs no performance penalty when it can take place without indirection. This in particular holds for anything the kernel references that exists in memory at known, predefined locations. For L4Ka::Pistachio this includes all of its text segment and statically allocated data such as jump tables for system call and interrupt handling, pointers to heads of scheduling queues and structures for ASID management. Such data can be addressed with equal ease both physically and virtually by any kernel. However physically addressing statically allocated data has the benefit of not polluting the TLB.

The performance gained by this reduced TLB footprint has long been studied in the literature. Clark and Emer [4], for example, observed that the VMS kernel on the VAX-11/780 was itself responsible for as much as 70% of total TLB misses when performing only 18% of all memory accesses. In this case the benefits were particularly strong because of poor locality exhibited by the VMS kernel compared to user programs. Although this decreased locality may not hold as severely for the L4 microkernel, a performance incentive nevertheless always exists for physically addressing statically allocated data. Hence it is not surprising that Pistachio chooses to physically address anything in memory whose location is predetermined.

3.2.2 Dynamically allocated data

Dynamically allocated data cannot be addressed physically without indirection. In Pistachio, there are three classes of such data. These are kernel thread control blocks (KTCBs), user thread control blocks (UTCBs) and page tables. Omitted from this list is the mapping database L4 maintains to represent the history of mapping operations used to recursively construct address spaces by user-level servers. This omission has been made because in Pistachio, the mapping database root nodes are augmented onto page-table structures. Hence addressing the mapping database derives to the task of addressing page tables.

For now let us neglect precisely how Pistachio chooses to address KTCBs and just accept there exists some mechanism by which given a thread's identifier, the microkernel can determine the location of that thread's KTCB. It is then not difficult to see that KTCBs can provide the indirection needed to address UTCBs physically. All that is needed is for each thread's KTCB to maintain a physical pointer to that thread's UTCB. Indeed this is how Pistachio addresses UTCBs.

What is unusual is that the Pistachio kernel addresses page tables in exactly the same manner it addresses UTCBs. This is surprising because threads, which are implemented by KTCBs, and address spaces, which are implemented by page tables, are orthogonal objects. What makes it natural, however, is that in the L4 Version 4 API address spaces are *implicit* objects — address spaces are not given identifiers and they are never referred to directly. Instead, address spaces are referred to by providing the thread ID of any thread executing inside that address space. New address spaces are created by providing a thread ID that is not attached to any existing thread (the thread ID in this case actually becomes attached to the first thread created in the new address space). With this subtlety in mind, it becomes natural for any implementation of the L4 v4 API to anchor page-table addresses inside KTCB structures. Since there is no performance penalty in making these anchored addresses physical (and reduced TLB footprint to gain), this is precisely what Pistachio does.

3.2.3 The kernel memory pool

In Section 3.1 we broadened the definition of physical addressing so that our work proves relevant to architectures where, although strict physical addressing is unavailable, a single superpage can be used to cover all of kernel space. For this to be feasible however, kernel-referenced memory in Pistachio must be sufficiently contiguous to fit on a single, but not excessively large, superpage.

Ensuring the kernel's text and statically allocated data is contiguous in memory is trivial. Ensuring the same holds for dynamically allocated data is, in general, much more difficult. However, all dynamic allocations in Pistachio are made from a single contiguous block of memory called the *kernel memory pool*. Hence at worst, a single superpage can be used to map all of the kernel's text and statically allocated data and a second superpage can be used to map the entirety of the kernel memory pool. When the base of the kernel memory pool can be placed close to the kernel's text and statically allocated data, the use of a single superpage suffices.

3.3 Addressing Thread Control Blocks

In Section 2.4.3 we stated that each thread in an L4-based system has a globally unique thread ID consisting of a thread number and a version number. At any instance in time, there is at most one active thread with a given thread number. Hence the task of addressing (kernel) TCBs in L4 derives to the task of determining the address of a thread's TCB given that thread's thread number.

Addressing TCBs in Pistachio proves more difficult than addressing any other data structure. As Section 3.2 showed, this is because all other data structures are either statically allocated or are addressed indirectly through a TCB structure. In this section we shall describe the two competing designs for addressing TCBs.

3.3.1 Direct addressing

The address of a thread's TCB can be calculated *directly* from its thread number if the kernel stores all TCBs in memory in a contiguous array indexed sequentially by thread number. If the kernel supports a space of 2^t thread numbers, this TCB array would occupy $\left(2^t \times \texttt{tcb\_size}\right)$ bytes in memory where `tcb_size` denotes the size of TCB structures and is typically 4096 bytes in Pistachio. All implementations of Pistachio currently provide support for at least 2^{16} thread numbers. Hence an array of TCB structures in Pistachio occupies at least $2^{16} \times 4096$ bytes or 256MB of memory.

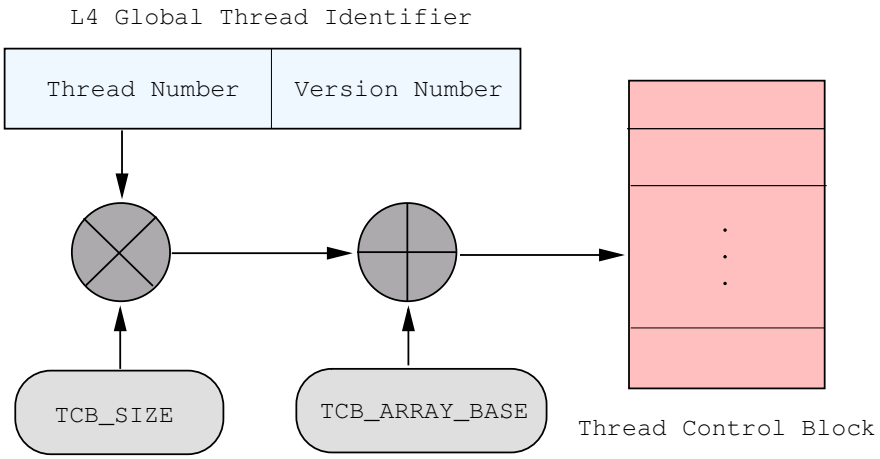

Figure 3.1: Directly addressing thread control blocks.

It is unreasonable to pre-allocate such a large amount of physical memory considering that only a fraction of the thread-number space represents active threads at any one instance in time. Hence addressing TCBs directly from thread numbers is generally only feasible when mapping the TCB array into virtual memory. Exceptions might exist, for example, on highly-specialised embedded systems that execute a small but fixed number of threads throughout their lifetime. We consider such examples as boundary cases and shall hitherto treat direct addressing of TCBs as synonymous with virtual addressing of TCBs.

3.3.2 Indirect addressing

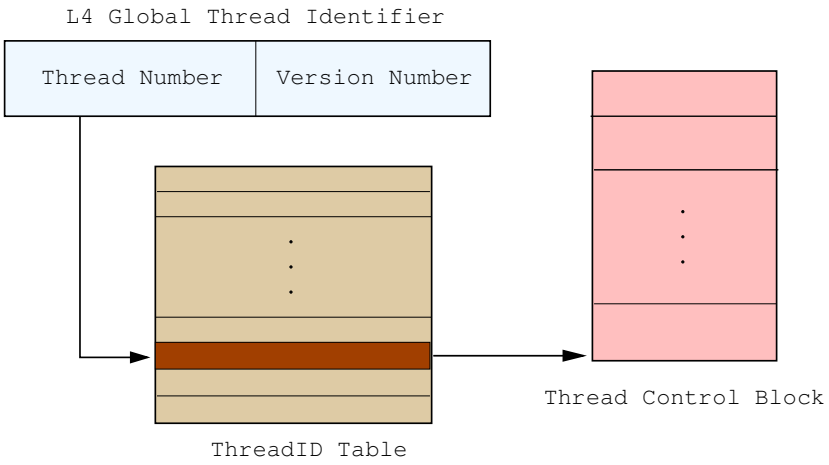

Figure 3.2: Indirectly addressing thread control blocks.

The alternative to direct addressing is for the kernel to *indirectly* map thread numbers to TCB addresses via an auxiliary lookup table indexed by thread number. We call such a table a *threadID table*. It is natural for the threadID table to hold physical rather than virtual TCB addresses — any overhead attached to performing a table lookup when indirectly addressing TCBs is independent of the form of address stored, but using physical addresses implies less TLB footprint. Of course to obtain a completely physically-addressed kernel, the threadID table itself must be physically addressed.

3.3.3 Implementation in L4Ka::Pistachio

All L4 implementations to date, including those that pre-date the Version 4 API, directly address TCBs in virtual memory [25, 26, 28, 31]. L4Ka::Pistachio is no exception. To understand why, we must compare the two methods in the context of the Intel 486 processor — the architecture on which the L4 microkernel was originally implemented by Jochen Liedtke.

The Intel 486 only supports one form of addressing for both privileged and unprivileged execution. Addressing on this architecture is always virtual with all pages restricted to 4KB in size. Hence the Intel 486 is not capable of physical addressing — in both the strict sense and in the broader sense we described in Section 3.1. Because of this, indirectly accessing TCBs through a threadID table cannot be used to reduce the kernel's TLB footprint but still incurs the overhead of additional memory accesses required to consult the table. In fact, indirectly addressing TCBs on the Intel 486 would require additional TLB entries for the threadID table itself.

Hence on the Intel 486 platform, indirectly addressing TCBs offers no performance advantages when compared to direct addressing. It does however offer performance disadvantages in terms of increased memory-cache and TLB-cache footprint for performing threadID-table lookups. It is prudent then for a microkernel on such an

architecture to address TCBs directly as an array in virtual memory. This is precisely the reasoning provided by Liedtke in *Improving IPC by Kernel Design* [29].

What is surprising is that this design choice that was originally made for the advantages it offers the Intel 486, has remained present even in those L4 implementations where the underlying architecture is capable of addressing memory physically. In particular, the L4Ka::Pistachio implementation of L4 addresses TCBs virtually even though every platform it supports is capable of physical addressing. We can only conjecture that this is the case because the performance impact of indirectly physically addressing TCBs on such architectures cannot be determined non-trivially. Beginning with Section 3.6 we shall conduct an investigation into the performance trade-offs involved.

3.4 A Simpler L4 Kernel

In this section we describe the key incentives for a completely physically-addressed L4 kernel. Common amongst all the motivating factors described here is that removing page faults and TLB-miss exceptions from privileged execution leads to a simpler L4 kernel.

3.4.1 Formal verification

An operating-system kernel is the most fundamental software component on all but the most non-trivial systems. Any application implicitly depends on the kernel to execute both correctly and reliably. Furthermore, the kernel is responsible for enforcing all security policies and hence constitutes a part of the trusted computing base on any system. For these reasons, it is a fundamental requirement that operating-system kernels execute correctly, reliably and securely themselves. That these properties hold, however, can only be guaranteed by mathematical proof through *formal verification*. Formal verification models a program using mathematical constructs and then vigourously proves those constructs possess certain desired properties. However, with the currently available tools, formal verification is only feasible when the subject's implementation is not excessively large or complex. In particular monolithic kernels and even first-generation microkernels are not yet suitable for formal verification.

From minimality, L4 not only derives increased efficiency and flexibility, but also enjoys an implementation size in the order of 10,000 lines of C++ and assembler code. Currently, an implementation of this size borders the limits of formal methods. This makes L4 a strong candidate for formal verification, however, similarly-sized implementations that have been mathematically formalised are typically application-level programs. Current formal-methods techniques are not tuned for modelling architecture-specific properties that are inherently intertwined into any kernel implementation. In particular, the presence of non-atomic, non-sequential execution streams significantly complicates the formal-methods process.

When a page fault or TLB-miss exception occurs, the current execution stream is halted and a kernel-supplied routine is invoked to handle the exception. Such interruptions do not impede verification of user-level programs because such programs are always dependent on correct functionality of the operating-system kernel and the kernel's fault handler is required to exit with users' execution context intact. On the other hand, page faults and TLB-miss exceptions generated by the kernel do impede its verification because the fault handler forms part of the kernel.

A completely physically-addressed L4 kernel removes the possibility of page faults and TLB-miss exceptions interrupting the kernel's smooth execution and thus facilitates formal verification. The overwhelming importance of mathematically proving an operating-system kernel functions correctly, reliably and securely makes this the chief motivating factor in our pursuit of a physically-addressed microkernel.

3.4.2 Register trashing

A page fault or TLB-miss handler is required to preserve the execution context of user-level programs. However they are not required to preserve any kernel-specific registers. This includes all registers accessible only in privileged mode, as well as any general-purpose registers reserved for kernel use. On the MIPS64 architecture, an example of the former is the `CP0_BADVADDR` co-processor register which is overwritten with the faulting address whenever a TLB-miss occurs. The `k0` and `k1` registers on the MIPS64 are an example of the latter and are generally used by TLB-miss exception handlers to traverse the kernel's page tables. Any kernel must regard such registers as volatile at any place where a page fault or TLB miss might occur. This makes the kernel implementation more error prone, especially on software-walked page table architectures as TLB misses occur more unpredictably than page faults. A physically-addressed kernel gains simplicity in this respect.

For a concrete example we turn to the L4Ka::Pistachio kernel on the MIPS64. When a program writes to a page present in the TLB but marked as read only, the MIPS64 architecture traps to a generic exception handler that, for Pistachio, begins by saving the user's state on a virtually-addressed stack. Should a nested exception occur here, the `CP0_BADVADDR` register containing the original faulting address will be overwritten. Although the Pistachio kernel addresses this by first saving `CP0_BADVADDR` to a static location in physical memory, it is nevertheless a complication a physically-addressed kernel avoids. As a second example, we merely note that the current IPC-fastpath implementation in the MIPS64 Pistachio kernel does not have complete freedom to use the `k1` register as the execution path is prone to TLB misses whose handler overwrites `k1`. Thus a physically-addressed L4 kernel gains complete use of an additional non-volatile register in the performance-critical IPC fastpath.

3.4.3 ARM exception handling

The simplicity a physically-addressed L4 kernel offers can be architecture specific. We have already noted that the PowerPC architecture automatically disables virtual addressing upon entering kernel mode. Hence current implementations of Pistachio on this architecture must re-enable the virtual memory subsystem on every user-to-kernel mode switch so that TCB structures can be virtually addressed. A physically-addressed L4 kernel clearly avoids this. A more interesting example of architecture-specific simplicity can be found in the ARM architecture.

The ARM offers not one but five different modes of privileged execution that are entered upon by exception events. The use of register banking by these execution modes has the corollary that each such mode is assigned its own dedicated stack pointer (`SP`), link register (`LR`) and saved program status register (`SPSR`). In particular, the stack pointer register used by one privileged mode cannot be accessed when executing in another privileged mode. This would not cause complication if a separate kernel stack was used for each privileged mode, but such a design cannot co-exist with Pistachio's per-thread kernel stack design without significantly increasing memory usage and cache footprint. Using a non-banked register as the kernel stack pointer would work around this problem, but this breaks the ARM C/C++ calling convention and hence is inappropriate for Pistachio which is mostly implemented in C++.

The current implementation of Pistachio addresses the intricacies of the ARM's exception model by trying to execute all kernel code in just the one privileged mode. This privileged mode is the ARM's abort mode which handles memory aborts such as page faults. It becomes the most appropriate privileged mode for exceptions to trap into once defining the L4/ARM system-call convention to simply involve a jump to kernel-reserved memory. Unfortunately if an unmapped TCB is accessed during system-call handling, the resulting page fault causes the processor to re-enter abort mode, trashing the link register in the process. This poses a dilemma for Pistachio on the ARM, that although can be dealt with by manually performing a mode switch from abort mode to an otherwise unused second privileged mode on the system-call path, is nevertheless a non-trivial complexity a physically-addressed L4 kernel avoids.

3.5 A Caveat: Long IPC

Hitherto we have shown that the L4Ka::Pistachio implementation of the L4 API addresses all of its text and data structures physically, with the exception of TCBs which are addressed virtually. Hence it would be natural at this point to assume that modifying Pistachio to address TCBs physically via a threadID table would provide us with a completely physically-addressed kernel. Unfortunately, the long-IPC mechanism introduced in Section 2.4.3 proves a stumbling block.

As described in Section 2.4.4, Pistachio kernels implement long IPC by establishing a temporary mapping between a copy window in the sender's address space and the receiving buffer. This allows the L4 kernel to carry out the transfer completely within the execution context of the source thread and hence removes the need for any internal buffering by the kernel. Naturally, virtual addressing is required here and this implies potential page faults. Any page faults raised during a cross-address-space copy will pre-empt the transfer and must be tunnelled to the recipient's pager even though the kernel is still executing in the sender's context. Although timeouts in L4 can be used to ensure a source thread cannot be blocked indefinitely by a malicious or malfunctioning recipient-thread pager, it has been well established in the literature that timeouts cannot prevent all denial-of-service attacks, and furthermore they limit predictability and testability of a system [41].

The complexity inherent in dealing with page faults during long IPC runs counter to the motivation driving our quest for a completely physically-addressed kernel. In particular, current formal-verification methods are not equipped to faithfully model such a kernel. Physical addressing, and only in the strictest sense, can be used to simplify long-IPC design only if the recipient's page tables are fully traversed on every cross-address-space long IPC. However such an implementation is sub-optimal and unlikely to be tolerated. For these reasons, combined with the fact that the presence of long IPC in L4's future is becoming increasingly doubtful due to an increasing number of concerns raised by the L4 community, we feel justified in disregarding long IPC in our pursuit of a completely physically-addressed kernel.

3.6 A Comparison of Thread-Control-Block Addressing Methods

We now turn to performing a comparison of the two methods of addressing TCBs introduced in Section 3.3. Although much of this thesis is devoted to investigating the performance trade-offs associated with physically addressing TCBs, we shall first begin by discussing the non-performance trade-offs.

3.6.1 Non-performance trade-offs

Virtual-memory usage

We have already presented the most significant advantage of addressing TCBs physically in Section 3.4 — it provides us with a completely physically-addressed kernel that provides benefits in increased simplicity. However there is a further significant non-performance advantage for some architectures which we now describe.

To directly address TCB structures, all TCBs must be stored in a contiguous array that can be indexed by thread number. In Section 3.3.1 we argued that this array must be placed in virtual memory because its sheer size makes allocating it in physical memory simply infeasible. However, we should note that virtual memory is not a limitless resource, and in some circumstances, its usage must be managed especially carefully. Certainly this is not the case for 64-bit architectures where address spaces are vast and a virtually-mapped TCB array consumes only a fraction of all virtual memory. However virtual-memory usage does become an issue on 32-bit architectures where TLB entries cannot be tagged in some way that distinguishes kernel space from user address spaces. Of the architectures L4Ka::Pistachio is currently implemented on, the Intel IA-32 best demonstrates this limitation.

The Intel IA-32 provides no means of associating TLB entries to any address space other than the current one. Hence on every address-space switch, the IA-32 TLB must be completely flushed. In fact this flushing is automatically performed by hardware whenever the page directory register is written to with a new address space's page table (the IA-32 features hardware-walked page tables). Thus on the IA-32, kernel memory must be mapped into all user address spaces to avoid a TLB flush on every user-kernel mode switch — an intolerable performance penalty particularly for microkernel-based systems where privilege-mode switches are frequent.

Currently, the Pistachio microkernel on the IA-32 reserves 512MB of each user address space for its own use. Even when the kernel memory pool is as large as 32MB, over 99% of kernel space is still used purely for mapping a TCB array that holds roughly 2^{17} TCBs, each 4KB in size (the precise number of TCBs being dependent on the configured size of the kernel memory pool). By physically addressing TCBs, an L4/IA-32 kernel need not consume such a large portion of each user address space — only the kernel's text, statically allocated data (including the threadID table) and kernel memory pool would need to be mapped.

Hardware support for virtual memory

In the following we list some potential advantages a kernel that virtually addresses TCBs kernel may enjoy by utilising hardware support for virtual memory. A physically-addressed kernel must forfeit these advantages.

- Liedtke argues that virtually addressing TCBs allows for an efficient hardware-enforced lock to be placed on a thread's TCB simply by unmapping that TCB's virtual page mapping [29]. We note however that no implementations of L4 to date use this technique.
- Virtual memory is not a requirement for paging kernel memory. Nevertheless, virtual memory does allow paging of kernel TCBs to be performed transparently to other kernel subsystems.
- Virtual memory provides the means to implement an efficient copy-on-write policy for TCBs. This proves useful, for example, for ensuring the consistency of TCBs during a kernel-managed checkpoint without incurring the intolerable overhead inherent in copying every allocated TCB on checkpoint commencement.
- Virtual addressing facilitates management of kernel memory by user-level servers [12]. The incentive here is to separate mechanism from policy — a corner stone of the microkernel philosophy.

3.6.2 Performance trade-offs

Understanding and evaluating the performance effect of physically addressing TCB structures is the chief objective of this thesis. To this end, we provide in the following, a qualitative description of the relevant performance trade-offs at hand. Beginning with Chapter 4 we shall seek to quantify the precise impact of these trade-offs in the context of the MIPS64 architecture.

Cost of translating thread identifiers

A virtually-addressed kernel translates thread numbers to TCB addresses simply by performing a shift and an addition. Whether a physically-addressed kernel needs to execute more or less instructions to locate TCBs depends on the precise implementation of the threadID table. Using a hash table with a non-trivial hash function, for example, will certainly require more than just a simple shift-and-add operation. The number of instructions required not only affects cycle counts, but also affects instruction-cache footprint.

Data cache and TLB footprint

Ultimately the L4 kernel needs thread numbers translated into physical TCB addresses. The kernel can perform this in one step via the threadID table introduced in Section 3.3.2. On the other hand, instead of maintaining a threadID table, the kernel can cheaply compute a virtual address for each thread number and use the pre-existing address-space translation mechanisms to translate that virtual address into a physical address. This is the technique discussed in Section 3.3.1.

When viewing the two competing TCB addressing methods from this perspective it is clear that both techniques at some point involve translation via a table. Whether this table is an explicitly-consulted threadID table or a transparently-handled page table depends on whether the kernel addresses TCBs physically or virtually. The key difference to performance is the presence of a hardware cache (the TLB) dedicated to caching page-table entries when no such dedicated hardware cache exists for the threadID table. A less significant difference is that indexing a page-table structure is potentially more involved than indexing a threadID table because thread-number spaces are typically smaller than the space of all valid virtual page numbers (particularly on 64-bit platforms). But because the cost of traversing the page table is included in the cost of a TLB-refill operation, we can succinctly summarise the cache and TLB footprint trade-offs associated with physically addressing TCBs as follows:

- Direct costs incurred by reading additional cache lines from the threadID table.
- Indirect costs incurred by displacing otherwise useful data cache lines.
- Direct costs saved by not incurring TLB-refill penalties.
- Indirect costs saved by not displacing otherwise useful TLB entries.

Compact addresses

On 64-bit architectures that support physical addressing of memory, it may be possible to store physical addresses in a 32-bit format that can be sign extended into valid 64-bit addresses. The MIPS R4700 processor provides an example of an architecture where this address compaction is possible. The R4700 translates 64-bit virtual addresses in the range 0xFFFFFFFF80000000 – 0xFFFFFFFF80000000 by simply masking out the upper 35 bits. Addresses in this range are thus directly mapped onto the first 512MB of physical memory, by-passing the TLB.

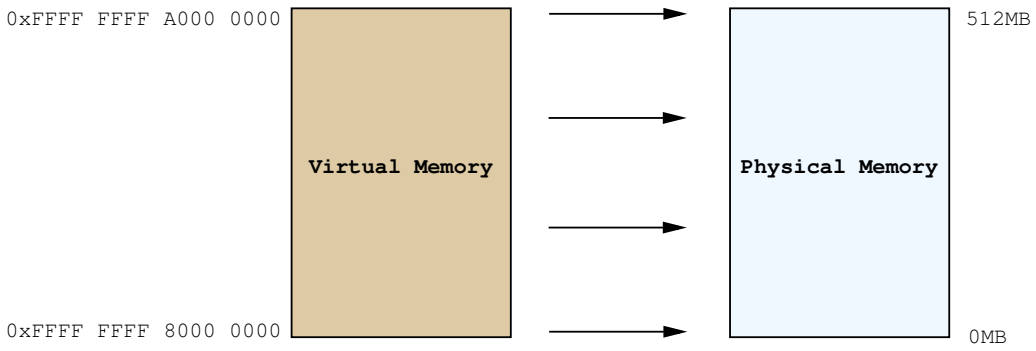

Figure 3.3: Physical addressing on the MIPS R4700.

On this processor, a physically-addressed L4 kernel can store TCB addresses in only half the space by simply truncating the 64-bit addresses to their lowest 32 bits. Because the upper bit of these 32-bit compressed addresses

are always set, and because the MIPS64 instruction set offers instructions that automatically sign extend 32-bit values loaded from memory, this design choice incurs no performance penalty.

For L4Ka::Pistachio, each TCB maintains one pointer representing the top of the kernel stack residing in that TCB, and another six pointers to other TCB structures that represent scheduling and IPC-related queues. On 64-bit architectures, Pistachio stores all these pointers as 64-bit words. A physically-addressed kernel on such an architecture might thus be able to compress these seven addresses to 32-bit values in hope of eliminating a cache line (or two) of cache footprint from the kernel's critical paths. Unfortunately, for current 64-bit implementations of Pistachio (including the MIPS64 implementation), compressing these seven addresses does not reduce the IPC fastpath's cache usage. However, there is nevertheless the prospect of this technique proving useful should Pistachio's TCB layout ever change (for example, due to changes in the evolving L4 API).

Before concluding this section, we should note that an L4 kernel that virtually addresses TCBs may still represent its internal TCB addresses in a more compact form. For example, if the kernel supports a thread-number space no greater than 2^{16} in size, any internal address that points to the start of a TCB structure may simply be represented by a 2-byte thread number. However, address compression in this case is not without drawbacks — the thread-number space is limited in size and arithmetic operations must be used to decompress the shortened addresses.

Efficiency in simplicity

In Section 3.4 we argued that a physically-addressed kernel gains simplicity by not having to deal with page faults or TLB-miss exceptions during privileged execution. This simplicity can at times also lead to greater efficiency. For example, we described in Section 3.4.3 how the possibility of kernel-generated page faults requires the current Pistachio ARM kernel to manually change execution modes when handling system calls. This in particular adds overhead to the ARM's IPC fastpath. The performance penalty involved depends on the precise model of ARM processor. For the Intel XScale [20], the penalty has been measured at 5 cycles, which accounts for 3% of total overhead when executing a best-case fastpath IPC.

Similarly, when the hardware's TLB is managed by software, a completely physically-addressed kernel may enjoy a more efficient TLB-miss exception handler. This holds true on architectures where current Pistachio kernels maintain a separate address space for the virtual TCB array. On these architectures, the TLB-miss exception handler must first determine whether the miss was generated from kernel or user mode, so that the appropriate page table can be selected for traversal. For the MIPS64 Pistachio kernel, this overhead translates to a 6-cycle penalty whenever a TLB miss demands the kernel's page tables be consulted.

Hardware support for virtual memory

Hardware support for virtual memory provides a kernel that addresses TCBs virtually with an optimistic rather than fundamental method of accessing TCB structures. It turns out, however, that this does not translate into any notable performance advantage for kernels that address TCBs directly in virtual memory. In fact the L4Ka::Pistachio kernel only takes advantage of hardware support for virtual memory when addressing TCBs for determining if a given thread number corresponds to a valid, allocated TCB structure. As part of the following section, we describe this process in detail and show that a physically-addressed kernel can implement an equally efficient means of validating thread numbers.

3.7 Translating Thread Identifiers in L4Ka::Pistachio

In Section 3.6.2 we determined that a physically-addressed L4 kernel incurs a performance penalty whenever it performs a threadID-table lookup. Hence it is appropriate to investigate exactly when thread numbers need to be translated into TCB addresses in Pistachio.

3.7.1 Validating thread identifiers

The kernel needs to verify that thread IDs passed to it from user-level servers (via system calls) correspond to valid threads. Recall from Section 2.4.3 that thread IDs in L4 are actually composed of two distinct parts — a thread number that is globally unique and a version number that the kernel attaches no semantics to. With this in mind, validating L4 thread IDs requires two conditions to hold:

1. There must exist a valid mapping from the thread-number component of the given thread ID to an allocated TCB. These mappings are maintained in the threadID table or the kernel's page tables depending on the TCB-addressing method.

2. When the thread number maps to a valid TCB, the version number maintained in that TCB must match with the version-number component of the given thread ID.

The dummy thread control block

Current implementations of the L4 kernel use the underlying architecture's virtual memory subsystem in conjunction with a zero-filled *dummy TCB* to elegantly verify that both of these two conditions hold with only a single check. When a page fault occurs on a TCB, the kernel's page-fault handler first determines if the fault resulted from a read or a write operation. If a write caused the fault, then the page-fault handler simply allocates a new TCB and maps it at the faulting address. If a read access causes a TCB fault however, the page-fault handler maps the faulting address to the dummy TCB. Consequently, current L4 implementations may validate a given thread identifier simply by naively comparing a given thread identifier with the (global) thread-identifier field in the virtually-addressed TCB obtained from the direct computation illustrated in Figure 3.1.

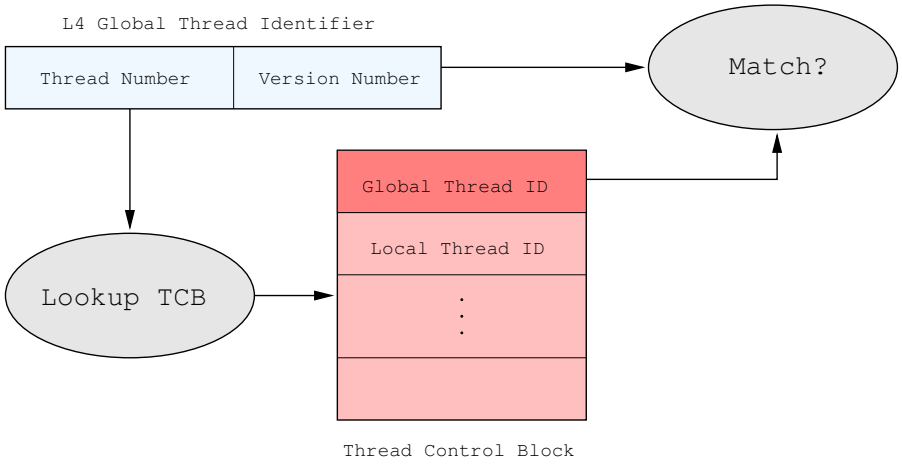

Figure 3.4: Validating thread identifiers in L4.

In the optimistic case where the computed TCB address has a valid virtual mapping (that is, the first of the two required conditions holds), this check will certainly validate the thread identifier correctly. In the unlikely case that the computed TCB address has no virtual mapping, the resulting read page fault will cause the zero-filled dummy TCB to be mapped at that address. In this case, the given thread identifier will be compared with zero which the L4 API guarantees will mismatch (see Section 2.4.3).

We conclude this section by noting that addressing TCBs virtually is not required to use the dummy-TCB trick to validate thread identifiers with just a single check. In particular a physically-addressed L4 kernel can use the same technique by simply initialising the threadID table to map every thread number to the dummy TCB. We will show how this can be done space efficiently for some common threadID-table data structures in Chapter 4.

3.7.2 System calls

Any thread identifier passed to an L4 kernel from user-level threads via system calls must be validated by the kernel. Given the validation process depicted in Figure 3.4, this implies that an L4 kernel must perform a TCB lookup in every system call that accepts a thread identifier as input. A list of these system calls is provided in Figure 3.5. No system calls other than these perform TCB lookups. This includes the LIPC system call which is executed at user level and hence does not address kernel TCBs [35].

IPC	THREADCONTROL	EXCHANGEREGISTERS	THREADSWITCH	SCHEDULE

Figure 3.5: L4 system calls that perform thread control block lookups.

Of course these system calls almost always use the TCBs they lookup for more than just validating thread numbers. For example the SCHEDULE system call may adjust the scheduling-priority field in a TCB and the THREADCONTROL system call might set the pager field in a thread's TCB.

The IPC system call deserves special attention. As described in Section 2.4, IPC is the most fundamental mechanism provided by a microkernel exporting the client-server paradigm. In an L4-based system, the IPC system call is typically invoked at least two orders of magnitude more frequently than any other system call. Hence in evaluating the performance overhead a physically-addressed kernel potentially suffers by performing threadID-table lookups, we must focus particularly on the impact it has on the IPC primitive. The impact on the remaining system calls will be negligible in comparison in all but the most extreme boundary cases.

3.7.3 Locating the current thread control block

Section 2.4.4 described how for each thread, L4 maintains a separate kernel stack contained entirely within that thread's TCB. This design choice allows the kernel to efficiently locate the base address of the current thread's TCB by simply performing a mask operation on the kernel's stack pointer. This works because TCB structures are always power-of-two sized and allocated aligned on their size. Hence masking out an appropriate number of lower-order bits in the kernel stack pointer computes the base address of the TCB structure that stack resides in.

A physically-addressed kernel can thus locate the current thread's TCB without consulting the threadID table. On the other hand, when virtually addressing TCBs, although exploiting the kernel stack saves one or two of the arithmetic operations depicted in Figure 3.1, a TLB entry is still used for the current thread's TCB. Hence this trick is much more beneficial to a physically-addressed kernel than it is to a kernel that addresses TCBs virtually.

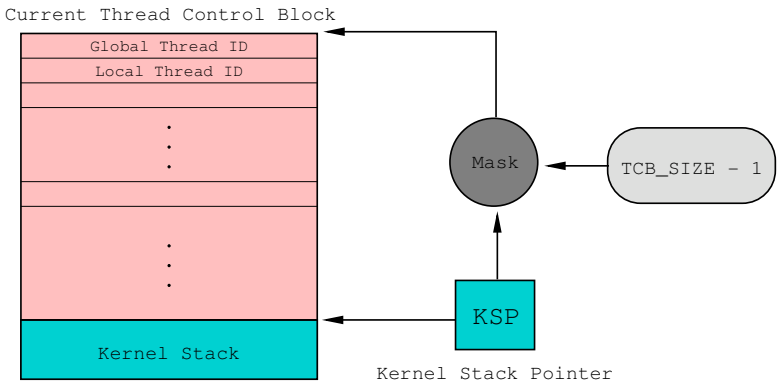

Figure 3.6: Locating the current thread's thread control block.

Chapter 4

Design & Implementation

A physically-addressed kernel offers advantages in simplicity. To obtain such a kernel we need to physically address TCBs using indirection provided by a threadID table. This change however has subtle performance trade-offs that in particular affect the performance-critical IPC path. Naturally, the overall effect of these trade-offs depends upon the precise implementation of the threadID table. However, properties of the underlying architecture are also influential. For example, TLB and memory-cache properties will affect both the frequency and severity of TLB and cache misses. Hence it is appropriate that our evaluation of these trade-offs becomes heavily architecture dependent.

In this chapter we begin investigating the effect a physically-addressed kernel has on performance in the context of the MIPS64 architecture. We pay particular attention to the impact on the IPC fastpath and introduce a number of design choices for the threadID table that aim to reduce the potential performance penalty a physically-addressed kernel incurs in increased cache footprint. We conclude with a summary of various implementations that we proceed to rigourously evaluate in the following chapter.

4.1 Introduction to the MIPS R4700

Instruction-set overview

The MIPS R4700 [14, 17] is a RISC architecture featuring 64-bit integer and floating-point operations. The register set consists of thirty-two general purpose 64-bit integer registers and thirty-two 64-bit floating-point registers. One general-purpose register is hardwired to a value of zero, and another two registers, `k0` and `k1`, are reserved for kernel use.

Register	Convention	Register	Convention
`zero`	Always zero	`AT`	Assembler temporary
`v0`–`v1`	Integer results	`a0`–`a7`	Integer arguments
`t0`–`t3`	Callee saved	`s0`–`s7`	Caller saved
`t8`–`t9`	Callee saved	`k0`–`k1`	Kernel reserved
`gp`	Global pointer	`sp`	Stack pointer
`s8`/`fp`	Frame pointer	`ra`	Return address

Table 4.1: MIPS64 general-purpose register set.

The R4700 features a five-stage pipeline that issues one instruction per clock cycle except when the pipeline stalls. Most instructions complete in just the one clock cycle. Arithmetic instructions operate on either three

registers or two registers with a 16-bit immediate. Multiplication and division instructions require between 10 and 133 cycles to complete. For these instructions, the pipeline stalls until the results become available, at which time they are placed into two special registers, `HI` and `LO`.

The instruction following a branch instruction is always executed whilst the target instruction is still being fetched by the processor. The branch itself does not take effect until after this instruction has been completed. The instruction following a branch is thus often referred to as being in a *branch delay slot*.

Access to external memory is performed by using load and store instructions. Both signed and unsigned 8-bit, 16-bit, 32-bit and 64-bit values may be fetched from or written to memory, but only on natural alignment boundaries. Unaligned access causes an exception to be raised. All addressing is performed using a single register with a signed 16-bit immediate that acts as displacement. The instruction immediately following a load instruction is said to be in a *load delay slot*. If an instruction in a load delay slot references the data fetched by the preceding load instruction, the R4700 stalls the pipeline for one cycle.

Co-processors

The R4700 processor has a number of co-processor units, each featuring its own set of registers. Co-processor zero (`CP0`) is the system co-processor and contains registers pertaining to memory management and exception handling. The status register is also located on `CP0`. Co-processor one (`CP1`), when present, implements a floating-point unit (FPU). Further co-processors are optional.

Translation lookaside buffer

The MIPS R4700 features a fully-associative, software-loaded TLB with 48 entries, each mapping an even-odd pair of virtual pages and tagged with an 8-bit address-space identifier (ASID). The page size is per-entry configurable and varies from 4KB to 16MB in powers of four. The TLB is responsible for caching both instruction and data page translations and is thus sometimes referred to as a joint TLB (JTLB). TLB misses are handled by a specialised TLB-refill exception. All other TLB-related exceptions, such as a write to a read-only page, are handled by the common exception handler. The refill handler may explicitly select which TLB entry to replace, or it may opt to use a pseudo-random replacement mechanism provided by hardware. The `tlbwi` instruction implements the former and the `tlbwr` instruction implements the latter. The `CP0_WIRED` co-processor register may be used to restrict which TLB entries can be replaced by the `tlbwr` instruction.

Memory caches

The MIPS R4700 features a primary 16KB instruction cache and a 16KB data cache on chip. Both caches use a 32-byte line size and are two-way set associative with FIFO replacement within a set. Hence both caches consist of precisely 256 sets. We number these sets consecutively, beginning with zero, so that they can be conveniently identified henceforth.

Both the data cache and instruction cache are virtually indexed and physically tagged. The presence of an external secondary cache is optional. The write policy used is configurable on a per-page basis. The combinations of write-back write-allocate, write-through write-allocate, and write-through no-write-allocate are available. Each virtual page may also be configured to be mapped uncached.

Address-space layout

The MIPS R4700 provides a 64-bit address space that is divided into a number of regions. Depending on the mode of execution, only some regions may be addressable. There are three such modes on the R4700: *user*, *supervisor* and *kernel*. The regions accessible in supervisor mode form a subset of those accessible in kernel mode, and a superset of those accessible in user mode. Each virtual memory region is translated either *mapped*

or *unmapped*. Mapped regions are translated by the TLB and unmapped regions are translated by masking out the most significant bits of the virtual address. Unmapped regions are only addressable in kernel mode.

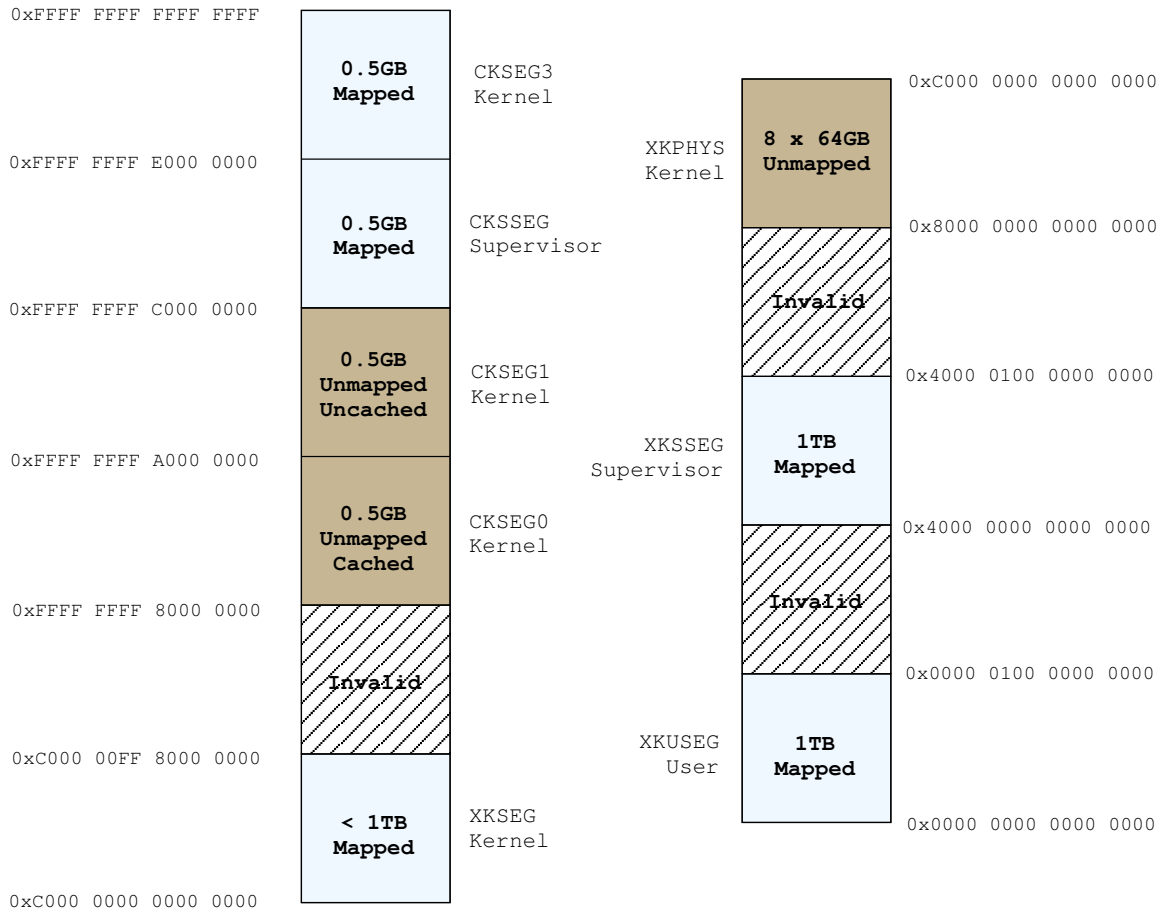

Figure 4.1: Address-space layout on the MIPS R4700.

Figure 4.1 depicts the address-space layout of the MIPS R4700. The `XKUSEG` segment effectively provides a 40GB virtual address space to user-level programs. The supervisor segment, `XKSSEG`, is used by L4Ka::Pistachio to map the virtual TCB array. The `CKSEG0` and `CKSEG1` regions are unmapped and can be used to physically address the first 512MB of memory, the latter of which by-passes the system's data cache. The write policy used when addressing the `CKSEG0` segment is specified by the `CP0_CONFIG` co-processor register.

4.2 The MIPS64 IPC Fastpath

Evaluating the performance impact of a physically-addressed kernel demands particular attention be paid to the performance-critical IPC primitive. Recall from Section 2.4.4 that the L4Ka::Pistachio implementation of the L4 API features a highly-optimised, assembler-written IPC fastpath that executes a subset of all IPC invocations. The remaining IPC system calls are handled by a C++ routine called the IPC slowpath. The IPC slowpath on the MIPS64 offers at a minimum, a four-fold increase in cycle count and a three-fold increase in data-cache footprint when compared to the fastpath. Because the fastpath is designed to handle the most frequently invoked IPC operations, and because the slowpath is sufficiently inefficient that the subtle performance trade-offs associated

with a physically-addressed kernel are most likely to have a negligible net effect on it, our analysis focuses exclusively on the former.

4.2.1 Criteria

The criteria used to distinguish between the IPC fastpath and slowpath varies across the different architectures the Pistachio microkernel is implemented on. The following lists the entry criteria for the MIPS64 IPC fastpath:

1. The IPC must non-trivially contain both a send phase and a receive phase. Equivalently stated, neither the `to-thread` or `from-thread` parameters to the L4 IPC system call may be `NilThread`.
2. The timeout specified for the receive phase must be infinite.
3. For a non-call IPC invocation, the thread performing the IPC system call must not be polled by any other thread. That is, no thread must be trying to send an IPC message to the thread initiating the IPC system call.
4. The recipient of the send phase must already be blocked and waiting for an IPC from the send-phase source thread.
5. The message transferred by the IPC operation must not contain any typed items.

To understand the rationale behind this criteria, recall from Section 2.4.4 that the key objective of the fastpath is to transfer hardware-register-mapped virtual registers by simply leaving those hardware registers untouched on context switch. This avoids copying overhead but requires an immediate context switch to the recipient thread after the send phase has completed. That condition 4 is necessary for this context switch to immediately occur is clear. Conditions 1 and 2 are additionally required to guarantee that the source thread will always block after the send phase has completed. Otherwise a context switch to the destination thread can only take place immediately if the destination thread has equal or higher priority than the source thread. Condition 3 also guarantees that when the IPC is a non-call invocation, switching to a thread polling the IPC system-call invoker instead of switching to `to-thread` after the send phase has completed is never an option. Hence conditions 1–4 above impose sufficient (but not necessary) requirements that ensure a context switch to the send-phase target can take place immediately on completion of the message transfer.

Of the sixty-four virtual message registers attached to each thread, nine are mapped to general-purpose registers on the MIPS64 (`v1` and `s0`–`s7`). The remaining fifty-five virtual message registers are mapped to memory locations in each thread's UTCB, beginning at offset 200. We now note that the final condition of the fastpath criteria only specifies that typed items must be absent from the transfer. Hence the MIPS64 fastpath does handle transfer of memory-mapped untyped items (by performing a copy between the source and recipient threads' UTCBs). But nevertheless, the transfer of the hardware-register-mapped untyped items always takes place without copying.

4.2.2 Data-cache footprint

Addressing TCBs physically increases the cache footprint of the IPC fastpath because a lookup table must be consulted every time a TCB (other than the currently executing thread's TCB) is located. Hence it is appropriate to investigate the current data-cache footprint of Pistachio on the MIPS64 so that we can better judge what effect additional threadID-table cache lines will have on performance.

There are four categories of data referenced by the MIPS64 IPC fastpath. The first is for referencing static data, the second is for referencing KTCB fields, the third contains those cache lines hit by the kernel stack, and the fourth is for referencing UTCB fields.

Static data

There is only one cache line used for referencing static data. It is used in fetching the end address of the currently executing thread's TCB from a fixed location in physical memory called K_STACK_BOTTOM. The kernel updates K_STACK_BOTTOM on every thread switch so that threads can setup their kernel stack on entering kernel mode. The cache line occupied by K_STACK_BOTTOM is aligned to fall on the third cache set in the R4700's data cache.

TCB fields

The IPC fastpath always accesses two TCBs — namely the TCB of the currently executing thread (i.e. the thread that invoked the IPC system call) and the TCB of the thread that acts as the recipient of the send phase (i.e. the thread specified by the to-thread parameter of the IPC system call). If a closed-wait, non-call IPC is being performed, then a third TCB belonging to from-thread is also accessed.

For each TCB accessed, two cache lines worth of data are acquired by referencing TCB fields. These two cache lines are in fact the very first two cache lines in each TCB structure. The TCB fields falling on these cache lines are summarised in Table 4.2 below:

TCB Field	Description
myself_global	This field contains the global thread identifier the TCB corresponds to. It is used for validating to-thread and from-thread and for determining if to-thread is waiting for an IPC from the current thread.
myself_local	In L4Ka::Pistachio, a thread's local identifier is always set to that thread's user-accessible UTCB address. A MIPS64 kernel stores a thread's UTCB address in the k0 when it is dispatched for the convenience of user-level servers.
utcb	This field maintains a physical pointer to the thread's UTCB and is only referenced when transferring memory-mapped virtual registers.
space	This field maintains a physical pointer to the page table backing the address space the thread executes in. On every thread switch, the kernel copies this pointer into the upper bits of the CP0_CONTEXT register for the convenience of exception handlers.
stack	On every thread switch, the thread being switched from has its kernel stack pointer saved in this field so that it can be restored next time it is dispatched.
asid	On every thread switch, the ASID of the address-space that the newly-dispatched thread executes in is obtained from this field and placed into co-processor register CP0_ENTRYHI so that it may be matched against TLB-entry tags by the hardware.
thread_state	This field represents the thread's execution state and must be modified when performing a thread switch. It is also used by the fastpath to determine if to-thread is in a blocked state, waiting to receive an IPC.
partner	When thread_state indicates that a thread is waiting for an IPC, this field specifies which thread (possibly AnyThread) it is willing to accept messages from. It is used to determine if to-thread is waiting for an IPC from the current thread.
send_head	The send_head field in a thread's TCB maintains a linked list of other threads who are currently polling it. It is used to determine if from-thread is not polling the current thread when handling a closed-wait, non-call IPC.

Table 4.2: Thread-control-block fields referenced by the MIPS64 IPC fastpath

Kernel stack

The IPC fastpath involves a context switch from the thread that invoked the L4 IPC system call to the recipient of the send phase. Hence on entering the system-call handler, the kernel must save enough of the current thread's user state to be able to resume it next time it is dispatched. The kernel saves this state on the current thread's kernel stack in what is called an *exception frame*.

For the IPC fastpath, the exception frame consumes two data cache lines and saves five registers: `CP0_EPC`, `CP0_STATUS`, `sp`, `fp` and `ra`. The first two are system co-processor registers containing the user program counter and status register and are required to resume the current thread's user-level execution. The last three are general-purpose registers that are overwritten by the IPC fastpath but must be preserved in accordance with the MIPS64 L4 v4 ABI. The remaining general-purpose registers (other than `zero`) are specified by the ABI as being in an undefined state after the L4 IPC system call returns and hence do not need to be preserved.

Similarly, upon switching to the send-phase recipient thread, the kernel must restore that thread's execution state from the exception frame previously stored on its kernel stack. Hence the IPC fastpath requires two cache lines of exception-frame data for each of the two participants of the send phase.

The IPC fastpath must also establish a *switch frame* on the kernel stack of the system-call invoker before it blocks. This switch frame covers only one cache line and contains a pointer to an assembly routine called `ipc_finish` that first loads the `v1` and `s0`–`s7` registers with data obtained from the currently executing thread's UTCB, and then exits the system-call handler to return to user mode. It is necessary because the C++ routine that implements the IPC slowpath transfers even hardware-register-mapped virtual registers to UTCB locations. The switch frame established by the IPC fastpath ensures that this data is moved into the appropriate general-purpose registers when the recipient of an IPC-slowpath message is next dispatched.

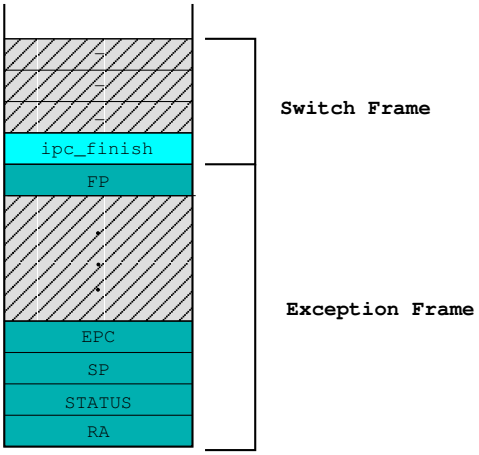

Figure 4.2: Exception and switch frames on the MIPS64 IPC-fastpath kernel stack.

When the IPC operation is a closed-wait non-call invocation, the IPC fastpath does not reference any data contained in the `from-thread`'s kernel stack. This is because the `from-thread` does not participate in the context switch performed by the IPC fastpath.

Before concluding this section, we note that a more optimised MIPS64 exception-frame layout would merge the exception and switch frames set up on the system-call invoker's kernel stack into only two (rather than three) cache lines. This could be achieved by ensuring that the fifth register saved on the exception frame occupies the same cache line used by the switch frame.

UTCB memory

The last category of cache lines touched by the IPC fastpath are those required to transfer memory-mapped virtual registers between two threads' UTCBs. Hence these cache lines are required only when more than nine virtual registers are involved in the message passing. The number of cache lines touched is proportional to the number of memory-mapped virtual registers transferred. Transferring all fifty-five such registers consumes fourteen cache lines in each the source and destination UTCBs.

Summary

When memory-mapped virtual registers are absent from the message transfer, every data cache line touched by the IPC fastpath corresponds to TCB memory, with `K_STACK_BOTTOM` being the sole exception. The fastpath references data in both the source and destination TCBs involved in the send phase. When a closed-wait, non-call invocation is handled, the fastpath also needs to reference the `from-thread`'s TCB. Data obtained from these two or three TCBs fall into two categories — those corresponding to TCB fields and those corresponding to kernel stack data. We summarise this information in Table 4.3 below:

Current Thread		`to-thread`		`from-thread`	
TCB Fields	Kernel Stack	TCB Fields	Kernel Stack	TCB Fields	Kernel Stack
2	3	2	2	2	0

Table 4.3: Number of cache lines the MIPS64 IPC fastpath references from thread-control-block memory.

Hence an open-wait or call IPC invocation that executes on the IPC fastpath uses $1 + (2 + 3) + (2 + 2)$ or 10 cache lines in the MIPS R4700's data cache. A round-trip call IPC between two threads will require 11 data cache lines because a switch frame is established on both threads' kernel stacks. It is important, however, to also consider which cache sets these cache lines can fall upon.

Cache colouring

TCBs in Pistachio/MIPS64 are 4096-bytes in size and always allocated aligned on this size. Hence each TCB field can only fall on one of two possible cache sets in the MIPS R4700's two-way associative 16KB-sized data cache. Because kernel stacks are contained in TCB structures, the same statement holds for cache lines touched by the IPC-fastpath kernel stack. Figure 4.3 below depicts precisely which cache sets are hit by the IPC fastpath when memory-mapped virtual registers are not involved in the message transfer.

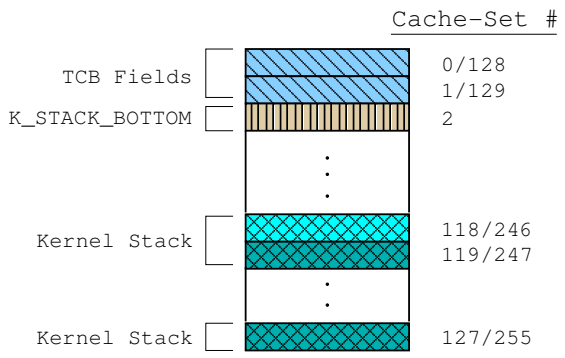

Figure 4.3: Data-cache colouring of the MIPS64 IPC fastpath.

4.2.3 TLB footprint

The IPC fastpath in L4Ka::Pistachio requires one virtual mapping for each TCB referenced. Two virtual mappings are always required — one for the currently executing thread and one for the send-phase destination (`to-thread`). A third is required for `from-thread` when the invocation is a closed-wait, non-call IPC.

4.3 Design of the ThreadID Table

A physically-addressed L4 kernel, by definition, does not pollute the system's TLB by requiring a virtual mapping per TCB referenced on the IPC fastpath. This benefit is automatic, and in particular, does not depend on the precise implementation of the threadID table. On the other hand, the threadID-table design does affect the severity of the (potential) performance penalties a physically-addressed kernel suffers in increased memory-cache usage and increased cost of translating thread numbers to TCB addresses.

In the following, we seek to investigate various design choices that may minimise any overhead inherent in performing threadID-table lookups. We do so because the performance-critical IPC fastpath requires at least one consultation of the threadID table when locating the TCB of the send-phase recipient (`to-thread`). A closed-wait, non-call IPC requires a second consultation to find the TCB corresponding to `from-thread`. The IPC fastpath never requires more than two threadID-table lookups (the TCB of the L4 IPC system-call invoker is located using the kernel-stack-pointer exploit described in Section 3.7.3).

Much of what is presented here applies equally well to all architectures. Nevertheless, some tricks are specific to the MIPS64 architecture or are dependent upon implementational features of the MIPS64 Pistachio kernel.

4.3.1 Address format

In its simplest form, each valid threadID-table entry would store the physical address of a TCB structure in 4 or 8 bytes depending on whether a 32-bit or 64-bit architecture is at hand. As shown in Section 3.6.2, however, a MIPS R4700 processor can always store physical addresses in 4 bytes despite executing on a 64-bit platform. No performance penalty is attached because the 4-byte truncated physical address naturally sign extends to the appropriate 64-bit virtual address in the `CKSEG0` segment of the R4700's address space (see Figure 4.1).

What is more interesting is the possibility of storing physical TCB addresses in a 2-byte format in an architecture-independent manner. To see how this can be achieved, note that:

1. Pistachio allocates all TCB structures from a contiguous, page-aligned block of physical memory (the kernel memory pool).

2. Pistachio always allocates TCB structures aligned on their size.

3. It is not unreasonable to expect the size of the kernel memory pool to be limited by $2^{16} \times$ `tcb_size` bytes in size where `tcb_size` represents the size of a TCB structure in Pistachio and is 4096 bytes for most platforms including the MIPS64.

Hence we can logically partition the kernel memory pool into `tcb_size` chunks and enumerate these chunks sequentially, starting at zero. The number assigned to each chunk can then be converted to and from that chunk's physical address using arithmetic operations not dissimilar to those performed by current Pistachio kernels when indexing the virtual TCB array.

Storing addresses in this 2-byte format clearly reduces the threadID table's memory usage. There is an additional benefit though in doubling the likelihood of two table entries falling on the same cache line. The degree to which this reduces the threadID table's net data-cache usage depends on the distribution of thread

numbers used to index the table. The chief drawback is of course the additional instructions required to translate threadID-table entries into usable TCB addresses. Furthermore, if the kernel memory pool cannot be statically allocated in memory, an additional cache line is required to obtain its base address. In Chapter 5, we ensure to evaluate the trade-offs involved here.

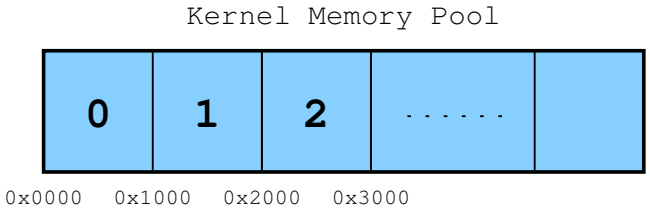

Figure 4.4: Partitioning the kernel memory pool into `tcb_size` chunks, numbered consecutively from zero.

4.3.2 Data structures

The threadID table assists a physically-addressed kernel in mapping a given thread number to the physical address of the TCB corresponding to that thread number. Hence its implementation is an instance of the classical dictionary interface, the key space being the range of valid thread numbers the kernel supports.

There are many data structures that implement the dictionary interface and it is not our intent to provide a thorough treatment of them all. We merely note that key-space size and key sparsity are two of the more important factors to consider when selecting an appropriate data structure. To this end, the L4 API restricts the thread-number space to 2^{18} or 2^{32} in size for 32-bit and 64-bit architectures respectively. Furthermore, the kernel has no control over the sparsity of active thread numbers as user-level servers have complete discretion over their assignment (with the exception of a few thread numbers reserved for interrupt and kernel threads). With these key-space properties in mind, commonly-used data structures for operating-system page tables become natural candidates for the threadID table.

There is one caveat that specifically applies to threadID tables, however. In Section 3.7.1 we described that initialising the threadID table to map all thread numbers to a zero-filled dummy TCB was necessary for cheaply validating thread IDs. Hence we ensure to assess the feasibility of this requirement for each data structure proposed.

Arrays

The threadID-table lookup is the source of the performance penalty incurred by a physically-addressed kernel on the IPC fastpath. A simple linear array indexed by thread number is therefore an attractive option for the implementation of our table. It requires a minimal number of instructions to index and just the one data cache line to translate thread numbers to TCB addresses. Furthermore, initialising an array to map all thread numbers to the dummy TCB is a trivial exercise. Memory requirements must be considered however — after all, the threadID table must reside entirely in physical memory if TCB addressing is to be made completely physical.

Pistachio typically provides somewhere between 2^{16} and 2^{22} thread numbers to user-level servers, the precise amount depending on the architecture at hand. A threadID array storing 4-byte TCB addresses consumes 256KB of memory in supporting 2^{16} thread numbers. Although this amount is not unreasonable on say a high-end server with an abundance of physical memory, it is likely to be intolerable on small or even medium-sized embedded systems where memory is particularly precious. It can be argued that on the smallest systems, a thread-number space much smaller than 2^{16} in size would suffice. Nevertheless, we shall present more resource-

friendly data structures for the benefit of systems where concerns for the memory usage of a threadID array cannot be overcome by simply reducing the amount of thread numbers made available by the kernel.

Hash tables

A hash function can be applied to thread numbers to index a hash table maintaining TCB addresses. Such a data structure reduces the memory usage of the threadID table at the expense of having to deal with collisions. Naturally, trade-offs exist between memory usage and collision rates.

Collision chains can be implemented by requiring each TCB maintain a `hash_next` field that points to the next TCB in its chain (if any). Each hash-table bucket would then simply store the physical address of the TCB at the head of the collision chain corresponding to that bucket. Locating a TCB then entails using the `hash_next` fields to traverse a collision chain, comparing the global thread identifier stored in each TCB encountered with that used to perform the threadID-table lookup. By placing `hash_next` on the same cache line as the global thread-identifier field in every TCB, the cache footprint required to traverse a collision chain can be minimised.

We do not need to consider initialising the hash table to map thread numbers to the dummy TCB (in fact this is not even possible). This is because the task of validating thread numbers is intertwined into the process of traversing collision chains. If the thread identifier used to perform a hash-table lookup matches against a TCB in the appropriate collision chain, then that thread identifier is automatically validated when a match is made. If the collision chain terminates before a valid match is found, then the thread identifier is known to be invalid.

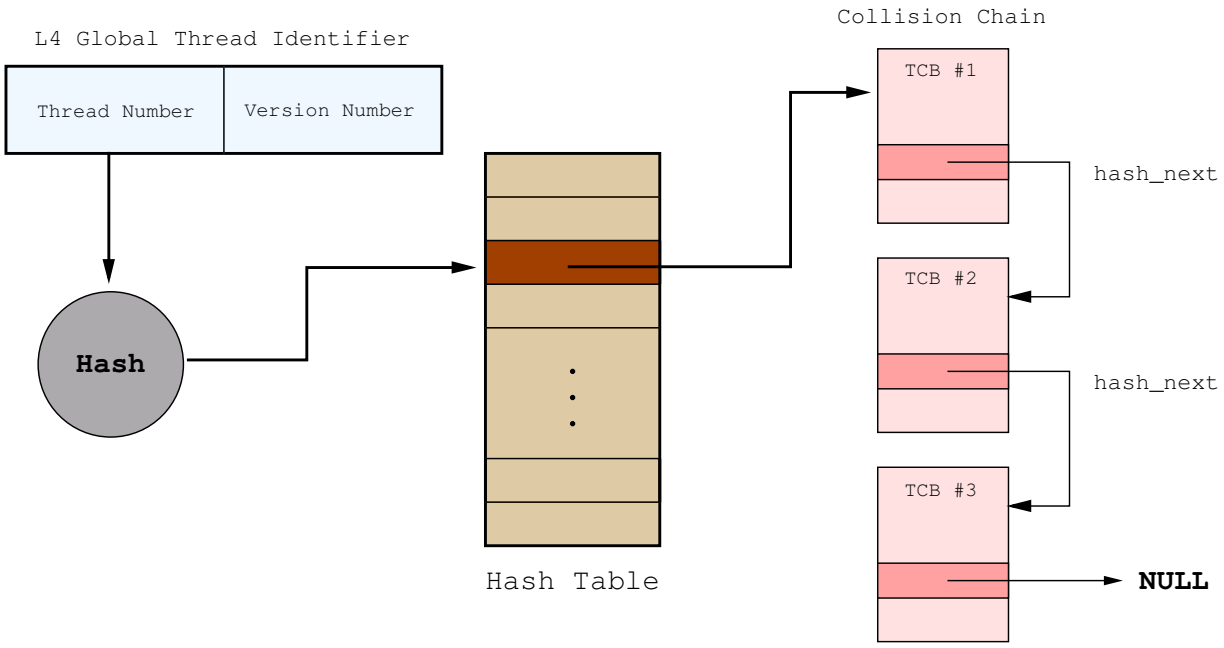

Figure 4.5: A hash-table data structure for the threadID table.

There are trade-offs inherent in selecting a suitable hash function. Indexing the hash table with the lower-order bits of a thread number is cheap, but more likely to result in a high collision rate than a hash function that performs, say, extensive bit-shifting and exclusive-or operations. Any overhead required to perform computationally-intensive hashing is likely to have a visible effect on the IPC fastpath which on most architectures has a best-case cycle count between 80 and 200. For the MIPS64 architecture where IPC best-case performance is in the order of 100 cycles, even a single multiplication instruction adds a minimum of 11%

overhead to the fastpath. Instruction-cache footprint must also be considered for hash functions requiring many arithmetic operations.

Hierarchial tables

The third class of data structures we consider for the threadID table are hierarchial tables. This implementation breaks the thread number used to perform a table lookup into a number of fragments which are used to index a hierarchial series of tables in turn. The number of fragments hence determines the depth of the traversal. Greater depth implies more expensive table lookups, but consumes less memory when user-level servers assign thread numbers sparsely.

The most naive method of initialising a hierarchial table to map all valid thread numbers to the dummy TCB would consume more memory than a linear-array implementation. A more responsible approach for, say, a three-level hierarchial table, would be to create a single dummy second-level node whose entries all point to a single dummy third-level node, whose entries in turn all point to the dummy TCB. The root-level table can then simply be initialised with pointers to the dummy second-level node. Whenever a new second-level node is allocated for the hierarchial table, its entries must all be initialised to point to the dummy third-level node.

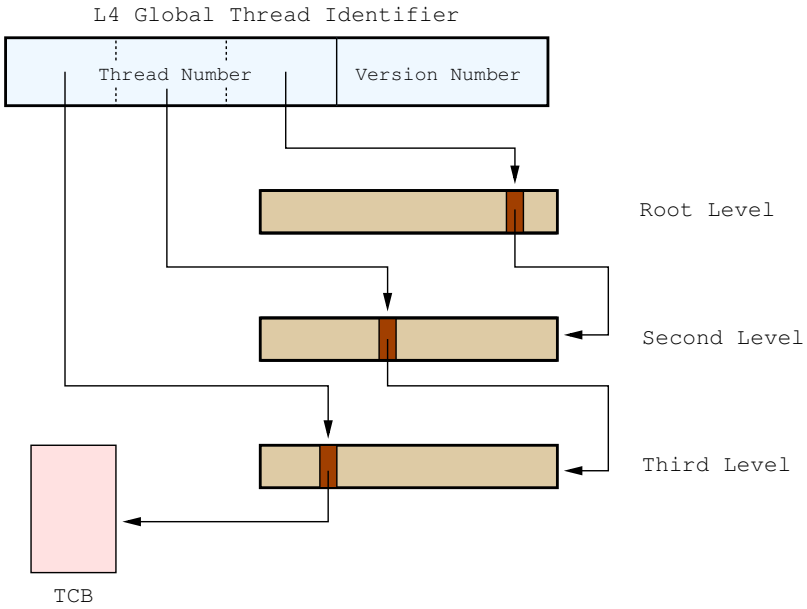

Figure 4.6: A hierarchial-table data structure for the threadID table.

4.3.3 Maximising cache-line value

Spatial locality

A physically-addressed kernel requires a potentially larger data-cache working set because of the need to consult an auxiliary table when translating thread numbers to TCB addresses. The severity of this increase will naturally depend upon the degree of spatial locality exhibited by threadID-table memory accesses. When the most frequently executed microkernel activity is IPC, the strength of this spatial locality is effectively determined by the distribution of thread numbers attached to threads participating in IPC operations.

Unfortunately, the distribution of thread numbers used to index the threadID table is largely determined by the operating-system personality running on top of L4 rather than by the microkernel itself. For one, thread

numbers are assigned by user-level servers and so the microkernel cannot anticipate the sparsity of thread numbers involved in IPC activity. Furthermore, the architecture of the user-level operating system also strongly influences the pattern of threadID-table lookups. For example, a multi-server environment where all critical user-level servers are assigned thread numbers sparsely will result in a much weaker locality of access to a hierarchial threadID-table than if the operating-system personality featured only a single system-call server whose thread number was numerically similar to all its worker threads.

Merged TCB fields

When spatial locality of threadID-table accesses is poor, a physically-addressed kernel will incur a higher data-cache miss rate. In the worst case, every cache line read from the threadID table will result in a cache miss and will only be useful in locating a single TCB. In hope of minimising any potential performance penalty here, we investigate the usefulness of increasing the amount of immediately-useful data contained in each cache line read from the threadID table. This can be achieved by copying fields from a thread's TCB into the threadID-table entry corresponding to that thread. Moving rather than copying data from TCBs into the threadID table poses problems because not every TCB is located using the threadID table (recall the kernel-stack-pointer trick in Section 3.7.3). We call TCB fields copied into the threadID table, *merged TCB fields*.

The rationale behind this design choice is that a cache line obtained from the threadID table might contain enough useful TCB data to eliminate the need to fetch one other cache line from a TCB structure. In short, two cache-line fetches may be merged into a single fetch.

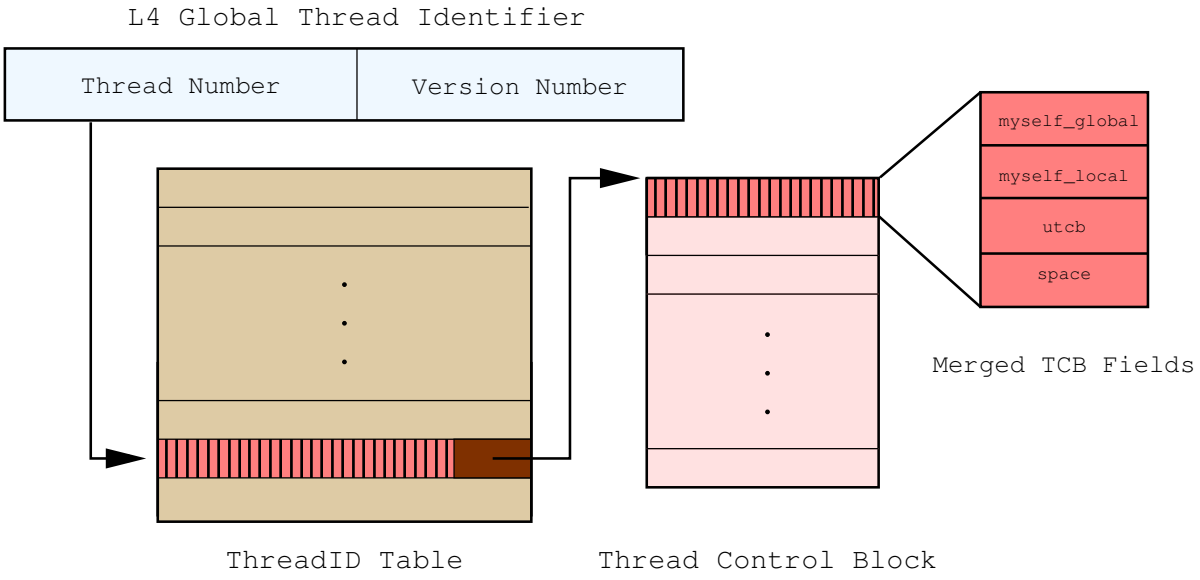

Figure 4.7: A threadID table where each entry duplicates a subset of TCB fields.

Drawbacks

Naturally, this design also has its drawbacks. Copying TCB fields into the threadID table increases the size of each threadID-table entry. This not only increases the memory requirements of the table, but also decreases the probability of two threadID-table entries falling on the same cache line. The latter drawback only reiterates that duplicating TCB fields in the threadID table is most suitable when threadID-table accesses result in a high data-cache miss rate. Furthermore, any TCB fields duplicated in the threadID table will need to be kept consistent

with the corresponding TCB-maintained values. Hence in choosing TCB fields to merge into the threadID table, those with relatively static values are more attractive candidates.

The IPC fastpath revisited

The TCB fields referenced by the IPC fastpath are the most appropriate candidates for merging into the threadID table, because these are the most likely fields to be referenced by the kernel soon after performing a threadID-table lookup.

In Section 4.2.2 we noted that the MIPS64 IPC fastpath only references the very first two cache lines of data from each of two or three TCBs accessed whilst servicing the system call. The first of these cache lines contains the TCB fields `myself_global`, `myself_local`, `utcb` and `space`. A detailed description of these fields has already been provided in Table 4.2. Here we simply note that all these fields hold highly-static values that are unlikely to frequently change (if at all) over the course of a thread's lifetime. In contrast, the second cache line of TCB fields referenced by the IPC fastpath contains an assortment of highly-volatile values including a thread's execution state and IPC partner (see Table 4.2). Hence on the MIPS64, the first cache line in each TCB structure is the most appropriate candidate for merging into threadID-table entries.

Compressing TCB fields

Performing the merge must be done without increasing the size of the resulting threadID-table entry beyond that of a cache-line size. Otherwise the threadID table will fail to merge two distinct cache-line fetches into a single cache-line fetch. For the MIPS64, this requirement is not overly burdening. Pistachio on the MIPS64 currently stores the `utcb` and `space` fields in each TCB in 8 bytes. But these fields are just physical pointers to UTCB and page-table structures in `CKSEG0`. As first described in Section 3.6.2, these fields can thus be truncated to 4 bytes without any performance penalty. Hence a MIPS64 threadID table can duplicate all the data contained in the first cache line of each TCB using only 24 bytes. On the MIPS R4700 where cache lines are 32-bytes wide, the remaining 8 bytes in each threadID-table entry can be used to maintain a pointer to the appropriate TCB (even 4 bytes would suffice for this).

Other architectures may need to resort to address-compression techniques similar to those introduced in Section 4.3.1. These techniques can be used to reduce the space requirements of the `utcb` and `space` fields, but imply overhead in address decompression. The alternative is to only merge a subset of the fields contained in each TCB's first cache line. For example, the `utcb` field could be neglected as it is only referenced by the IPC fastpath when transferring memory-mapped virtual registers. This is more suitable for RISC architectures where hardware-register-mapped virtual registers are generally more abundant than on their CISC counterparts.

A Caveat: The currently executing thread

Consider a threadID table on the MIPS64 where each threadID-table entry duplicates the `myself_global`, `myself_local`, `utcb` and `space` TCB fields. It may be hoped that such a design might completely eliminate the kernel's need to fetch the first cache line in any TCB structure, except for the purpose of updating threadID-table entries. Certainly the kernel does not need to perform this fetch for TCBs located via the threadID table. However, a dilemma is posed by the currently executing thread whose TCB is located via the kernel stack pointer and not the threadID table.

If the currently executing thread is to obtain its `utcb` field (say) without accessing the first cache line in its TCB, the kernel must, on each thread switch, store a pointer to the newly-dispatched thread's threadID-table entry to some pre-defined static memory location (or a kernel-protected register). Otherwise the only means by which a thread may locate its own threadID-table entry is to read its thread number from its own TCB and use that to index the threadID table. But this clearly requires the first cache line of the current thread's TCB to be fetched from memory and hence defeats the purpose of obtaining TCB fields from the threadID table.

There are thus two competing designs for merging TCB fields into the threadID table. In one design the kernel stores a pointer to each thread's threadID-table entry to a pre-defined location in memory, and the currently executing thread uses that pointer to fetch its own merged TCB fields from the threadID table. The advantage here is that fetching the first cache line in any TCB structure is never required, except for updating threadID-table entries (in fact with this design, merged TCB fields could be moved rather than copied into the threadID table). The drawback is overhead involved in storing and possibly retrieving a pointer to the current thread's threadID-table entry on every thread switch. For a MIPS64 kernel, this does not increase the kernel's data-cache footprint because the threadID-table entry pointer can be stored to the same cache line as the K_STACK_BOTTOM variable that is already updated on each thread switch with a thread's kernel stack base.

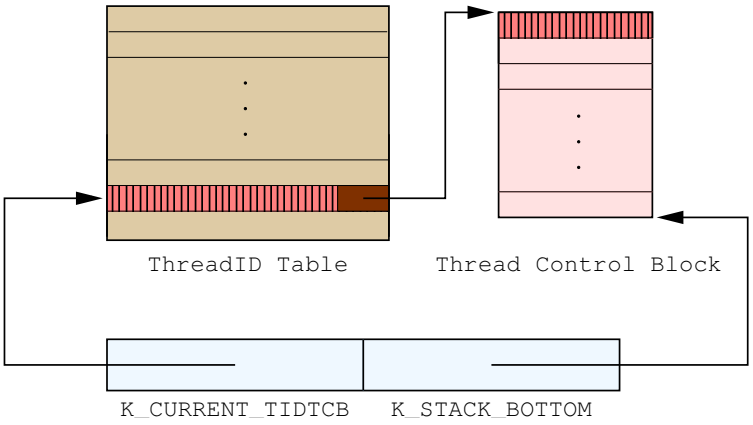

Figure 4.8: Locating the current thread's threadID-table entry on the MIPS64.

The alternative is for the current thread to simply obtain its own merged TCB fields from its TCB structure and not the threadID table. This design is unlikely to reduce the kernel's cache footprint, as even though sending an IPC from thread A to thread B does not itself require fetching the first cache line of thread B's TCB, that cache line must still be fetched should thread B send an IPC reply to thread A. Nevertheless, this design still reduces the number of cache lines touched by the IPC fastpath by one, and hence may lead to a smaller number of data-cache misses there. For example, if thread A sends thread B an IPC on a clean cache, one cache miss will be saved by not accessing the first cache line in thread B's TCB. On the other hand, if thread B immediately sends thread A an IPC reply, the cache miss is not actually saved but merely delayed.

The trade-offs involved here are highly non-trivial. A chief objective of the following chapter will be to not only evaluate the usefulness of merging TCB fields into the threadID table, but to also compare the two competing designs for handling how the current thread references its own merged TCB fields.

4.3.4 A threadID-table cache

There is a trade-off between the complexity of threadID-table lookups and the memory usage of the threadID table. An array is the cheapest data structure to index, but consumes a large, fixed amount of memory. A hierarchial table on the other hand may be more resource friendly, but indexing it with a thread number touches a number of data cache lines equal to its depth. For this reason, it becomes appropriate to consider a software-managed cache of threadID-table entries backed by a more complex but space-efficient data structure like a hash or hierarchial table. When threadID-table cache hit rates are sufficiently high, the cost of traversing the more complex data structure should be concealed by the cost of querying the cache.

Cache design

Indexing a cache requires consideration of the same trade-offs involved in selecting a hash function for a hash-table data structure. We only consider bit selection for indexing a cache because anything more complicated is likely to have a visible effect on the performance of the IPC fastpath.

Suppose for now that the threadID table does not duplicate TCB fields and that the threadID-table cache is direct mapped. In general, cache entries need to be tagged to distinguish between items mapped to the same cache set. For the threadID-table cache, thread identifiers or even thread numbers suffice as tags. However, the threadID-table cache does not require explicit tagging of its entries. Instead, the global thread-identifier field contained in the TCB mapped by a cache entry can be used as a tag. This of course requires invalid cache entries map to the dummy TCB.

An untagged threadID-table cache reduces memory usage for a fixed number of cache entries, but requires an extra line of data to be read when querying the cache. On a threadID-table cache hit, however, this extra line of data is no other than the first cache line in the newly-located TCB and is therefore very likely to be referenced anyway. In particular, it is always referenced by the IPC fastpath and so we do not consider it real overhead, unless a threadID-table cache miss occurs. Of course, querying a non-direct-mapped untagged cache is likely to reference the `myself_global` field in TCBs other than the one being located, even when a threadID-table cache hit occurs. Hence we consider this design choice most suitable for direct-mapped caches.

A further advantage of untagged threadID-table cache entries is that the cache-fetch process effectively incorporates the thread-identifier validation process. Hence on a threadID-table cache hit, the thread identifier used to index the cache is automatically validated. This can also be achieved by explicitly tagging cache entries with complete thread identifiers rather than just thread numbers, but one must be careful not to insert a mapping to the dummy TCB into the cache, unless it is tagged with `NilThread` (zero).

Increasing the associativity of a software-managed cache may reduce conflict misses, but increases the cost of querying the cache. This is because not only must each entry in a cache bucket be searched through sequentially, but on a data-cache miss the processor must wait for a larger portion of the missing cache line to be refilled from memory before it can resume execution. The latter is only hurtful on architectures that permit program execution to occur in parallel with data-cache refills. Furthermore, a replacement policy must be implemented for non-direct-mapped caches, which adds non-negligible overhead to the refill process, and possibly even to the lookup process.

Finally, we note that a threadID table implemented as a hash table features an implicit cache if whenever a TCB is located, it is moved to the head of its collision chain. A separate, external cache may still be desirable, however, if a cache-indexing function other than the hash table's (potentially expensive) hash function is desired.

Caches and merged TCB fields

Consider a threadID table where each entry is 32 bytes in size and not only maintains a TCB address, but also maintains duplicate copies of the `myself_global`, `myself_local`, `utcb` and `space` fields from that TCB. Using a linear array to store 2^{16} such entries would consume 2MB of memory which is certainly not reasonable for all systems. Hierarchial and hash tables are more suitable data structures when threadID-table entries are this large, but unfortunately, maintaining a cache of threadID-table entries for these more complicated data structures enjoys some non-trivial intricacies when the threadID table duplicates TCB fields.

It is clear that threadID-table cache entries need to maintain copies of merged TCB fields themselves. Otherwise, in the desirable case where threadID-table cache hit rates are high, performing a TCB lookup will not eliminate the need for a second data-cache-line fetch when reading that TCB's merged fields. On the other hand, it is not necessary for the threadID table that backs the cache to also duplicate these TCB fields. This is because the cache-refill process can simply obtain the merged TCB fields from the actual TCB structure found by indexing the backing store. The trade-off to consider here is memory usage versus an extra data cache line

to read on a threadID-table cache miss. When threadID-table cache hit rates are sufficiently high, the reduced memory usage gained by not copying TCB fields into the backing threadID table comes essentially for free.

More subtle intricacy exist. Reading merged TCB fields from the threadID-table cache is dangerous because threadID-table cache entries are volatile. It is possible that before the kernel has finished using the merged TCB fields in a certain threadID-table cache entry, that cache entry is replaced by some other thread's data. We call the occurrence of this anomaly a *premature merged-cache conflict*. Premature merged-cache conflicts can be minimised by increasing cache associativity. Doing so, however, is likely to increase the size of each threadID-table cache entry beyond that of a data cache-line size. In particular this is true for the MIPS R4700 where the primary data cache features 32-byte line sizes. Hence we shall only consider direct-mapped threadID-table caches in the following.

Premature merged-cache conflicts and the MIPS64 IPC fastpath

We propose that merged TCB fields should only be referenced from a threadID-table cache on the IPC fastpath. This is because there are few opportunities for a premature merged-cache conflict to occur on the fastpath, and we shall shortly show how these can be efficiently handled. Everywhere else, the kernel should read merged TCB fields from the actual TCB structures. When fastpath IPCs are the most frequently executed kernel primitive, this design choice should still see the benefits (if any) of merging TCB fields into the threadID-table structures.

On the IPC fastpath, a premature merged-cache conflict occurs when a closed-wait, non-call IPC is at hand and the `to-thread` and `from-thread` IPC arguments index the same threadID-table cache entry. For the MIPS64 IPC fastpath, the TCB corresponding to `to-thread` is always located before the TCB corresponding to `from-thread`. Hence one solution would be to simply branch to the IPC slowpath when a threadID-table cache miss occurs when locating `from-thread`'s TCB. A better solution would be for the threadID-table cache-refill routine to explicitly check if `from-thread`'s cache entry conflicts with `to-thread`'s cache entry, and then only branch to the IPC slowpath. Performing this check is not expensive when the cache-indexing function is cheap, and furthermore, it does not affect the performance of threadID-table cache hits. A third optimisation is possible by simply skipping the threadID-table cache-refill process when the cache-miss handler detects that `from-thread`'s threadID-table cache entry conflicts with `to-thread`'s entry. In this case, the kernel must reference `from-thread`'s merged TCB fields from either its actual TCB structure or the threadID-table backing store. But this avoids a branch to the slowpath which would not read the merged TCB fields from the threadID-table cache anyway. We cease to consider further tricks because closed-wait, non-call IPC invocations are relatively rare, and because we anticipate that threadID-table cache hit rates will be very high. The following chapter shall justify these claims.

Premature merged-cache conflicts and the currently executing thread

In Section 4.3.3 we noted that if the currently executing thread is to read its own merged TCB fields from the threadID table rather than its TCB structure, then the kernel must on each thread switch, store a pointer to the newly-dispatched thread's threadID-table entry to a pre-defined static memory location. We shall call this memory location `K_CURRENT_TIDTCB` (see Figure 4.8).

Whenever the kernel dispatches a new thread, that thread's TCB must first be located so that its execution context can be restored. Hence on each thread switch, it is not unreasonable to expect that the newly-dispatched thread's threadID-table entry will already be present in the threadID-table cache. It then becomes natural for the `K_CURRENT_TIDTCB` variable to point to the currently executing thread's threadID-table cache entry rather than to an entry in the threadID-table backing store. The alternative requires the IPC fastpath to always locate the `to-thread`'s entry in the more complicated backing-store data structure, and this of course defeats the purpose of maintaining a threadID-table cache in the first place.

The caveat is that, upon re-entering kernel mode, a thread cannot automatically assume that the cache entry pointed to by K_CURRENT_TIDTCB still caches that thread's threadID-table entry. This can only cause a problem for the IPC fastpath because everywhere else, the kernel reads merged TCB fields from TCB structures and hence does not need to use K_CURRENT_TIDTCB.

As an illustration of this problem, suppose thread A invokes the L4 THREADCONTROL system call to modify thread B's scheduler. In handling this (non-blocking) system call, the kernel will locate thread B's TCB, manipulate it, and then resume executing thread A. If thread B's thread number maps to the same threadID-table cache entry as thread A's thread number, then should thread A later invoke a (fast) IPC system call before the next timer interrupt, the cache bucket pointed to by K_CURRENT_TIDTCB will no longer contain thread A's merged TCB fields.

Hence on entering the IPC fastpath, the kernel needs to determine if the current thread's threadID-table entry is still cached. There is overhead involved in this, but the severity can be minimised. The idea is to simply check if the kernel stack pointer points somewhere inside the TCB whose physical address is stored in the cache bucket identified by K_CURRENT_TIDTCB. For a MIPS64 kernel this derives to the task of checking if K_STACK_BOTTOM points to the last byte of the TCB pointed to by the current thread's cache bucket.

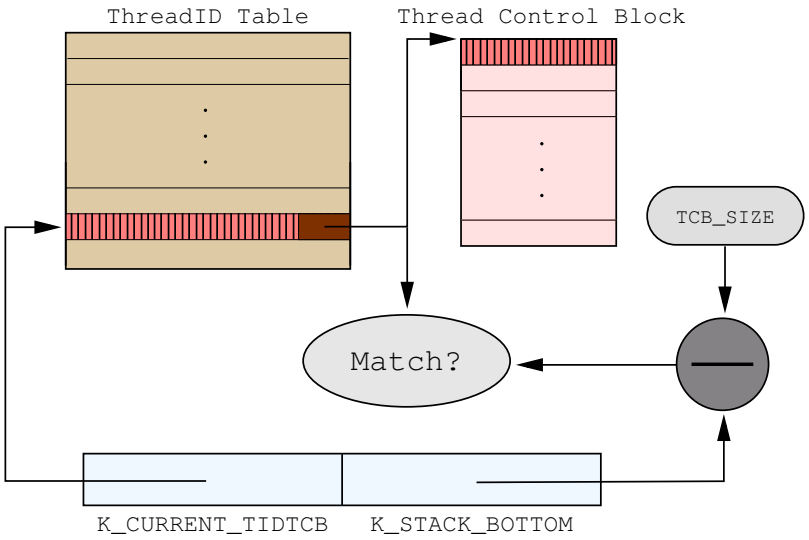

Figure 4.9: Determining if the current thread's threadID-table entry is still present in the threadID-table cache.

When to perform this check is not obvious. The best solution is to ensure the IPC fastpath locates the TCBs corresponding to `to-thread` and (when appropriate) `from-thread` before it references any of the currently executing thread's merged TCB fields. Then by performing the check immediately after the `to-thread` and `from-thread` TCBs are located but before any of the current thread's merged TCB fields are required, the kernel not only determines if the current thread's cache bucket has been mapped out of the threadID-table cache prior to the IPC system-call invocation, but also if it has been mapped out by `to-thread` or `from-thread`.

It remains to address how the kernel should respond after it determines the cache bucket pointed to by K_CURRENT_TIDTCB cannot be used to read the current thread's merged TCB fields from the threadID-table cache. In this situation we propose that the kernel simply reference the current thread's merged TCB fields from the current thread's actual TCB structure (located via the kernel stack pointer and not the threadID table). This is not a burden because the kernel already requires the current thread's TCB for reading the `send_head`, `partner` and `thread_state` fields (see Table 4.2).

We conclude this section by noting that although a software-managed cache of threadID-table entries dupli-

cating TCB fields enjoys some non-trivial intricacies, almost all of these can be addressed without incurring any performance penalty when threadID-table cache hit rates are very high. The only exception is that a kernel that maintains a pointer to the currently executing thread's cache entry in some pre-defined location must expend a few cycles on the IPC fastpath ensuring that the current thread's threadID-table entry is still present in the cache.

4.4 Implementation & Analysis

We now turn to presenting the various implementations of a physically-addressed L4 kernel that are subjected to evaluation via benchmarking in the following chapter. Each of the concrete implementations presented here is for the MIPS64 architecture and differs from the others only in the design of its threadID table.

As Section 4.3 clearly illustrated, there are many design choices to consider when implementing a threadID table. It is beyond the scope of this document to evaluate every single alternative. Instead we have selected just a few key designs that we conjecture will best illustrate the performance impact of physically addressing TCBs. To this end we begin by describing the rationale behind our selection of threadID-table designs for implementation.

4.4.1 Selecting threadID-table designs

Implementing a completely physically-addressed L4 kernel (neglecting long IPC) is not itself very difficult. At a minimum, all that is required is maintaining an array of physical TCB addresses that can be indexed by thread number. Such an implementation will not have a significant effect on the best-case performance of the IPC fastpath because indexing a table to locate a TCB involves the same level of complexity as indexing a virtual array of TCBs. However this implementation does, of course, increase the IPC fastpath's data-cache footprint.

A chief objective of our work is to evaluate whether it is worthwhile expending a few extra cycles on the IPC fastpath in hope of reducing its data-cache footprint. Two design choices introduced in Section 4.3 are suitable for such an investigation. The first involves compressing threadID-table entries (see Section 4.3.1) and the second involves copying fields from TCB structures into the threadID table (see Section 4.3.3). For the latter, we also seek to determine whether it is beneficial for the kernel to store the currently executing thread's threadID-table entry to a pre-defined location in memory, so that each thread can reference its own merged TCB fields without fetching the first cache line of its TCB.

Our second and final (implementation-oriented) chief objective is to evaluate the usefulness of a software-managed cache for concealing the overhead of a more sophisticated but resource-friendly data structure. We do so for structures that maintain only TCB addresses and also for structures that additionally maintain merged TCB fields. The latter is so that we can evaluate the performance impact (if any) of handling the premature merged-cache conflicts described in Section 4.3.4.

We anticipate that a software-managed cache of threadID-table entries will perform sufficiently well that the precise data structure used as a backing store becomes immaterial. Given this, it is not our objective to evaluate the performance of every data structure suitable for backing a cache. Hence all our threadID-table cache implementations are backed by the same hierarchial table (the decision to use hierarchial tables over hash tables was made arbitrarily).

4.4.2 Nine concrete implementations

There are nine implementations we subject to benchmarking in the following chapter. We have elected not to provide here the actual source code used for these implementations because no threadID-table data structure used is complicated enough that its implementation becomes non-trivial. Instead we merely describe each design in sufficient detail so that its precise implementation is made unambiguous. The curious reader can nevertheless find the source code in Appendix B.

We also provide the best-case IPC-fastpath cycle counts (on the MIPS R4700) for each of the nine implementations. In measuring these cycle counts, we created a ping-pong environment where two threads in different address spaces bounced a null message back and forth between each other in a loop. The IPC system call used was an open-wait invocation and only one hardware-register-mapped, untyped virtual register was involved in the transfer. No TLB or system memory-cache misses occurred (hence best-case performance). The best-case cycle count was computed by dividing the time taken to complete the entire loop by the number of loop iterations and then subtracting the loop overhead. The loop was repeated a sufficient amount of times to negate the effect of any timing costs. Hence the cycle counts presented include the cost of entering and leaving kernel mode.

Further more, we ensure to highlight how the particular design choices involved in each implementation are reflected in the cycle counts provided. This entails comparing the various implementations and accounting for discrepancies in their best-case performances. We believe that presenting the best-case performance of each implementation and accounting for each such measured cycle is more revealing than simply pasting code snippets.

`kern_virt`

The L4 kernel we christen `kern_virt` is not actually one of our nine physically-addressed kernel implementations but a kernel that addresses TCBs virtually. It is identical in every manner to each of the following physically-addressed kernels except for the manner in which it addresses TCBs. `kern_virt` is effectively the Version 0.4 release of L4Ka::Pistachio on the MIPS64, with the exception of its TCB layout (displayed in Appendix A) and TLB-refill routine differing. The former change was made to reduce the number of cache lines the IPC fastpath reads from TCB structures from four to two (see Section 4.2.2). The latter change is irrelevant to the current topic at hand, but will be treated appropriately in the following chapter where we consider the impact a physically-addressed kernel has on TLB miss rates.

The best-case IPC performance of the `kern_virt` kernel was measured at 114 cycles, of which only 5 are used to index a virtual TCB array to locate the send-phase target's TCB.

`kern_phys_4b`

The `kern_phys_4b` kernel is our simplest implementation of a physically-addressed L4 kernel. It simply uses a statically-allocated array that can be indexed by thread number to obtain one of 2^{16} physical TCB addresses. Each address is stored in 4 bytes, but can be sign extended without penalty to a valid 64-bit `CKSEG0` address. Because the array's location in memory is pre-defined, a memory access is not required to load its base address. That is, the base address of the threadID-table array can be loaded as an immediate value.

The best-case IPC performance of the `kern_phys_4b` kernel was measured at 114 cycles, of which 5 are used to index the threadID-table array to locate the send-phase target's TCB.

`kern_phys_2b`

The `kern_phys_2b` kernel is identical to the `kern_phys_4b` kernel except that its threadID-table array stores physical TCB addresses in the compressed 2-byte format described in Section 4.3.1. For this implementation, we ensured that the kernel memory pool was statically allocated so that its base address could be loaded as an immediate value. Obtaining the base address of the kernel memory pool is critical in this design because it is required for expanding 2-byte compressed addresses.

The best-case IPC performance of the `kern_phys_2b` kernel was measured at 118 cycles, of which 5 are used to index the threadID-table array to obtain a 2-byte value that is decompressed into a valid 64-bit TCB address in a further 4 cycles.

kern_phys_merge

The kern_phys_merge kernel also uses an array for implementing its threadID table. Each entry in this array is 32 bytes in size and contains duplicates of all TCB fields that fall on the first cache line of a TCB structure. The precise merged TCB fields are myself_global, myself_local, utcb and space. In fact the format of the first 24 bytes of each entry in the threadID-table array is identical to the first 24 bytes of a TCB structure (see Appendix A). Of the remaining 8 bytes in each array entry, 4 are used to store the physical address of a TCB and the other 4 remain unused. This works because the last 8 bytes of the first cache line in a TCB structure are not used for any purpose.

The kern_phys_merge kernel does not maintain a pointer to the currently executing thread's threadID-table entry. Hence this kernel reads the current thread's merged TCB fields from that thread's actual TCB structure. It does, however, read the merged TCB fields of every thread other than the currently executing one from the threadID table, but even then only on the IPC fastpath.

The best-case IPC performance of the kern_phys_merge kernel was measured at 115 cycles, of which 5 are used to index the threadID-table array to locate the send-phase target's TCB. A further 1 cycle is used to setup reading the send-phase target's merged TCB fields from the threadID table rather than a TCB structure.

kern_phys_merge_store

The kern_phys_merge_store kernel is implemented identically to the kern_phys_merge kernel except that it maintains a pointer to the currently executing thread's threadID-table entry. It maintains this pointer in a pre-defined memory location called K_CURRENT_TIDTCB that resides on the same cache line as K_STACK_BOTTOM — the location where the kernel stores each thread's kernel stack base on thread switch.

The kern_phys_merge_store kernel thus always reads merged TCB fields from the threadID table when executing on the IPC fastpath. When not on the IPC fastpath, these fields are read from actual TCB structures. Hence this kernel's fastpath references exactly the same number of data cache lines as the kern_virt kernel.

The best-case IPC performance of the kern_phys_merge_store kernel was measured at 117 cycles or 2 cycles more expensive than the kern_phys_merge kernel. Of these 2 extra cycles, one is used for loading the address of the current thread's threadID-table entry from K_CURRENT_TIDTCB, and another is used for updating K_CURRENT_TIDTCB with a pointer to the send-phase target's threadID-table entry on thread switch.

kern_phys_cache_1way, kern_phys_cache_2way, kern_phys_cache_4way

These three physically-addressed kernels all feature a software-managed cache of threadID-table entries backed by a hierarchial table. They only differ in the level of associativity used by their caches. Neither the caches nor the hierarchial tables duplicate any TCB fields.

The hierarchial-table backing store features three levels of nodes with each node capable of holding 1024 4-byte physical addresses. These addresses point to next-level nodes or, in the case of third-level nodes, point to actual TCB structures. Hence this hierarchial table supports a thread-number space of 2^{30} in size and is indexed by partitioning the lower-order bits of a given thread number into three 10-bit segments. The root node of the hierarchial table is pre-allocated in memory at a fixed location so that its address can be obtained as an immediate value. Thus indexing this data structure to successfully locate a TCB touches three cache lines of data.

All three kern_phys_cache_*way kernels maintain a software-managed cache of the hierarchial table's entries. Each entry in the cache consists of a 4-byte physical TCB address and a 4-byte thread number that acts as a tag. The caches can hold at most 32 of these entries. The kern_phys_cache_1way kernel's threadID-table cache is direct mapped, where as the kern_phys_cache_2way and kern_phys_cache_4way kernels feature 2-way and 4-way associative caches respectively. Hence the kern_phys_cache_1way kernel features 32 cache sets and the kern_phys_cache_4way kernel features 8 cache sets. All caches are indexed using the lower-order bits of the thread number whose TCB is being located, and all caches are allocated at a pre-defined,

known location in memory so that their base address can be obtained without referencing memory. A threadID-table cache hit thus touches only one cache line of data for any of the three `kern_phys_cache_*way` kernel implementations.

For the two kernels featuring non-direct-mapped caches, the cache-refill process uses the `CP0_COUNT` co-processor register to implement a pseudo-random replacement algorithm. The `CP0_COUNT` register on co-processor unit zero contains a 32-bit value that is incremented by the hardware every two pipeline cycles. When a threadID-table cache miss occurs, the `kern_phys_cache_2way` and `kern_phys_cache_4way` kernels use the lower bits of the `CP0_COUNT` register to select an entry within a cache set for replacement.

The best-case IPC performance of the `kern_phys_cache_1way` kernel naturally occurs on a threadID-table cache hit and has been measured at 118 cycles. Of these, 6 are used to index the threadID-table cache and 3 are used to match the indexed cache entry's tag against the thread number whose TCB is being located. For the curious, the best-case IPC performance of this kernel on a threadID-table cache miss was measured at 142 cycles.

The `kern_phys_cache_2way` kernel enjoys a best-case IPC performance of 118 or 122 cycles depending on whether the threadID-table cache lookup resulted on a hit in the first or second entry of the cache set indexed. Its best-case threadID-table cache-miss performance was measured at 150 cycles, 8 cycles slower than the analogous measurement for the `kern_phys_cache_1way` kernel. Of these 8 extra cycles, 4 are because the `kern_phys_cache_2way` kernel must use the `CP0_COUNT` register to select a cache entry for replacement, and 4 are simply because two cache buckets rather than one must be checked before a cache miss is known to have occurred.

Lastly, the `kern_phys_cache_4way` features a best-case IPC performance of 118, 122, 125 or 128 cycles depending on how many threadID-table cache entries must be checked in a cache set before a hit occurs. The best-case threadID-table cache-miss performance of this kernel is 156 cycles.

`kern_phys_merge_cache`, `kern_phys_merge_store_cache`

These two kernels are similar to the `kern_phys_cache_1way` kernel in that they both feature a direct-mapped cache of threadID-table entries backed by a hierarchial table. However, the threadID-table structures in these kernels do maintain duplicates of the TCB fields falling on the first cache line of a TCB structure (that is, they merge the same TCB fields as the `kern_phys_merge` and `kern_phys_merge_store` kernels).

The hierarchial table used by the `kern_phys_merge_cache` and `kern_phys_merge_store_cache` is identical to that used by the `kern_phys_cache_1way` kernel, except that its third-level nodes consist of 128 32-byte entries that maintain duplicates of the `myself_global`, `myself_local`, `utcb` and `space` TCB fields in the first 24 bytes, and store a physical TCB address in 4 of the remaining 8 bytes. That is, the entries of the third-level nodes have precisely the same format as the 32-byte entries of the threadID-table arrays used by the `kern_phys_merge` and `kern_phys_merge_store` kernels.

The direct-mapped software cache of threadID-table entries used by the `kern_phys_merge_cache` and `kern_phys_merge_store_cache` kernels consists of 32 entries, each 32-bytes in size and a complete replication of the hierarchial-table entry being cached. The `myself_global` TCB field merged into each threadID-table cache entry acts as a cache tag.

The `kern_phys_merge_cache` and `kern_phys_merge_store_cache` kernels differ in the same way the `kern_phys_merge` and `kern_phys_merge_store` kernels differ. Hence the latter stores a pointer to the currently executing thread's threadID-table cache entry in the `K_CURRENT_TIDTCB` memory location so that even the current thread's merged TCB fields can be obtained without referencing the first cache line of a TCB structure.

Both kernels must handle premature merged-cache conflicts that arise when the TCBs corresponding to the `to-thread` and `from-thread` parameters of a closed-wait, non-call IPC invocation fall on the same threadID-table cache set. To this end, separate cache-refill routines are used for handling `to-thread` and

from-thread TCB-lookup threadID-table cache misses. The two refill handlers are identical except that the routine used for from-thread misses first checks if from-thread's threadID-table cache entry coincides with that of to-thread's. When a clash is detected, the from-thread's refill handler simply skips the cache-refill process and the kernel references from-thread's merged TCB fields from the hierarchial-table backing store rather than the threadID-table cache.

The kern_phys_merge_store_cache implementation must also ensure that the currently executing thread's hierarchial-table entry is still cached each time the kernel enters the IPC fastpath. Otherwise the pointer stored in K_CURRENT_TIDTCB cannot be used by the kernel to obtain the current thread's merged TCB fields from the threadID-table cache instead of its actual TCB structure. The kern_phys_merge_store_cache kernel performs this check using the trick described in Section 4.3.3. In short, the kernel determines if its stack pointer lies inside the TCB whose address is contained in the threadID-table cache bucket identified by K_CURRENT_TIDTCB. For a closed-wait, non-call IPC invocation, this check needs to be performed after the TCBs corresponding to both to-thread and from-thread have been located, but before any of the currently executing thread's merged TCB fields are referenced. Unfortunately our kernel locates from-thread's TCB after performing this check and so it is prone to malfunction should a closed-wait, non-call IPC be issued where from-thread's thread number differs from the IPC invoker's thread number by a multiple of 32 (the threadID-table cache size). This does not pose a problem for the evaluation conducted in the following chapter because our benchmarking environment performs only open-wait and call IPCs. We are also confident that the necessary change to close this loophole can be made without any degradation of the kern_phys_merge_store_cache kernel's performance.

The best-case IPC performance of the kern_phys_merge_cache kernel was measured at 119 cycles. The best-case threadID-table cache-miss performance was measured at 147 cycles. The analogous measurements for the kern_phys_merge__store_cache kernel are 123 and 151 cycles respectively, or 4 more cycles than the kern_phys_merge_cache-kernel counterparts. Of these extra 4 cycles, 1 cycle is used for reading the base address of the current thread's threadID-table cache bucket from K_CURRENT_TIDTCB, 2 cycles are for ensuring this cache bucket has not been invalidated by a premature merged-cache conflict, and the last cycle is for updating K_CURRENT_TIDTCB when to-thread is switched to.

4.4.3 Implementational drawbacks

In this section we highlight the shortcomings of the nine implementations we introduced in Section 4.4.2. In particular we acknowledge two inefficiencies affecting the kernels that employ a software cache of threadID-table entries backed by a hierarchial table.

In Section 4.3.4 we noted that a threadID-table cache hit automatically validates the thread identifier whose TCB is being located, provided cache entries are tagged (perhaps implicitly) using complete global thread identifiers rather than just thread numbers. Unfortunately none of our kernel implementations take advantage of this possibility. The end effect of this shortcoming is that all five of our physically-addressed kernels that use a threadID-table cache enjoy an unnecessary 2-cycle overhead by explicitly validating the given thread identifier even after the cache-lookup process results in a cache hit. A more efficient implementation would have intertwined this validation into the cache-lookup process.

A second shortcoming is that the backing stores used for the threadID-table caches are far from space efficient. At a minimum, the hierarchial tables used could store physical TCB addresses in a compressed 2-byte format. Furthermore, the backing stores used by the two kern_phys_merge*_cache kernels need not maintain copies of merged TCB fields themselves. Instead the threadID-table cache-refill process could simply obtain these merged fields from the actual TCB structure whose address is being cached. The performance overhead of these resource-usage improvements should be completely hidden when threadID-table cache hits are high. Of course if memory usage is of particular concern, then more advanced multi-level tables that employ path and level-compression techniques should be considered [44]. But investigating the performance and not space

efficiency of physically-addressed kernels is the principle objective of this thesis, and so we do not consider the shortcomings of our hierarchial tables cause for concern.

4.4.4 Implementation summary

Nine physically-addressed kernels featuring different threadID-table designs were presented in Section 4.4.2. The `kern_phys_4b` kernel is our most basic implementation in that it simply stores physical TCB addresses in a 4-byte format inside an array indexed by thread number. Its best-case performance is equal to that of the `kern_virt` kernel that addresses TCBs virtually. The `kern_phys_2b` kernel's purpose is to investigate the usefulness of expending an extra 4 cycles on the IPC fastpath in hope of reducing the threadID table's cache footprint by up to 50%.

The `kern_phys_merge` and `kern_phys_merge_store` kernels' purpose is to investigate the usefulness of merging TCB fields into the threadID table in hope of merging two cache-line fetches into one, but at the expense of 1–3 cycles overhead. By comparing the performance of these two kernels we can also determine if the benefits of maintaining a pointer to the currently executing thread's threadID-table entry are worth a 2-cycle performance penalty. The remaining five implementations are for investigating the effectiveness of a software-managed cache of threadID-table entries in concealing the lookup overhead of a more sophisticated but resource-friendly data structure.

IPC-fastpath performance

Kernel	Best-case IPC-fastpath cycle count		
	Call	Open-wait	Closed-wait
`kern_virt`	109	114	128
`kern_phys_4b`	109	114	128
`kern_phys_2b`	113	118	136
`kern_phys_merge`	110	115	130
`kern_phys_merge_store`	112	117	132
`kern_phys_cache_1way`	113	118	135
`kern_phys_cache_2way`	113–117	118–122	—
`kern_phys_cache_4way`	113–123	118–128	—
`kern_phys_merge_cache`	114	119	137
`kern_phys_merge_store_cache`	118	123	141

Table 4.4: Best-case IPC-fastpath performance of our kernel implementations on the MIPS R4700.

Table 4.4 summarises the best-case IPC-fastpath performance of our various kernel implementations on the MIPS R4700 when only hardware-register-mapped virtual registers are involved in the transfer. We include the best-case performance for all three different styles of IPC invocation: call, open-wait and closed-wait (non-call) IPCs. The open-wait IPC cycle counts correspond precisely to the measurements presented in Section 4.4.2.

Call IPCs are similar to open-wait IPCs in that they only require one threadID-table consultation for locating the send-phase destination's TCB. Call IPCs are, however, always 5 cycles faster than the analogous open-wait IPC invocation. This 5-cycle improvement is independent of the threadID-table design used, and in fact independent of the TCB-addressing method. The principle reason for this performance difference is that when a thread executes a non-call IPC operation, the kernel must ensure that thread is not being polled before it can execute the system call on the fastpath (see Section 4.2.1). This check is not required for call IPCs because, by definition, a call-IPC invoker is only willing to accept messages from the specified destination thread (`to-thread`). Hence it is irrelevant if a thread initiating a call IPC is being polled by some other thread.

Table 4.4 also provides the best-case IPC-fastpath cycle counts for closed-wait, non-call IPCs. Such invocations demand two thread identifiers be subjected to the TCB-lookup process. In particular this means two threadID-table consultations for physically-addressed kernels. Hence it is natural that closed-wait, non-call IPCs are more expensive than their call and open-wait IPC counterparts. Because the benchmarking environment we setup for evaluation purposes in the following chapter does not perform any closed-wait, non-call IPC operations, we neglect studying these system calls in detail. In fact we confidently assert that the presence of these IPCs in most L4-based systems is rare.

Data-cache footprint

Recall from Section 4.2.2 that an open-wait or call round-trip IPC between two threads touches 11 data cache lines, assuming UTCB-mapped virtual registers are not involved in the message transfer and the fastpath is used. By round trip we mean a ping IPC from thread `A` to thread `B`, followed immediately by a reply pong IPC from `B` to `A`. Assuming threadID-table cache hits, all but two of our physically-addressed kernel implementations increase this cache footprint by one or two cache lines. These additional cache lines are for indexing the threadID table to locate the TCBs of each thread participating in the round-trip IPC. When both thread's thread numbers index the same cache line of the threadID table, the increase is just the one cache line.

As previously hinted, two implementations are exempt from the above data-cache footprint increase. The `kern_phys_merge_store` and `kern_phys_merge_store_cache` kernels suffer precisely the same data-cache footprint as the `kern_virt` kernel (under the assumption of threadID-table cache hit rates for the latter). It is important to realise however, that although the size of the cache footprints are identical, the cache colouring differs. In Section 4.2.2 we rationalised why the fields falling on the first cache line of a TCB structure can only map to one of two possible data cache sets on the MIPS R4700. In contrast, the merged TCB fields read from a threadID-table can map to any one of the 256 cache sets in the R4700's primary data cache.

Instruction-cache footprint

Thus far we have focused primarily on data-cache footprint. Table 4.5 summarises the instruction-cache footprint for open-wait IPC system calls executing on the fastpath that transfer only register-mapped virtual registers. For implementations featuring a threadID-table cache, the instruction-cache footprints (and code sizes) presented assume cache hits. For non-direct-mapped threadID-table caches they assume a cache hit on the last cache entry searched within a set.

Kernel	Code size (bytes)	ICache coverage (# lines)
`kern_virt`	388	13
`kern_phys_4b`	388	13
`kern_phys_2b`	404	13
`kern_phys_merge`	392	13
`kern_phys_merge_store`	400	13
`kern_phys_cache_1way`	404	13
`kern_phys_cache_2way`	420	14
`kern_phys_cache_4way`	444	14
`kern_phys_merge_cache`	404	13
`kern_phys_merge_store_cache`	420	14

Table 4.5: IPC fastpath (open-wait) instruction-cache footprint of our kernel implementations.

Only three out of the nine physically-addressed kernels suffer any increase in instruction-cache footprint over the `kern_virt` kernel. One should keep in mind however that the `kern_virt` kernel's IPC fastpath executes only one more instruction beyond the capacity of 12 cache lines.

Chapter 5

Evaluation

Thus far this thesis has motivated the quest for addressing TCBs physically and described the performance trade-offs associated with this design choice. The preceding chapter presented concrete implementations of various physically-addressed kernels, each featuring a different threadID-table design, and analysed the impact of these kernels on the performance-critical IPC fastpath. In this chapter we proceed to evaluate the overall effect of our physically-addressed kernel implementations on the performance of an L4-based system. We aim to show that the simplicity gained by addressing TCBs physically can be enjoyed without any notable performance loss for at least the MIPS R4700 processor.

5.1 Factors Influencing Performance

Before we describe our evaluation environment and present benchmark results, we first discuss our evaluation objectives and summarise all the critical factors that influence the end-effect of a physically-addressed kernel's performance trade-offs. There are in fact three classes of such performance-influential factors we consider in our evaluation. We list these in the following:

1. Influence of the threadID-table design on the performance of a physically-addressed kernel.
2. Influence of the design of the operating-system personality executing on top of a physically-addressed L4 microkernel on overall system performance.
3. Influence of hardware architectural properties on the performance of a physically-addressed kernel, with particular attention paid to memory-management-unit properties.

In short our evaluation objective is to not only quantify the performance impact of a physically-addressed kernel, but to also understand the impact these three classes of influential factors has on that performance. We begin by discussing the significance of these influences.

5.1.1 ThreadID-table design

Chapter four introduced numerous design alternatives for a physically-addressed kernel's threadID table. These alternatives centred around minimising the potential performance impact of a threadID table's data-cache footprint and also around minimising a threadID table's resource usage. There are three specific design choices for the threadID table we are interested in evaluating. These objectives were first stated in Section 4.4.1 and are summarised below.

1. To evaluate the usefulness of compressing physical TCB addresses stored in a threadID table in hope of reducing a physically-addressed kernel's data-cache footprint, but at the expense of computational overhead required for decompressing these addresses.

2. To evaluate the benefit (if any) of merging TCB fields into threadID-table entries in hope of merging two data cache-line fetches into one (on the IPC fastpath). For this design choice we also seek to determine the usefulness of maintaining a pointer to the currently executing thread's threadID-table entry so that its merged TCB fields can be obtained without referencing the first cache line of its TCB structure.

3. To evaluate the effectiveness of a software-managed cache of threadID-table entries in concealing the cost of consulting a more sophisticated but resource-friendly backing store.

5.1.2 User-level operating-system design

A physically-addressed kernel is unique in the manner in which it translates thread numbers to TCB addresses. Hence the performance trade-offs associated with such a kernel will be more visible on systems that demand the microkernel perform a greater number of threadID-table consultations per timeslice. Since IPC is the most frequently executed microkernel activity, this effectively means the more IPC activity exhibited by an L4-based system, the more noticeable a physically-addressed kernel's performance impact. At one extreme, the two methods of addressing TCBs are likely to perform identically on systems void of any significant thread interaction.

It is not hard to see that the design of the operating-system personality executing on top of an L4 microkernel can influence the amount of threadID-table consultations performed by a physically-addressed kernel. For example, a system where user-level system-call servers dispatch tasks amongst a number of subservient worker threads is likely to involve more IPC message passing than a system where all system calls are directed to and handled by a single server.

It should be stressed, however, that not only should the frequency of IPC invocations be considered, but also the degree to which the endpoints of these IPC operations vary during a single timeslice. This is because each thread participating in IPC activity potentially requires either a TLB entry for its TCB or data cache lines for its threadID-table entry, depending on the kernel's TCB addressing method. Hence our physically-addressed kernels are more likely to perform differently than an unmodified L4Ka::Pistachio kernel in a multi-server environment where system calls are directed amongst a number of user-level servers.

5.1.3 Architectural properties

In Section 3.6.2 we argued that physically addressing TCB structures reduces pressure exerted on the system's TLB at the expense of a potential increase in data-cache footprint. Given this, the performance impact of a physically-addressed kernel will depend on the TLB and data-cache miss penalties of the system at hand. Such a kernel will perform more favourably on platforms where the TLB acts as a serious performance bottleneck and memory accesses are (comparatively) cheap.

Naturally, TLB and data-cache miss penalties are influenced by architectural properties of the underlying hardware. In fact on architectures featuring hardware-walked TLBs, the severity of these penalties is almost entirely determined by architectural properties. In the following we provide a survey of such properties, describing how each impacts TLB and data-cache performance. We pay particular attention to those factors that influence TLB behaviour because it turns out that TLB efficiency is the single most critical influence on the performance of a physically-addressed kernel.

TLB properties

Given that the TLB is an integral part of the memory-management subsystem on modern hardware, it is not surprising that architectural properties affect TLB-refill costs or even TLB-miss rates. The TLB's capacity trivially serves as an example. There are however more subtle properties we list below.

- A significant architectural factor influencing TLB-miss penalties is whether the TLB-refill process is conducted by hardware or software. Hardware-walked page-table architectures enjoy more efficient TLB refills at the expense of flexibility in page-table design. In fact hardware-managed TLB-miss penalties can be as small as 10–25 cycles whereas the cost of a TLB refill performed by software may exceed even 100 cycles [23]. Aside from the cost of traversing page tables, TLB misses handled by a software routine often suffer from additional overhead by requiring a complete pipeline flush on trap.

  The IA-32 [19], ARM [24] and 32-bit PowerPC [36] architectures walk page tables in hardware. All other architectures for which there exists an implementation of L4Ka::Pistachio feature software-managed TLBs. In particular this holds for the 64-bit platforms where the complexity inherent in page tables efficiently mapping large address spaces, and the flexibility demanded from such page tables (in providing superpage support for example), severely limits the suitability of hardware-refilled TLBs [8].

- When a user-level thread generates a TLB miss on a platform featuring a software-managed TLB, the processor traps into a TLB-miss exception handler. TLB-miss exception handlers are typically highly tuned because they are invoked frequently enough that overall system performance is tied to their execution speed.

  It is important to recognise however, that many of the TLB misses saved by a physically-addressed kernel are those that would otherwise have been generated by kernel execution. On some architectures, TLB misses generated during privileged execution trap to a general exception handler rather than a highly-tuned, specialised TLB-miss handler. General exception handlers must handle a variety of exception events that may include interrupts, system calls, and floating-point exceptions amongst others. Such handlers are thus inherently more inefficient in dealing with TLB misses when compared to specialised software routines.

  The MIPS R2000 processor provides an example of an architecture where all kernel TLB misses are directed to the general exception handler. Nagle et al. [37] determined that handling kernel-generated TLB misses on this processor running Mach 3.0 typically incurred a 300-cycle penalty, whereas TLB misses from user level costed as little as 20 cycles on the same system. One would expect the performance of a physically-addressed kernel on this and similar processors where handling kernel TLB misses are especially expensive, to be particularly favourable.

- The two TLB properties we have presented thus far affect the severity of TLB-refill penalties. We now provide an example of an interesting architectural design choice for TLBs that affects TLB-miss rates. It proves particularly relevant to the evaluation we conduct on the MIPS R4700 in the remainder of this chapter.

  Support for *subblocking* in a TLB entails mapping an aligned block of multiple, contiguous pages per TLB entry [45]. The number of pages mapped by each entry in a subblocked TLB is referred to as the *subblocking factor*. The MIPS R4700's provides an example of a TLB supporting subblocking in that each TLB entry maps an adjacent even-odd pair of virtual pages. That is, the MIPS R4700's TLB exhibits a subblocking factor of two.

  A subblocked TLB uses fewer tags by associating a single tag with multiple mappings. This allows for constructing larger TLBs for a fixed area size on chip. The performance of a subblocked TLB depends largely on the spatial locality exhibited by the processor's workload. When spatial locality is high, a

program is more likely to access consecutive pages and so the TLB-miss handler effectively implements a primitive form of prefetching by inserting multiple page-table entries into the TLB during a single refill operation. In the worst case, however, a subblocked TLB's coverage may be reduced by a factor equal to its subblocking factor.

For an L4 kernel that locates TCB structures by indexing a large TCB in virtual memory, the likelihood of a large number of pages being referenced in each TLB block is almost entirely determined by the distribution of thread numbers attached to threads participating in IPC activity. Because in general there is no reason to expect a group of threads conducting IPC in an L4-based system to share numerically similar thread numbers, spatial locality of references to pages in a Pistachio kernel's TCB array is unlikely to hold as strongly as that exhibited by typical user programs. For this reason we confidently assert that subblocking is more likely to adversely affect the TLB performance of an L4 kernel that addresses TCBs virtually, and hence in comparison a physically-addressed kernel should perform more favourably on such systems.

Memory-cache properties

A physically-addressed kernel's potential increase in data-cache footprint is more likely to adversely affect overall system performance on those architectures where accessing memory is costly. Of the three TLB properties we introduced in the preceding discussion, the latter two had distinct performance implications specifically for TCB footprint in the TLB. In contrast there are no common data-cache properties that have a particularly profound corollary for the specific case of threadID-table footprint in the data cache. Hence we merely state that any architectural property that improves the average latency of main-memory references induces more favourable conditions for a physically-addressed L4 kernel to perform in. For example, it is more likely that addressing TCBs physically will offer performance improvements over virtually addressing TCBs in the presence of a second-level cache or a fast memory bus.

5.2 Evaluation Environment

In this section we describe the hardware and software environment we use for benchmarking and evaluating the performance of a physically-addressed L4 kernel.

5.2.1 The U4600

The U4600 machine was developed at the University of New South Wales by Kevin Elphinstone and Dave Johnson. It features the MIPS R4700 processor first introduced in Section 4.1. The processor is clocked at 100MHz and features no co-processors other than the system co-processor and floating-point unit. The U4600 machines run diskless but include a PCI network card used for loading boot images. Our particular box was fitted with 128MB of 80ns DRAM. It lacked a secondary-level cache and had a cache-miss penalty of 16 cycles best case and 18 cycles average.

As first argued in Section 5.1.3, any performance comparison between the two competing TCB-addressing methods will be sensitive to architectural properties. With this in mind, it is worth noting here that our hardware platform provides an almost best-case environment for a physically-addressed kernel to perform in (at least amongst those architectures for which a port of L4Ka::Pistachio is available). The small software-loaded TLB is notorious for functioning as a serious performance bottleneck on the R4700, and the subblocking property is only likely to exacerbate this when TCBs are virtually addressed. Furthermore, the modest processor clock speed is largely responsible for a relatively cheap cache-miss penalty when compared to other processors (100-cycle cache-miss penalties are not uncommon). The lack of a secondary-level data cache in the U4600s would be the chief complaint of a physically-addressed kernel.

5.2.2 Microkernel

The L4 microkernels we benchmark were first introduced in Section 4.4. Ten kernels were presented, of which nine addressed TCBs physically. The tenth kernel (`kern_virt`) only differed from the remaining nine in that it addressed TCBs in virtual memory. We stated in Section 4.4.2 that the `kern_virt` was identical to the Version 0.4 release of L4Ka::Pistachio on the MIPS64, with the exception of two subtle but important modifications. We describe these changes in the following.

TCB layout

A primary goal of this thesis is to analyse the performance penalty a physically-addressed kernel may potentially incur due to increased pressure exerted on the system's data cache. This potential penalty will be most visible when the microkernel's cache footprint has been optimised to cover the least amount of cache lines. This is because for such microkernels, the increase in cache usage attributed to threadID-table references will constitute a larger fraction of the overall cache footprint.

That TCB data constitutes a dominant component of the IPC fastpath's (and hence microkernel's) cache footprint was illustrated in Section 4.2.2. To minimise this footprint, the layout of TCB fields (and in particular those referenced by the IPC fastpath) must be carefully arranged. It is not surprising that the optimal TCB layout varies across architectures. For one, the IPC fastpath must perform a thread switch — a mechanism dictated almost entirely by the underlying architecture. A MIPS64 kernel for example must read the target thread's address-space identifier and place it in a co-processor register where it is used by the processor to match against TLB entries. It must also store a thread's user-accessible UTCB address into the `k0` general-purpose register on thread switch in accordance with the L4 Version 4 ABI. These two MIPS64-specific kernel functions necessitate the referencing of two TCB fields on the IPC fastpath (`asid` and `myself_local`) that on other architectures may be completely neglected (or not even exist).

It is a weakness of L4Ka::Pistachio's design that the TCB layout is defined in the platform-independent part of the source. For historic (and even political) reasons, this layout has been optimised for the Intel IA-32 processors at the expense of other architectures. Hence before conducting our evaluation, we ensured to adjust the TCB structure definition so as to obtain an optimal TCB layout for the MIPS R4700. Describing the details of each adjustment made does not steer us any closer to the objectives of this thesis. Hence we merely state that the TCB layout used in our implementations (and the layout on which the analysis conducted in Section 4.2.2 was based) reduced the number of TCB-field cache lines touched by the IPC fastpath from four to two per TCB accessed. For the curious, a description of our TCB layout can be found in Appendix A.

TLB-miss handler

A kernel that addresses TCBs in virtual memory suffers from a (potentially) higher TLB-miss rate due to TCB-occupied pages competing for TLB entries. Hence in comparing the performance of such a kernel with that of a physically-addressed kernel on the MIPS R4700, the TLB-miss exception handler should be made as efficient as possible so as to ensure any obtained measurements are not biased by poor implementation.

Unfortunately the TLB-miss handler in the stock MIPS64 L4Ka::Pistachio v0.4 kernel walks a generic multi-level radix table in C++ code and so not surprisingly performs woefully inadequately — its best-case performance for handling a non-page-fault TLB miss was measured in excess of 300 cycles. To address this, the kernels we subject to benchmarking all feature a software TLB or *STLB* backed by a simple three-level hierarchial table. The STLB is effectively a global software-managed direct-mapped cache of 8192 TLB entries. Each (valid) STLB entry maps an even-odd pair of pages and is tagged with both an address-space identifier and the virtual page number (divide by two) of the even mapping. The STLB is indexed using only virtual page-number bits. In particular the address-space identifier is given no weight in the indexing function and so identical pages

in different address spaces always map to the same STLB cache entry. On an STLB miss, the appropriate entry is reloaded from the per-address-space hierarchial-table that serves as a backing store.

The performance of an STLB in a microkernel environment (on the R4x00) has been studied previously [7, 44] and shown to be a highly attractive option for use by a microkernel. In fact our implementation of the STLB is almost identical to the implementation used by the L4/MIPS [15] Version 2 API kernel that has been reported [44] to handle TLB misses in as little as 23 cycles (when discounting kernel entry and exit overhead). In comparison, our STLB implementation suffers a handicap of 2 cycles because any exception handler in a MIPS64 kernel adhering to the L4 Version 4 specifications is deprived the unhindered use of the `k0` register which the ABI reserves for the current thread's user-accessible UTCB address. Hence our TLB-miss exception handler must save and restore an additional register which raises its best-case performance to 25 cycles (discounting mode-switch overhead).

On an STLB hit, the TLB-miss exception handler touches only two data cache lines. One cache line is for the STLB entry indexed by the faulting address, and the second is for preserving three general-purpose registers that would otherwise be trashed by the handler. The cache line used by the TLB-miss handler for saving (and restoring) registers to has been deliberately aligned to not conflict with the cache sets touched by the IPC fastpath (at least when UTCB-mapped virtual registers are not being transferred).

It turns out that in our evaluation, the STLB performs sufficiently well that it effectively conceals the cost of traversing the hierarchial-table backing store. Hence we deliberately neglect to describe the STLB-refill process in detail, and simply offer the best-case non-page-fault STLB-miss performance of our TLB-miss exception handler as 88 cycles (excluding kernel-mode entry and exit but including the cost of refilling the STLB).

5.2.3 Linux on L4

Running Linux on top of the L4 microkernel is an attractive option as it immediately provides us with a wealth of applications and standardised benchmark suites for use in evaluating the performance trade-offs associated with physically addressing TCBs. Linux on L4 was pioneered by the Dresden University of Technology in developing L^{4}Linux [13] — a port of Linux 2.0 to the L4 microkernel. Although L^{4}Linux ports for the Linux 2.2 and 2.4 kernels have since been developed, L^{4}Linux is nevertheless focused entirely on the Intel IA-32 platform and hence not suitable for our use. Fortunately the *Wombat* Linux 2.6 server developed by National ICT Australia provides a MIPS64 port in addition to IA-32 and ARM ports, and so fulfills our needs comfortably.

Iguana and Wombat framework

The Wombat Linux server runs on top of the *Iguana* resource-manager framework that in turn runs on top of L4. Iguana is responsible for providing a basic set of services such a memory protection model, facilities for sharing memory and a general resource manager.

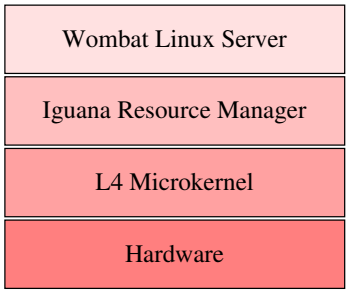

Figure 5.1: Iguana and Wombat servers.

The design aspect of Iguana most relevant to our evaluation is its support for minimising address-space overlap by making use of single-address-space operating-system techniques. The rationale for this design choice is to avoid flushing virtually-addressed caches when switching between processes. This is important for Wombat which targets embedded devices (with memory-management units) amongst which the ARM family of processors utilising virtually-addressed caches is very prominent. Although embedded systems are not specifically relevant to our evaluation, it is important to note that our STLB cache performs optimally when process address spaces do not overlap (recall from Section 5.2.2 that the address-space identifier bits of a faulting address do not take part in the STLB-indexing function).

Linux system-call convention

Linux user processes are implemented in Wombat as L4 threads executing in their own address space. The Wombat server acts as the pager, scheduler and exception handler for all such threads. Linux and L4 MIPS64 kernels both share the same system-call convention, but use non-overlapping system-call numbers. Hence when a Linux user process performs a Linux system call, the microkernel recognises the supplied system-call number as invalid and invokes L4's exception-handling mechanism which forwards an exception message from that thread to the Wombat server via IPC. The IPC message received by the Wombat server appears as though it was sent by the L4 thread implementing the Linux process that issued the Linux system call. The exception message contains enough of the exception-raising thread's execution state for Wombat to handle the system call and return the results directly to the Linux process via IPC. In summary, Linux system calls are implemented via two IPC messages, the first of which is fabricated by the microkernel on the Linux user process's behalf, and hence often referred to as a *trampoline*.

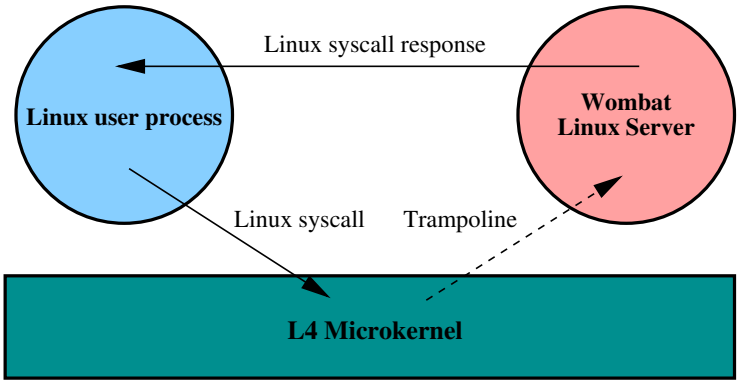

Figure 5.2: Linux system-call convention implemented via trampoline.

Unfortunately it is a weakness of the current implementation of Pistachio on the MIPS R4700 that any IPC fabricated by the microkernel (such as page-fault or exception IPCs) or any reply to such an IPC can only be handled by the IPC slowpath. Without overcoming this limitation, any measurements obtained by our evaluation would fail to provide a representative picture of a physically-addressed kernel's true performance. Although modifying Pistachio to directly overcome this limitation is possible (but non-trivial), we chose instead to modify the C library Linux user processes were linked against so as to implement a Linux system-call convention that directly performs an L4 IPC call invocation to the Wombat server. In fact we deliberately fine tuned the contents of the message transferred by this IPC so as to contain only the minimal amount of exception state necessary for the Wombat server to correctly handle the system call. In the end, the Linux system-call convention used by our Linux processes involved a round-robin fastpath-handled call IPC between user process and Wombat server that transferred only hardware-register-mapped virtual registers in both directions.

5.3 Evaluation Methodology

5.3.1 Benchmarks

Constraints

Running Linux on top of L4 provides us with a plethora of applications for use in evaluating our physically-addressed kernels. A set of benchmarks with a diverse range of working-set sizes and varying amounts of thread activity would be most suited for our purpose. Unfortunately we were restricted to the use of only two such benchmarks by time and practicality constraints.

To understand the latter constraint, recall that the U4600 boxes run diskless. Hence the Linux file system used in our evaluation environment was severely limited in size so that in could be loaded and maintained entirely in memory as a ramdisk. Of the 128MB of memory installed in our U4600 box, 80MB was reserved for the Wombat and Iguana servers and 16MB was reserved for use by the microkernel itself. Hence our root file system was limited to 32MB in size. With this in mind, we describe the two benchmarks used in evaluating our kernels in the following.

Kernel compilation

A traditional method of benchmarking Linux servers is to measure the time required to compile a Linux kernel from source using the GNU Compiler Collection (GCC). It has been well established that the working-set size (in both TLB and instruction and data caches) involved in such a compilation is sufficiently large to stress a given system's memory-subsystem components. This in conjunction with an expectation of a moderate-to-high level of system-call activity makes this benchmark particularly suited to our needs. Additionally, by running the compilation process entirely in memory, our timing results are not prone to bottlenecks or fluctuations induced by disk activity.

The major caveat to compiling a modern Linux kernel on the U4600 is that the code base far exceeds 32MB in size. Even when compiling older Linux kernels with unnecessary drivers and irrelevant architecture-specific code removed, the combined source and build tree grows beyond our modest limit. For this reason we chose instead to benchmark compiling the L4Ka::Pistachio microkernel from source. This proved highly apt as the modest 4MB code base was small enough to fit into our ramdisk (alongside GCC and system-related files) yet large enough to provide a substantial workload for the R4700 processor (Pistachio compiled in no less than 12 minutes for all benchmark configurations used in our evaluation).

AIM7 multiuser benchmark suite

The AIM7 multiuser benchmark suite uses a technique called *load mix modelling* to measure the performance of a system under different application workloads. The benchmark executes a fixed number of processes throughout its lifetime with each process executing the same set of *jobs*, but possibly in different order. A job is a unit of work that may vary from performing matrix multiplication to traversing directory hierarchies. A *workfile* defines the precise set of jobs that is executed by each running AIM7 process. By appropriately selecting the jobs contained in a workfile, the benchmark may be customised to simulate the workload of a desired multiuser system (such as a compute or login server).

In choosing a workfile for our evaluation, we targeted jobs that exhibited the highest level of system-call (and hence L4 IPC) activity (in terms of number of system calls performed per second rather than percentage time spent handling Linux system calls). In the end, a workfile simulating a file-server workload was selected for use in evaluating our physically-addressed kernels. The workfile's primary drawback was the lack of a large working set. Unfortunately the jobs in the AIM7 benchmark suite with the larger working sets are mostly purely computational and hence void of any system-call activity.

As previously mentioned, the AIM7 benchmark spawns a certain number of processes that each execute a list of jobs specified by a given workfile. A conventional AIM7 benchmark run steadily increases the number of such spawned processes until the subject system's peak throughput (in terms of jobs completed per minute) is determined. Unfortunately running the benchmark in this way results in a completion time of over 12 hours — an unreasonable amount given our time constraints and the number of benchmark runs required to evaluate ten kernels. Instead we ran the benchmark conventionally just once and noted that the U4600 performed within 1.0% of its peak throughput with anywhere between 2 and 8 processes running simultaneously. Hence in evaluating our physically-addressed kernels with the AIM7 benchmark suite, we simply fixed the number of spawned processes at 4 and measured the time required for the 4 processes to execute all their jobs. This style of benchmark run enjoyed a completion time of less than 15 minutes, and hence was much more appropriate for our use.

5.3.2 Cache behaviour

In evaluating the behaviour of a physically-addressed kernel, it is appropriate to ensure that the performance impact attributed to threadID-table cache footprint in the system's data cache is not concealed by pathological cache behaviour. The L4 microkernel running on the MIPS R4700 processor is unfortunately highly susceptible to such pathological behaviour, given that there is a 50% chance that any two thread's TCB fields will be mapped to the same two-way-associative data cache sets. For example, a Linux thread invoking many system calls is likely to perform much more poorly if its TCB is mapped to the same cache sets as the Wombat server's TCB than if its TCB is mapped to the other half of the R4700's data cache.

In our evaluation we ensure to not only avoid pathological data-cache behaviour where most or all Linux user thread's TCBs are mapped to the same data cache sets as the Linux Wombat server's TCB, but also ensure that in multiple runs of a fixed benchmark (kernel compilation or the AIM7 benchmark suite), a thread's TCB is mapped to the same half of the R4700's data cache in each run. In particular we ensure this holds irrespective of the microkernel used. Doing so is not difficult but not trivial either. Because the R4700's data cache is virtually indexed, the data cache sets a thread's TCB occupies is determined by its thread number when TCBs are addressed virtually. On the other hand when TCBs are physical objects, the data-cache half a thread's TCB is mapped to is determined by the physical address of that thread's TCB which is purely dictated by the kernel memory-pool allocator.

For our evaluation we modified the kernel memory-pool allocator in each of our physically-addressed kernels so that it allocated TCB structures with the appropriate alignment that forced a TCB to occupy the same data cache sets it would have occupied if it were addressed virtually. Hence for all microkernels evaluated and all benchmarks ran, the data-cache half a thread's TCB occupied was always determined solely by that thread's thread number. This ensured repeatability of benchmark results and allowed for more meaningful comparisons to be made between measurements obtained from running physically-addressed and virtually-addressed kernels.

Furthermore, by controlling the allocation of thread numbers to Linux user threads, we guaranteed that no Linux user thread's TCB competed with the Wombat server's TCB for data-cache entries (by assigning an odd thread number to the Wombat server but an even thread number to all Linux user threads). Under such conditions, any performance penalty attributed to a physically-addressed kernel's increased data-cache usage is more likely to be visible in timing measurements and less likely to be concealed by pathological cache behaviour resulting from TCB-related conflict misses.

Before we conclude this section, we note that pathological TLB behaviour is not a grave concern due to the MIPS R4700's TLB being fully associative. We state however than in no benchmark ran during our evaluation of the `kern_virt` kernel did a Linux user thread's TCB occupy the same TLB subblock as the Wombat server's TCB. Hence any `kern_virt` IPC operation resulting from a Linux user thread issuing a Linux system required 2 (out of 48) TLB entries for mapping TCB structures. In fact no two Linux user thread's TCBs occupied the same TLB subblock either (due to all such thread's having an even thread number). This had no bearing on our

results, however, as within a timeslice only zero or one Linux user threads were ever engaged in L4 IPC activity.

5.3.3 Measurements

Given the qualitative performance trade-off analysis conducted in Section 3.6.2, fully understanding the performance impact a physically-addressed kernel has on an L4-based system requires at a minimum the following measurements be made available in addition to any benchmark-duration timing results:

1. Time spent traversing threadID-table structures, discounting memory-cache miss penalties.

   This measurement quantifies the algorithmic cost of indexing threadID-table structures.

2. Time spent servicing data-cache misses resulting from threadID-table accesses and time spent servicing data-cache misses caused by threadID-table data displacing otherwise useful data cache lines.

   These measurements quantify the overall performance impact attributed to a physically-addressed kernel's increased data-cache footprint.

3. Time saved by not incurring TLB misses resulting from TCB accesses and time saved by not incurring TLB misses caused by TCB mappings displacing otherwise useful TLB entries.

   These measurements quantify the performance penalties a physically-addressed kernel avoids by not requiring a virtual mapping for each TCB accessed.

The data in the first category is not difficult to obtain given the analysis of each kernel's performance impact on the IPC fastpath that was conducted in Section 4.4.2 — in fact Table 4.4 is a convenient reference for comparing the algorithmic cost of translating thread numbers to TCB addresses for our various kernels. All that remains is to instrument the kernel to maintain a running count of the number of threadID-table lookups performed during a benchmark.

Unfortunately measurements in the latter two categories are more difficult to obtain without hardware support for profiling — a feature lacking in the MIPS R4700 processor. It was hoped that representative simulation would fill this void, but unfortunately the sheer amount of time required to simulate either of our two chosen benchmarks to completion deemed this course of action infeasible. Nevertheless a third option still remained for measuring some (but not all) aspects of TLB performance.

Instrumentation

Since the MIPS R4700 features a software-managed TLB, its TLB-refill handler can be instrumented directly to maintain statistical data pertaining to TLB-miss rates and penalties. Specifically, in conducting our evaluation, we modified the `kern_virt`'s TLB-refill handler to maintain a running count of the number of non-page-fault TLB misses and the total cycle count required for servicing those misses (excluding the cost of performing the actual instrumentation). In fact we maintained a separate TLB-miss count for misses on TCB-filled pages. This was so we could determine what percentage of any additional TLB misses the `kern_virt` kernel incurred over the physically-addressed kernels could be attributed directly to TCB-mapped virtual pages.

Modifications were also made for profiling IPC activity and threadID-table lookups (including threadID-table cache-hit rates). This data was required for validating our expectation that the IPC fastpath would perform the majority of threadID-table lookups, and also for examining threadID-table cache behaviour.

All data accumulated through instrumentation was stored in uncached unmapped memory so that TLB and memory-cache miss rates were not influenced. Of course in timing the actual benchmarks to completion, instrumentation was completely disabled. Hence a second benchmark run was used for profiling kernel behaviour.

Unfortunately our work failed to directly measure the severity of the data-cache miss increase a physically-addressed kernel potentially incurs. We also failed to quantify the indirect performance penalty the `kern_virt`

kernel suffers by polluting the system's memory caches when servicing TLB-miss exceptions that would not have been raised by a physically-addressed kernel. Despite these shortcomings, we believe the following analysis still sheds much light on the performance impact a physically-addressed kernel has on an L4-based system.

5.4 Results & Analysis

We now turn to presenting and analysing the results obtained from benchmarking physically-addressed kernels' overall performance impact on an L4-based system.

Statistical accuracy

In this section average values for measurements obtained by running the appropriate benchmark a total of four times are presented in table form and later analysed. Conducting more repetitions would have been preferable, but time constraints restricted us to only four. The deviations within each set of measurements obtained from multiple runs however was surprisingly very small. In every set of four such measurements, the maximum deviation never exceeded 0.43% of the averaged value (we feel reporting the maximum rather than the standard deviation is most appropriate given the small number of repetitions conducted). We attribute this small deviation largely to use of a ramdisk eliminating fluctuations and bottlenecks caused by I/O activity. Given the small deviations, we feel justified in omitting references to averages and deviations in the result-tables provided below.

5.4.1 Benchmark Set #1

This section evaluates the performance of those physically-addressed kernels that do not feature a threadID-table cache. We postpone an evaluation of the effectiveness of threadID-table caches to Benchmark Set #2. The evaluation environment and methodology used here is precisely that described in Sections 5.2 and 5.3.

For each of the kernel-compilation and file-server-simulation benchmarks executed, three tables worth of results are produced. The first table provides timing measurements for benchmark runs and also indicates the average cost of fastpath-handled IPCs during those runs. The latter two tables display profiling data for TLB performance and IPC activity respectively. For these tables, a single row represents the behaviour of all physically-addressed kernels evaluated. This is because the profiling data contained in the last two tables is independent of the threadID-table design used by a physically-addressed kernel.

It should be stated that the cycle counts for (average) IPC-fastpath and TLB-refill performance provided in the tables below include the cost of entering and leaving kernel mode. In particular when comparing the IPC-fastpath cycle counts with those presented in Table 4.4, one must deduct the 10-cycle overhead required for entering and leaving kernel mode from the figures found in this section (the MIPS R4700 incurs a 5-cycle penalty on each mode transition for flushing the processor pipeline).

Results — Simulation of a file-server workload with AIM7

Benchmark Results		
Kernel	Completion time	Avg. IPC-fastpath cost
`virt`	782 seconds	135 cycles
`phys_4b`	781 seconds	154 cycles
`phys_2b`	780 seconds	158 cycles
`phys_merge`	781 seconds	143 cycles
`phys_merge_store`	779 seconds	133 cycles

Table 5.1: Benchmark results for simulating a file-server workload.

Profiling Results — TLB performance				
Kernel	# Misses ($\times 10^6$)	% Misses on TCBs	STLB hit rate	Avg. refill penalty
`virt`	40.8 (+0.00%)	2.92%	98.0%	60.6 cycles
`phys_*`	38.2 (-6.35%)	0.00%	99.5%	57.8 cycles

Table 5.2: Profiling results for TLB behaviour when simulating a file-server workload.

Profiling Results — IPC Activity			
Kernel	# IPCs ($\times 10^6$)	% IPCs on fastpath	% ThreadID-table lookups for IPC
`virt`	21.4	99.9%	-
`phys_*`	21.4	99.9%	99.9%

Table 5.3: Profiling results for IPC activity when simulating a file-server workload.

Results — Compilation of L4Ka::Pistachio source with GCC

Benchmark Results		
Kernel	Completion time	Avg. IPC-fastpath cost
`virt`	735 seconds	725 cycles
`phys_4b`	722 seconds	586 cycles
`phys_2b`	720 seconds	585 cycles
`phys_merge`	721 seconds	585 cycles
`phys_merge_store`	725 seconds	583 cycles

Table 5.4: Benchmark results for kernel compilation with GCC.

Profiling Results — TLB performance				
Kernel	# Misses ($\times 10^6$)	% Misses on TCBs	STLB hit rate	Avg. refill penalty
`virt`	148.7 (+0.00%)	2.47%	99.0%	62.2 cycles
`phys_*`	132.8 (-10.7%)	0.00%	99.4%	59.9 cycles

Table 5.5: Profiling results for TLB behaviour when compiling a kernel with GCC.

Profiling Results — IPC Activity			
Kernel	# IPCs ($\times 10^6$)	% IPCs on fastpath	% ThreadID-table lookups for IPC
`virt`	1.87	24.4%	-
`phys_*`	1.87	24.4%	96.7%

Table 5.6: Profiling results for IPC activity when compiling a kernel with GCC.

Analysis objectives

We now turn to analysing the results contained in the preceding tables for both the file-server-simulation and kernel-compilation benchmarks. To begin with note that the benchmarking completion times show that all the physically-addressed kernels evaluated performed within 0.38% and 2.0% of the `kern_virt` kernel's standard for the AIM7 and GCC benchmarks respectively. The analysis we conduct here culminates by arguing that these discrepancies, although small, are accounted for effectively entirely by differences in TLB-miss rates. We begin, however, by focusing our analysis on IPC activity and then TCB performance alone.

Analysis of IPC activity

The AIM7 file-server-simulation benchmark featured a very high rate of L4 IPC invocations with one such system call occurring every $\frac{780}{21.4} = 36.4$ microseconds. The GCC kernel-compilation benchmark's IPC activity was more modest with one occurring only roughly every $\frac{730}{1.87} = 390$ microseconds. Given that the 100MHz-clocked U4600 boxes execute at most 100 cycles per microsecond, both benchmarks exhibit a level of IPC activity not unreasonable for evaluating a physically-addressed kernel's performance. Having said this, a slightly higher IPC invocation rate for the GCC benchmark would have been welcomed (we were expecting a figure closer to 100 μs than to 400 μs).

Almost all (99.9%) AIM7-benchmark IPC invocations were executed on the fastpath, and for the physically-addressed kernels, these in turn accounted for effectively all the threadID-table lookups. The GCC benchmark was not as well behaved.

Surprisingly only 24.4% of IPC operations during a kernel compile were candidates for the fastpath. Despite this, L4 IPC was still responsible for as much as 96.7% of threadID-table consultations performed by the `kern_phys_*` kernels (the remaining threadID-table lookups are accounted for by THREADCONTROL and SPACECONTROL L4 system calls invoked by the Wombat Server for spawning Linux threads).

Further instrumentation of our kernels revealed that page faults were responsible for most of the remaining 75.6% IPC invocations not handled by the fastpath during a kernel compile. As first discussed in Section 5.2.3, the MIPS64 L4Ka::Pistachio implementation's IPC fastpath is not equipped to deal with IPCs generated by the kernel, or replies to such IPCs. In particular page-fault-related IPCs are always executed on the slowpath by our kernels. Unfortunately we did not have time to invest in rationalising the abundance of page faults generated by a kernel compile. Nevertheless the results obtained from this benchmark still prove highly enlightening.

Analysis of IPC performance

The average cycle count required for handling L4 IPC's on the fastpath during a kernel compile was very poor, indicating a very high memory-cache miss rate. It would certainly be reasonable to expect an instruction-cache miss rate of close to 100% given that instructions on the fastpath were executed very infrequently by the GCC benchmark. A high data-cache miss rate would not be surprising either. Compiling a kernel resulted in an IPC being invoked roughly only every 39,000 cycles, which is plenty of time for the GCC compiler's large working set to map any IPC-related data out of the system's caches. The abundance of memory-cache and TLB misses completely conceals the algorithmic cost of translating thread numbers to TCB addresses on the fastpath during a GCC benchmark run. Also note that the difference in IPC fastpath performance between the `kern_virt` kernel and the physically-addressed kernels is certainly accounted for by each such system call raising a TLB-miss exception in the `kern_virt` kernel for both TCBs referenced (given that the number of TLB misses incurred per IPC can be approximated by $\frac{148.7 \times 0.0247}{1.87 \times 0.967} = 2.0$).

In contrast to GCC, IPC-fastpath performance during an AIM7 benchmark run did not differ from the best-case figures (reported in Table 4.4) by more than 5–24%. The `phys_2b` and `phys_merge_store` kernels serve as examples of the different extremes. That this is the case is not surprising given that L4 IPC occurs an order of magnitude more frequently when simulating a file-server workload than when compiling a kernel.

The kern_virt kernel's average IPC-fastpath cycle count was 135 (for the AIM7 benchmark) which is 16 cycles greater than the best-case figure of 119 cycles. On the other hand the kern_phys_4b kernel's average IPC-fastpath performance is 35 cycles more expensive than its best-case figure, which strongly indicates this physically-addressed kernel's fastpath incurred an additional cache miss when compared to the kern_virt kernel's fastpath. Of course this extra cache miss is almost certainly attributed to the single threadID-table lookup performed when handling a call IPC.

The kern_phys_2b kernel's AIM7 IPC-fastpath performance was on average 4 cycles more expensive than for the kern_phys_4b kernel. Given the best-case figures in Table 4.4, we confidently assert that these extra 4 cycles are attributed to the algorithmic cost of decompressing 2-byte entries in the kern_phys_2b kernel's threadID table into valid addresses. In particular this indicates that the address-compression technique used by the kern_phys_2b kernel did not eliminate any of the threadID-table cache misses the kern_phys_4b kernel incurred. This is not surprising because in all our benchmark runs, the thread numbers assigned to Linux user threads were contrived so that the corresponding threadID-table entries never fell on the same data cache line occupied by the Wombat server's threadID-table entry.

Of all the average IPC-fastpath cycle counts on display in Table 5.1, the kern_phys_merge_store kernel's data is most interesting in that its IPC-fastpath performance is not only closest to its best-case value but also outperforms the kern_virt kernel's IPC fastpath by 2 cycles. Given that the algorithmic cost of translating a thread number to a TCB address in the kern_phys_merge_store kernel is 3 cycles more expensive than the analogous kern_virt-kernel cost, the kern_phys_merge_store kernel saves 5 cycles on memory-cache and TLB miss penalties for each fastpath IPC when compared to kern_virt. We now turn to determining how many of these 5 cycles are due to memory-cache miss savings.

During an AIM7 kern_virt benchmark run, one in every $\frac{21.4}{40.8 \times 0.0292} = 18.0$ IPC invocations raised a TLB-miss exception. Hence the average TLB-miss penalty a fastpath IPC suffered from during an AIM7 benchmark run on the kern_virt kernel was $\frac{60.6}{18.0} = 3.37$ cycles. Hence the kern_phys_merge_store kernel saved at least 1.63 (and at most 5) cycles worth of memory-cache miss penalties over the kern_virt kernel per fastpath IPC handled. Since these two kernels touch the same number of cache lines per fastpath IPC, the memory-cache misses saved on the IPC fastpath by kern_phys_merge_store when compared to kern_virt are mostly likely attributed to that kernel's more flexible cache colouring of the first TCB cache line (as explained in Section 4.4.4).

The kern_phys_merge kernel's average AIM7 IPC performance on the fastpath was 23 cycles more expensive than the corresponding best-case value. Comparing this to the 35-cycle overhead the kern_phys_4b kernel's AIM7 IPC fastpath incurred beyond its best-case figure indicates that the kern_phys_merge kernel was successful in eliminating data-cache misses by merging TCB fields into its threadID table. The magnitude of this success however was not as strong as that enjoyed by the kern_phys_merge_store kernel.

Analysis of TLB performance

Time spent handling TLB misses when simulating a file-server workload was $40.8 \times 0.606 = 24.7$ seconds and $38.2 \times 0.578 = 22.1$ seconds for the kern_virt and kern_phys_* kernels respectively. The analogous measurements for the kernel-compilation benchmark show a much greater difference in TLB performance between the two categories of kernels under investigation — kern_virt spent 92.5 seconds handling TLB misses when the physically-addressed kernels spent only 79.5 seconds. This is not surprising given our expectation of a much larger TLB working set for a GCC benchmark run than for an AIM7 benchmark run. In fact in compiling a kernel with GCC, the R4700 processor spent 11–12% of its time in the TLB-refill handler indicating that the TLB was acting as a serious bottleneck to performance. In contrast the processor spent only 3% of the time handling TLB misses during an AIM7 benchmark run.

The kern_virt kernel incurred 6.35% and 10.7% more TLB misses for the AIM7 and GCC benchmarks respectively when compared to the physically-addressed kernels. Naturally the increased pressure virtual TCB

mappings exert on the kern_virt kernel's TLB is responsible for this increase. Of the additional TLB misses kern_virt suffers from, $\frac{40.8 \times 0.0292}{40.8 - 38.2} = 45.8\%$ (AIM7) and $\frac{148.7 \times 0.0247}{148.7 - 132.8} = 23.1\%$ (GCC) occur on TCB-occupied pages. This suggests that for each kernel-generated TLB miss (on a TCB), an average of $\frac{1 - 0.458}{0.458} = 1.1$ or $\frac{1 - 0.231}{0.231} = 3.3$ otherwise useful user pages were displaced from the TLB during an AIM7 or GCC benchmark run on kern_virt. The differences here are reasonable given that previous figures strongly indicate GCC's but not AIM7's TLB working set constitutes a challenge for the MIPS R4700 processor. If the GCC benchmark enjoyed the same level of IPC activity exhibited by the AIM7 benchmark, the disparity between TLB-miss rates for the two competing methods of addressing TCBs during a kernel compile would be substantially greater.

The average TLB-refill penalty for all benchmark runs fell within a range of $57.8 - 62.6$ cycles. A likely best-case TLB-refill penalty given our refill-handler implementation is $25 + 10 + 18 = 53$ cycles when taking into account pipeline flushes and assuming a data-cache miss when successfully indexing the STLB. Hence the TLB performance of our benchmark runs fell within half a memory-cache miss penalty of a plausible best-case figure. This is not unreasonable.

It is interesting to note that the kern_virt enjoyed a marginal but distinct increase in average TLB-refill cost over the physically-addressed kernels. On the surface this is surprising given that the kern_virt kernel incurs a higher TLB-miss rate (and so its memory caches are more likely to contain data required by the TLB-refill handler). The figures in Tables 5.2 and 5.5 suggest this peculiarity is caused by a stronger STLB hit rate in the physically-addressed kernels. We attribute the different STLB hit rates to the fact that the STLB is only indexed with the lower bits of a virtual address, hence disregarding the upper address bits that distinguish between the XKSSEG memory region (where TCBs are mapped to by kern_virt) and the XKUSEG memory region (where user-level virtual pages reside).

Accounting for overall-performance discrepancies

Our kern_phys_* kernels spent 2.65 and 12.9 seconds less time servicing TLB misses than the kern_virt kernel. It is immediately apparent that these savings completely account for the benchmark-duration timing discrepancies in Tables 5.1 and 5.4. Nevertheless this does not concretely prove the assertion that a physically-addressed kernel's reduced TLB footprint is the only trade-off associated with physically addressing TCBs that has a non-negligible impact on the performance of an L4-based system. Nevertheless we strive to provide a reasonable argument for the validity of this assertion in the following. We ask the reader to focus primarily on the kernel-compilation benchmark results because although the disparity in AIM7 benchmark timing results coincides with TLB-miss-penalty savings, it is sufficiently small to fall within our 0.43% statistical error margin.

It is not hard to see that the algorithmic cost of translating thread numbers to TCB addresses in all our kernels has only a negligible impact on overall system performance. For examining the rate of IPC activity shows that a fluctuation of 36.4 (AIM7) or 390 (GCC) cycles in average IPC performance is required to induce a 1.0% change in timing results. Table 4.4 shows, however, that the algorithmic cost of translating thread numbers differs by no more than 4 cycles amongst the five kernels benchmarked.

Amongst the performance trade-offs associated with physically addressing TCBs (see Section 3.6.2), data-cache miss penalties attributed to a physically-addressed kernel's potential increase in data-cache footprint is the only remaining trade-off likely to have a non-negligible impact on the performance of an L4-based system. It is not difficult to show this is not the case for the kernel-compilation benchmark. In the worst case each threadID-table lookup performed by a physically-addressed kernel (during the GCC benchmark) results in an immediate data-cache miss for indexing the table and then up to two further (indirect) data-cache misses for restoring the contents of the cache set touched by that threadID-table lookup (the R4700's primary data cache is only two-way set associative). But even when each threadID-table lookup induces 3 cache misses each costing say 20 cycles, the overall performance penalty cannot sway the kernel-compilation duration results by more than $\frac{1.87}{0.967} \times \frac{3 \times 20}{100} = 1.16$ seconds which is well within our 0.45% statistical error margin. In fact given that the average TLB refill incurs a cache miss when indexing the STLB, and given that the kernel-compilation

benchmark enjoyed a TLB-miss rate 79 times higher than the rate of IPC activity, it is clear that the `kern_virt` kernel's penalty in polluting memory caches by servicing TLB misses that would not have been raised if TCBs were addressed physically is much more severe than the `kern_phys_*` kernels' penalty in polluting the data cache by accessing a table to locate TCB objects.

Further benchmarking

In this section we present a more systematic and universally-applicable technique of determining the overall performance penalty attributed to a physically-addressed kernel's increase in data-cache footprint.

Suppose we timed a second benchmark run with profiling disabled but the threadID table placed in uncached physical memory (`CKSEG1`) and suppose T' represents the result of that timing and that T'' represents the analogous timing measurement when the threadID table is placed in cached physical memory (`CKSEG0`). Further suppose that A represents the number of threadID-table lookups performed during a benchmark run and that M represents the latency (in seconds) of accesses to the `CKSEG1` memory region (measured at 18.2 cycles on average and 16 cycles best case). Then because all the physically-addressed kernels appearing in Table 5.1 touch only one data cache line when performing a threadID-table lookup, the expression $T'' - (T' - A \times M)$ measures the performance penalty a physically-addressed kernel incurs by polluting the system's data cache with threadID-table data.

Unfortunately due to time constraints we were unable to conduct a comprehensive evaluation using the above technique. The limited evaluation we managed to perform using this technique centred around the `kern_phys_4b` kernel and the AIM7 benchmark. The results we scavenged all confirmed our belief that threadID-table footprint in the system's data cache was responsible for a negligible fraction of the overall performance achieved by a physically-addressed kernel (less than 0.4% in this case). Similar conclusions for the other three physically-addressed kernels appearing in Table 5.4 are likely to be proven with this technique given that amongst all four physically-addressed kernels evaluated here, the `kern_phys_4b` kernel is most likely to suffer the greatest increase in data-cache footprint.

5.4.2 Benchmark Set #2

The purpose of the set of benchmarks presented in this section is to evaluate the effectiveness of a threadID-table cache in concealing the cost of traversing a more expensive but resource-friendly threadID-table backing store. Time constraints limited the scope of this evaluation, nevertheless we feel the correct conclusions to be drawn from the evaluation conducted here are immediately apparent.

Results — Simulation of a file-server workload with AIM7

The results in Table 5.7 were obtained by subjecting those physically-addressed kernels featuring a threadID-table cache to the AIM7 file-server-simulation benchmark in a manner identical to that used for the previous set of benchmarks. The results in Table 5.8 were obtained in the same way, but after having reduced the size of all threadID-table caches to contain just one cache set.

Benchmark Results		
Kernel	Completion time	# ThreadID-Table Cache Misses
`phys_cache_1way`	719 seconds	20.0
`phys_cache_2way`	723 seconds	20.0
`phys_cache_4way`	722 seconds	20.0
`phys_merge_cache`	719 seconds	20.0
`phys_merge_store_cache`	720 seconds	20.0

Table 5.7: Benchmark results for threadID-table cache performance when simulating a file-server workload.

Benchmark Results (Near-Worst Case)		
Kernel	Completion time	# ThreadID-Table Cache Misses ($\times 10^6$)
`phys_cache_1way`	730 seconds	19.6
`phys_cache_2way`	722 seconds	0.544
`phys_cache_4way`	724 seconds	0.453
`phys_merge_cache`	733 seconds	19.2
`phys_merge_store_cache`	732 seconds	18.7

Table 5.8: Benchmark results for near-worst-case threadID-table cache performance when simulating a file-server workload.

Analysis of threadID-table cache performance

The five kernels featuring a threadID-table cache performed effectively identically (given the statistical error margin of 0.43%) to the performance level set by the four physically-addressed kernels on display in Table 5.1. This is not at all surprising. During each AIM7-benchmark thread's timeslice, only that thread issues Linux system calls to the Wombat server. Hence within a timeslice only two threadID-table cache entries are accessed — one for the AIM7 thread owning the current timeslice, and one for the Wombat server. The fact that only 20 out of over 21 million threadID-table lookups resulted in a threadID-table cache miss indicates that none of the thread's executing in our AIM7 benchmark runs had conflicting threadID-table cache buckets (which is given for the two-way and four-way set-associative threadID-table caches but not necessarily for the direct-mapped threadID-table caches).

We also benchmarked close-to-worst-case threadID-table cache behaviour by reducing the number of sets in each threadID-table cache to just one. The two-way and four-way set associative threadID-table caches still enjoyed a hit rate in excess of 97.5% (they incurred no more than two cache misses per timeslice) which was expected. The hit rate of the direct-mapped threadID-table caches naturally suffered, reaching a low of 8.4% for the `phys_cache_1way` kernel and resulting in a spike in benchmark-duration timing results. The spike however was only marginally greater than 1% of the benchmark's overall duration.

Due to time constraints we were not able to benchmark threadID-table cache performance with more than two L4 threads engaging in IPC activity during a single timeslice. Hence the results presented in Table 5.7 and Table 5.8 fail to investigate the importance of associativity in threadID-table caches.

5.4.3 Benchmark Set #3

In all the benchmark runs conducted thus far, at most two L4 threads engaged in IPC activity within the window of a timeslice. In this section we examine the behaviour of the `kern_virt` and `kern_phys_4b` kernels when more than two threads concurrently interact via IPC. We simulate this by creating a number of threads subservient to the Wombat server and then modifying the Linux system-call convention so that when the Wombat server receives an exception IPC from an L4 thread, it passes a token once around a ring formed by its subservient threads and itself, and then proceeds to handle the exception IPC normally. Under these conditions the increase in TLB and data-cache footprint incurred by the `kern_virt` and `kern_phys_4b` kernels respectively is accentuated by a factor controlled by the number of subservient threads assigned to the Wombat server. We should also state for the record that the thread numbers assigned to the Wombat server, its subservient threads and to Linux user threads have all been contrived so that no two of these threads' TCB virtual mappings occupy the same TLB subblock when the benchmark is ran on `kern_virt`.

In conducting this evaluation we operate under the assumption that any differences between the `kern_virt` and `kern_phys_4b` kernels' performance is completely accounted for by differences in TLB behaviour. We feel justified in making this assumption since we only execute the GCC kernel-compilation benchmark which features a sufficiently low level of IPC activity that the mild performance side-effects caused by threadID-table cache footprint are invisible in overall benchmark timing results. Furthermore, the abundance of TCB-related data-cache misses during these benchmark runs resulting from the low associativity of the MIPS R4700's data cache (and 4096-byte alignment of TCB structures), serves as a perfect blanket for concealing any threadID-table-induced cache-miss penalties.

Time constraints prevent us from performing an analysis at the same level of detail conducted for Benchmark Set #1 (see Section 5.4.1). Instead for this set of benchmarks we simply summarise the most interesting figures pertaining to TLB performance in table form and let the results speak for themselves.

Results — Compilation of L4Ka::Pistachio source with GCC

Table 5.9 displays results derived from measurements made by running the kernel-compilation benchmark under conditions already described in the preceding paragraphs.

The first column in Table 5.9 represents the number of threads participating in IPC activity during the execution of a Linux system call. It is equal to the number of subservient threads assigned to the Wombat server plus two. It also represents the number of TLB entries (out of 48) typically occupied by virtually-mapped TCB pages for the `kern_virt` kernel when Linux system calls are executed very frequently.

The second column in Table 5.9 provides the difference in TLB-miss rates incurred by the `kern_virt` and `kern_phys_4b` kernels as a percentage of `kern_phys_4b`'s total TLB misses.

The data in the third column is most interesting. It is a measure of the TLB size a `kern_phys_4b` kernel needs to execute on to incur as many TLB-miss exceptions as a `kern_virt` kernel executing on a full 48-entry TLB. Hence it succinctly summarises the performance penalty attached to TCB-page footprint in the TLB for a kernel that addresses TCBs virtually.

We calculated the figures in the third column by running the GCC benchmark on the `kern_phys` kernel but with the MIPS R4700's TLB truncated to anywhere between 44 and 48 in size. The TLB truncation was achieved by inserting invalid virtual mappings into the first N entries of the processor's TLB and then using the `CP0_WIRED` co-processor register to ensure these invalid mappings were never replaced by the TLB-refill handler. The 5 TLB-miss figures obtained by running the `kern_phys` kernel with a truncated TLB of $44-48$ entries were then plotted on a graph. By intersecting the `kern_virt`'s TLB-miss rate with a smooth curve interpolating the points on that graph, the figures in the third column of Table 5.9 were acquired.

TLB performance for `kern_virt`		
# Concurrent threads	% Additional TLB misses	Effective TLB size
2	10.7 %	47.0
3	16.1 %	46.4
4	22.4 %	45.8
6	35.8 %	44.4

Table 5.9: TLB performance for the `kern_virt` kernel when compiling a kernel with GCC.

5.5 Conclusions & Discussion

In the following we summarise and discuss conclusions drawn from the work conducted in Section 5.4. We believe that the two workloads used for evaluating our kernels were sufficiently well behaved for the conclusions presented here to hold when executing a wider range of applications on the Iguana/Wombat framework.

To begin with we address the impact each of the key performance trade-offs listed in Section 3.6.2 had on overall system performance.

- The algorithmic cost of translating thread numbers to TCB addresses had no bearing on overall system performance. In fact amongst those physically-addressed kernels not featuring a non-direct-mapped threadID-table cache, the algorithmic cost of translating thread numbers was within 10 cycles of the analogous figure for `kern_virt`. Neither of our selected benchmarks exhibited a level of IPC activity required for an additional 10 cycles on the IPC path to have a noticeable impact on system performance. In fact it is hard to imagine any non-trivial real application workload enjoying a sufficiently high rate of IPC invocations on the modestly-clocked 100MHz U4600 boxes for an additional 10-cycle overhead in IPC handling to become non-negligible.

- Any increase in data-cache footprint enjoyed by our physically-addressed kernels had negligible impact on system performance. In contrast TLB entries required for TCB objects in a non-physically-addressed kernel sufficiently increased TLB-miss rates to have some notable influence on system performance when the processor's TLB working set was large. But even then the impact was marginal — compiling a kernel with GCC on a physically-addressed kernel improved system performance only by 2.0% .

  That excess TLB footprint inherent in addressing TCBs virtually has a significantly more profound effect on system performance than a physically-addressed kernel's (possible) increase in data-cache footprint can readily be rationalised. For one, given that threadID-table caches enjoy an almost 100% hit rate, no threadID-table lookup performed by a physically-addressed kernel should touch more than one data cache line. Hence a round-trip call IPC executed by a physically-addressed kernel requires (at most) two additional data cache lines which constitutes only 0.39% of the R4700's data-cache capacity. On the other hand a kernel that treats TCBs as virtual objects requires two TLB entries for TCB mappings — constituting 4.16% of the R4700's TLB capacity — to execute a round-trip call IPC. On top of this, the typical penalty attached to a data-cache miss is merely 18 cycles but the typical penalty attached to TLB misses is in excess of 53 cycles on the MIPS R4700.

We now turn to addressing the significance of those performance-influential factors described in Section 5.1.

- A notion explored when discussing threadID-table design alternatives was to spend a few extra cycles on the IPC path in hope of reducing any performance penalties a physically-addressed kernel may incur from a possible increase in data-cache footprint. Specifically we implemented and evaluated kernels that spent up to an extra 9 cycles on the IPC fastpath in hope of reducing data-cache usage. It turned out however that neither the additional (up to) 9-cycle IPC overhead in these kernels or the overhead attributed to the `kern_phys_4b` kernel's excess data-cache footprint had a non-negligible impact on overall system performance.

  It is worthwhile noticing, however, that the analysis of AIM7 benchmark results for Benchmark Set #1 (in Section 5.4.1) revealed the `kern_merge_*` kernels were effective in reducing total data-cache miss penalties by merging TCB fields into their threadID tables. In fact the `kern_merge_store` kernel's IPC fastpath outperformed the `kern_virt`'s IPC fastpath (but not in a manner that visibly affected overall system performance). This kernel implementation solidly proves that TCBs can be addressed physically on the MIPS R4700 without increasing the microkernel's consumption of system memory caches.

We cannot draw similar conclusions for the effectiveness of address compression of threadID-table entries in reducing IPC-fastpath performance. This is because in all our benchmark runs, thread numbers were assigned to threads in a manner that ensured no Linux user thread's threadID-table entry fell on the same data cache line occupied by the Wombat server's threadID-table entry (this design choice was made to exacerbate a physically-addressed kernel's potential increase in data-cache footprint in hope that penalties attached to this increase would be more inclined to show up in benchmark timing results).

- Our limited evaluation of threadID-table cache performance was consistent with expectations of high hit rates that completely conceal the cost of traversing more expensive but resource-friendly backing stores — at least when no more than two threads engaged in IPC activity during a single timeslice. This is not surprising since caching two threadID-table cache entries per timeslice is not a challenging task. Even when these two threadID-table cache entries consistently conflicted (for direct-mapped threadID-table caches), the overall performance degradation was not much more than 1%.

  Unfortunately we did not investigate threadID-table cache performance when more than two L4 threads interacted via IPC within the window of a timeslice. We confidently assert however that a physically-addressed kernel featuring a threadID-table cache will not perform significantly worse than `kern_virt` when many threads concurrently participate in IPC. In fact if in any L4-based system running on the MIPS R4700 processor that features a sufficiently large number of threads engaging in IPC during a single timeslice that caching threadID-table entries proves a challenge, then the resulting abundance of TCB-related data-cache misses will certainly conceal any penalties attributed to threadID-table cache trashing.

- Our evaluation was conducted purely within the context of the Iguana/Wombat framework. Hence we did not thoroughly investigate the influence of user-level operating-system design on the performance impact associated with physically addressing TCBs. Ideally we would liked to have benchmarked our physically-addressed kernels running an operating system promoting greater thread interaction in terms of number of threads concurrently engaging in IPC. A multi-server operating-system design might achieve this, as might the presence of Gigabit network devices (or any other hardware device that frequently generates interrupts). Nevertheless the results in Benchmark Set #3 strongly suggest that any environment where a large number of threads are likely to invoke L4 IPC during a single timeslice is likely to accentuate the performance difference between the two categories of kernels under investigation. Of course the accentuation of this difference will be in the favour of physically-addressed kernels when executing on a MIPS R4700 processor.

- We have already seen that the reduced TLB-refill penalties a physically-addressed kernel incurs by not polluting the TLB with TCB mappings is the most significant performance trade-off associated with TCB-addressing methods. Given this it is certain that architectural properties pertaining to TLB performance will influence a physically-addressed kernel's behaviour. For our evaluation, the MIPS R4700's small subblocked TLB accentuated the number of TLB misses the `kern_virt` kernel incurred due to TCB references polluting the TLB. Furthermore the severity of the penalty attached to servicing those TLB misses was increased by the processor's use of a software-managed rather than hardware-managed TLB.

  Unfortunately we cannot quantify the extent to which TLB properties of the MIPS R4700 influenced our physically-addressed kernels' performance since we failed to concretely evaluate those kernels on other architectures. For the same reason we cannot conclude if architectural memory-subsystem properties may result in the presence of threadID-table data in a system's memory caches having a more profound impact on overall system performance than that which occurred for the MIPS R4700 processor. It is the primary weakness of our work that a cross-architectural analysis of a physically-addressed kernel's performance was not conducted.

Chapter 6

Epilogue

In this thesis we modified the L4Ka::Pistachio implementation of the L4 microkernel to address TCB structures physically rather than virtually. With only a minor exception in long IPC, the kernels obtained through this process were completely void of virtual addressing. The primary advantage of such physically-addressed kernels is simplicity that facilitates formal verification of the microkernels' design and implementation. Our overriding objective was to provide a performance evaluation of those kernels. We conducted our evaluation in a platform environment set by the MIPS R4700 processor and focused largely on a physically-addressed kernel's impact on the performance-critical IPC primitive and also on overall system performance when running Linux on top of the L4 microkernel.

The results obtained from our evaluation demonstrated that choosing to physically address TCB structures does not significantly alter the performance of an L4 microkernel. In particular we found that compiling an L4 microkernel using the Wombat Linux server completed 2.0% faster on a physically-addressed kernel than on L4Ka::Pistachio. More generally our results showed that any performance difference between a physically-addressed kernel and stock L4Ka::Pistachio is likely to be in the physically-addressed kernel's favour and then accounted for by a reduction in TLB-refill penalties due to TCB virtual mappings not polluting the TLB when TCBs are treated as physical objects.

The three most important factors affecting the severity of any performance difference between the two categories of kernels benchmarked were found to be i) the size of the processor's TLB working set, ii) the frequency of L4 IPC invocations as well as the number of L4 threads engaging in IPC activity during a single timeslice and iii) architectural properties influencing TLB performance. Any additional data-cache footprint a physically-addressed kernel incurs by requiring an auxiliary lookup table to locate TCB objects had only negligible impact on overall system performance. Hence our exploration of design choices seeking to reduce a physically-addressed kernel's primary potential performance drawback proved largely fruitless.

The most significant weakness of this thesis is the absence of a cross-architectural evaluation of a physically-addressed kernel's performance. Our evaluation was restricted to the MIPS R4700 processor which offers a highly favourable environment for a physically-addressed kernel to perform in. This is because the MIPS R4700 features a small software-managed TLB that accentuates the TLB-refill penalties L4Ka::Pistachio incurs in consuming precious TLB entries with TCB virtual mappings. It also enjoys a forgiving memory-cache miss penalty when clocked at a mere 100Mhz. Given this, as part of any future work conducted in line with the goals of this thesis, we recommend that physically-addressed kernels be evaluated on an architecture where TLB misses are cheap and memory-cache misses expensive when compared to the MIPS R4700. The Intel IA-32 architecture, featuring a TLB that can be refilled in as little as 15 cycles assuming L2 cache hits and featuring memory-cache miss penalties reaching 100 cycles for some processor models, serves as a highly suitable candidate for such an evaluation. Nevertheless, we are confident that even on this architecture the simplicity gained from a completely physically-addressed kernel can be enjoyed without any significant performance degradation.

Appendix A

Thread Control Block Layout

The TCB layout of our physically-addressed modification of L4Ka::Pistachio is provided below.

```
class tcb_t {

    /* CACHE LINE 1 BEGINS HERE */
    threadid_t      myself_global;
    threadid_t      myself_local;
    u32_t           utcb;
    u32_t           space;
    word_t          dummy;
    /* CACHE LINE 1 ENDS HERE */

    /* CACHE LINE 2 BEGINS HERE */
    u16_t           asid;
    cpuid_t         cpu;
    thread_state_t  thread_state;
    threadid_t      partner;
    resource_bits_t resource_bits;
    u32_t           stack;
    u32_t           send_head;
    /* CACHE LINE 2 ENDS HERE */

    /* IPC FASTPATH DOES NOT TOUCH ANYTHING BELOW HERE (OTHER THAN THE KERNEL STACK) */
    word_t          pdir_cache;
    queue_state_t   queue_state;
    ringlist_t      present_list, ready_list, wait_list, send_list;
    spinlock_t      tcb_lock;

    u64_t           total_quantum, timeslice_length, current_timeslice, absolute_timeout;
    prio_t          priority, sensitive_prio;
    u16_t           current_max_delay, max_delay;
    threadid_t      scheduler;

    bitmask_t       flags;
    arch_ktcb_t     arch;
    misc_tcb_t      misc;
    threadid_t      saved_partner;
    thread_state_t  saved_state;
    resources_t     resources;

    word_t          kernel_stack[0];
}
```

Appendix B

The IPC Fastpath

The IPC fastpath of our physically-addressed L4Ka::Pistachio MIPS64 modification is reproduced below. The `tidtable_lookup` macros implement (global) thread identifier to thread-control-block address translations for various implementations of the threadID table.

```
#include INC_ARCH(asm.h)
#include INC_ARCH(regdef.h)
#include INC_GLUE(context.h)
#include INC_GLUE(syscalls.h)
#include <asmsyms.h>
#include <tcb_layout.h>

    .set noat
    .set noreorder

/* SYSCALL ENTRY POINT INTO THE KERNEL */
BEGIN_PROC(__mips64_interrupt_fp)
    mfc0  k1, CP0_CAUSE
    li    k0, 8<<2
    andi  k1, k1, 0x7c
    bne   k0, k1, other_exception
    mfc0  k0, CP0_STATUS
    li    AT, SYSCALL_ipc
    bne   v0, AT, _goto_mips64_l4syscall
    lui   t5, %hi(K_STACK_BOTTOM)

    j     _mips64_fastpath
    move  t7, k0

    _goto_mips64_l4syscall:
    j     _mips64_l4syscall
    nop

    other_exception:
    dsll  k1, k1, 1
    lui   k0, %hi(exception_handlers)
    add   k0, k0, k1
    ld    k0, %lo(exception_handlers)(k0)
    jr    k0
    nop
END_PROC(__mips64_interrupt_fp)
```

```
    #define to_tid          a0
    #define from_tid        a1
    #define timeout         a2
    #define current         a3
    #define to_tcb          t0
    #define from_tcb        t1
    #define to_state        t2
    #define current_global  v0

#if defined(CONFIG_TIDTABLE_MERGE_TCB)
    #define to_tidtcb       k1
    #define from_tidtcb     t5    /* abi : overlap with tmp1, so be careful */
#else
    #define to_tidtcb       to_tcb
    #define from_tidtcb     from_tcb
#endif

#if defined(CONFIG_STORE_CURRENT_TIDTCB)
    #define current_tidtcb  k0
#else
    #define current_tidtcb   current
#endif

    #define tmp0            t4
    #define tmp1            t5
    #define tmp2            t6
    #define tmp3            t7
    #define tmp4            t8
    #define tmp5            t9
    #define tmp6            t3

    #define mr0             v1
    #define mr1             s0
    #define mr2             s1
    #define mr3             s2
    #define mr4             s3
    #define mr5             s4
    #define mr6             s5
    #define mr7             s6
    #define mr8             s7


#ifdef CONFIG_TIDTABLE_MERGE_TCB
#define    OFS_TCBADDR        (28)
#endif

// abi : tidtable_lookup macros : begin -->

#if    defined(CONFIG_PHYSICAL_KERNEL)

#if    !defined(CONFIG_TIDTABLE_MERGE_TCB)

#if    defined(CONFIG_TIDTABLE_ADDR_8BYTES)
    #define tidtable_lookup(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC) \
        dsrl    tmpA,  tid_r, 32;                               \
        lui     tmpB,  %hi(ASM_START_ADDRESS_TIDTABLE);     \
        dsll    tmpA,  3;                                       \
        daddu   tmpB,  tmpA;                                    \
        ld      tcb_r, %lo(ASM_START_ADDRESS_TIDTABLE)(tmpB);
```

```
#elif  defined(CONFIG_TIDTABLE_ADDR_4BYTES)

#if    !defined(CONFIG_TIDTABLE_MPT_CACHED)
    #define tidtable_lookup(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC) \
        dsrl    tmpA,  tid_r, 32;                           \
        lui     tmpB,  %hi(ASM_START_ADDRESS_TIDTABLE);     \
        dsll    tmpA,  2;                                   \
        daddu   tmpB,  tmpA;                                \
        lw      tcb_r, %lo(ASM_START_ADDRESS_TIDTABLE)(tmpB);

#else  // CONFIG_TIDTABLE_MPT_CACHED

#define ASM_TIDTABLE_CACHE_LOG2BUCKETS (ASM_TIDTABLE_CACHE_LOG2SIZE - ASM_TIDTABLE_CACHE_LOG2WAYS)

#if    (ASM_TIDTABLE_CACHE_LOG2WAYS == 0)
    #define __tidtable_lookup(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC, refill, resume)    \
        dsll    tmpB,  tid_r, (32 - ASM_TIDTABLE_CACHE_LOG2BUCKETS); \
        lui     tmpA,  %hi(ASM_START_ADDRESS_TIDTABLE);             \
        dsrl    tmpB,  tmpB, (64 - ASM_TIDTABLE_CACHE_LOG2BUCKETS);  \
        dsll    tmpB,  3 + ASM_TIDTABLE_CACHE_LOG2WAYS;             \
        daddu   tmpA,  tmpB;                                        \
        lwu     tmpB,  %lo(ASM_START_ADDRESS_TIDTABLE + 0)(tmpA);    \
        dsrl    tmpC,  tid_r, 32;                                   \
        bne     tmpB,  tmpC, refill;                                \
        lw      tcb_r, %lo(ASM_START_ADDRESS_TIDTABLE + 4)(tmpA);    \
    resume:

#elif    (ASM_TIDTABLE_CACHE_LOG2WAYS == 1)
    #define __tidtable_lookup(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC, refill, resume)    \
        dsll    tmpB,  tid_r, (32 - ASM_TIDTABLE_CACHE_LOG2BUCKETS); \
        lui     tmpA,  %hi(ASM_START_ADDRESS_TIDTABLE);             \
        dsrl    tmpB,  tmpB, (64 - ASM_TIDTABLE_CACHE_LOG2BUCKETS);  \
        dsll    tmpB,  3 + ASM_TIDTABLE_CACHE_LOG2WAYS;             \
        daddu   tmpA,  tmpB;                                        \
                                                                    \
        lwu     tmpB,  %lo(ASM_START_ADDRESS_TIDTABLE + 0)(tmpA);    \
        dsrl    tmpC,  tid_r, 32;                                   \
        beq     tmpB,  tmpC, resume;                                \
        lw      tcb_r, %lo(ASM_START_ADDRESS_TIDTABLE + 4)(tmpA);    \
                                                                    \
        lwu     tmpB,  %lo(ASM_START_ADDRESS_TIDTABLE + 8)(tmpA);    \
        nop;                                                        \
        bne     tmpB,  tmpC, refill;                                \
        lw      tcb_r, %lo(ASM_START_ADDRESS_TIDTABLE + 12)(tmpA);   \
    resume:
```

```
#elif    (ASM_TIDTABLE_CACHE_LOG2WAYS == 2)
    #define __tidtable_lookup(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC, refill, resume) \
        dsll    tmpB,  tid_r, (32 - ASM_TIDTABLE_CACHE_LOG2BUCKETS); \
        lui     tmpA,  %hi(ASM_START_ADDRESS_TIDTABLE);              \
        dsrl    tmpB,  tmpB, (64 - ASM_TIDTABLE_CACHE_LOG2BUCKETS);  \
        dsll    tmpB,  3 + ASM_TIDTABLE_CACHE_LOG2WAYS;              \
        daddu   tmpA,  tmpB;                                         \
                                                                     \
        lwu     tmpB,  %lo(ASM_START_ADDRESS_TIDTABLE + 0)(tmpA);    \
        dsrl    tmpC,  tid_r, 32;                                    \
        beq     tmpB,  tmpC, resume;                                 \
        lw      tcb_r, %lo(ASM_START_ADDRESS_TIDTABLE + 4)(tmpA);    \
                                                                     \
        lwu     tmpB,  %lo(ASM_START_ADDRESS_TIDTABLE + 8)(tmpA);    \
        lw      tcb_r, %lo(ASM_START_ADDRESS_TIDTABLE + 12)(tmpA);   \
        beq     tmpB,  tmpC, resume;                                 \
                                                                     \
        lwu     tmpB,  %lo(ASM_START_ADDRESS_TIDTABLE + 16)(tmpA);   \
        lw      tcb_r, %lo(ASM_START_ADDRESS_TIDTABLE + 20)(tmpA);   \
        beq     tmpB,  tmpC, resume;                                 \
                                                                     \
        lwu     tmpB,  %lo(ASM_START_ADDRESS_TIDTABLE + 24)(tmpA);   \
        nop;                                                         \
        bne     tmpB,  tmpC, refill;                                 \
        lw      tcb_r, %lo(ASM_START_ADDRESS_TIDTABLE + 28)(tmpA);   \
    resume:
#endif

    #define tidtable_lookup(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC)  \
        __tidtable_lookup(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC,    \
            tidtable_cache_miss, continue_after_lookup)
    #define tidtable_lookup2(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC) \
        __tidtable_lookup(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC,    \
            tidtable_cache_miss2, continue_after_lookup2)
#endif    // CONFIG_TIDTABLE_MPT_CACHED

#elif    defined(CONFIG_TIDTABLE_ADDR_2BYTES)
    #define tidtable_lookup(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC) \
        dsrl    tmpA,  tid_r, 32;                                    \
        lui     tmpB,  %hi(ASM_START_ADDRESS_TIDTABLE);              \
        dsll    tmpA,  1;                                            \
        daddu   tmpB,  tmpA;                                         \
        lui     tcb_r, %hi(ASM_START_ADDRESS_DUMMYTCB);              \
        lhu     tmpA,  %lo(ASM_START_ADDRESS_TIDTABLE)(tmpB);        \
        addiu   tcb_r, %lo(ASM_START_ADDRESS_DUMMYTCB);              \
        dsll    tmpA,  KTCB_BITS;                                    \
        add     tcb_r, tmpA;

#endif    // CONFIG_TIDTABLE_ADDR_*
```

```
#else    // CONFIG_TIDTABLE_MERGE_TCB
#if    !defined(CONFIG_TIDTABLE_MPT_CACHED)
    #define tidtable_lookup(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC) \
        dsrl    tmpA, tid_r, 32;                                      \
        lui     tidtcb_r, %hi(ASM_START_ADDRESS_TIDTABLE);            \
        dsll    tmpA, 5;                                              \
        daddu   tidtcb_r, tmpA;                                       \
        lw      tcb_r, %lo(ASM_START_ADDRESS_TIDTABLE + OFS_TCBADDR)(tidtcb_r); \
        daddiu  tidtcb_r, %lo(ASM_START_ADDRESS_TIDTABLE);
#else    // CONFIG_TIDTABLE_MPT_CACHED
    #define __tidtable_lookup(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC, refill, resume) \
        dsll    tmpA, tid_r, (32 - ASM_TIDTABLE_CACHE_LOG2SIZE);  \
        lui     tidtcb_r, %hi(ASM_START_ADDRESS_TIDTABLE);        \
        dsrl    tmpA, tmpA, (64 - ASM_TIDTABLE_CACHE_LOG2SIZE);   \
        dsll    tmpA, 5;                                          \
        daddu   tidtcb_r, tmpA;                                   \
        ld      tmpB, %lo(ASM_START_ADDRESS_TIDTABLE)(tidtcb_r);  \
        daddiu  tidtcb_r, %lo(ASM_START_ADDRESS_TIDTABLE);        \
        bne     tmpB, tid_r, refill;                              \
        lw      tcb_r, OFS_TCBADDR(tidtcb_r);                     \
    resume:

    #define tidtable_lookup(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC)     \
        __tidtable_lookup(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC,       \
                tidtable_cache_miss, continue_after_lookup)
    #define tidtable_lookup2(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC)    \
        __tidtable_lookup(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC,       \
                tidtable_cache_miss2, continue_after_lookup2)

#endif    // CONFIG_TIDTABLE_MPT_CACHED
#endif    // CONFIG_TIDTABLE_MERGE_TCB

#else    /* !CONFIG_PHYSICAL_KERNEL */
    #define tidtable_lookup(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC)        \
        dsrl   tmpA, tid_r, 32;                   \
        li     tcb_r, 0x4;                        \
        dsll   tmpA, KTCB_BITS;                   \
        dsll   tcb_r, 60;                         \
        or     tcb_r, tmpA, tcb_r;
#endif

/* abi : tidtable_lookup2 is the same as tidtable_lookup, but it branches to
 * abi : tidtable_cache_miss2 instead of tidtable_cache_miss on a tidtable_cache_t miss.
 * abi : It is used only when looking up the from_tid.
 */
#ifndef tidtable_lookup2
#define tidtable_lookup2 tidtable_lookup
#endif

// abi : tidtable_lookup macros : end <---
```

```
// abi : t5 must have lui bits set to %hi(K_STACK_BOTTOM)
// abi : t7 must contain CP0_STATUS
        .set reorder
        .set noat
    BEGIN_PROC(_mips64_fastpath)
    srl   t6, t7, 5
    move  t4, sp
    sll   t6, t6, 5
    mtc0  t6, CP0_STATUS

    ld    sp, %lo(K_STACK_BOTTOM)(t5)

#ifdef CONFIG_STORE_CURRENT_TIDTCB
    ld    current_tidtcb, %lo(K_CURRENT_TIDTCB)(t5)
#endif

    dmfc0 t6, CP0_EPC

    sd    ra, PT_RA-PT_SIZE(sp)
    sd    t7, PT_STATUS-PT_SIZE(sp)
    sd    t4, PT_SP-PT_SIZE(sp)

    daddu t6, t6, 4
    dsubu sp, sp, PT_SIZE

    dli   tmp4, -KTCB_SIZE
    sd    t6, PT_EPC(sp)

/*** START FASTPATH ***/
send_path:
    .set    noreorder
    /* Look for a nil from-tid / to-tid. */
    beqz   to_tid, _mips64_slowpath            /
    and    current, sp, tmp4    /* get current tcb */

    /* Check that the receive timeout is infinite */
    andi   tmp2, timeout, 0xffff
    bnez   tmp2, _mips64_slowpath

    tidtable_lookup(to_tid, to_tcb, to_tidtcb, tmp0, tmp2, tmp5)

    /* We don't do typed words or propagation.*/
    andi   tmp0, mr0, 0xffc0
    bnez   tmp0, _mips64_slowpath

/* abi : check current_tidtcb cache entry hasn't been trashed */
/* abi : we make use of the load/branch delay slots so this only costs 2 cycles */
#if    defined(CONFIG_STORE_CURRENT_TIDTCB) && defined(CONFIG_TIDTABLE_MPT_CACHED)
    lw     tmp5, OFS_TCBADDR(current_tidtcb)
    ld     tmp3, OFS_TCB_RESOURCE_BITS(to_tcb)
    bne    tmp5, current, resolve_tidtcb_clash
    ld     tmp0, OFS_TCB_RESOURCE_BITS(current)
continue_after_clash:
#else
    /* Check if any resource bits are set */
    ld     tmp3, OFS_TCB_RESOURCE_BITS(to_tcb)
    ld     tmp0, OFS_TCB_RESOURCE_BITS(current)
#endif


    bnez   tmp3, _mips64_slowpath
    ld     tmp1, OFS_TCB_MYSELF_GLOBAL(to_tidtcb)

    bnez   tmp0, _mips64_slowpath
    lw     to_state, OFS_TCB_THREAD_STATE(to_tcb)

    /* Check to_tcb->get_global_id == to_tid */
    bne    tmp1, to_tid, _mips64_slowpath
```

```
    /* Check partner is waiting */

    dli    tmp3, -1
    ld     tmp6, OFS_TCB_PARTNER(to_tcb)

    bne    to_state, tmp3, _mips64_slowpath
    ld     current_global, OFS_TCB_MYSELF_GLOBAL(current_tidtcb)

    /* (tcb->get_partner() == current->get_global_id()) || tcb->get_partner().is_anythread() */
    /* is_anythread() */

    beq    tmp3, tmp6, 1f
    lui    ra, %hi(ipc_finish)               /

    /* tcb->get_partner() == current->get_global_id() */
    bne    current_global, tmp6, _mips64_slowpath

1:
    lw     tmp4, OFS_TCB_SPACE(to_tidtcb)

    /* abi : tmp5 = number of untyped items */
    andi   tmp5, mr0, 0x3f
    beqz   tmp4, _mips64_slowpath     /* Null space = interrupt thread */
    sub    tmp5, 8

    /* Check that receive phase blocks */
    beq    to_tid, from_tid, continue_ipc
    daddiu ra, %lo(ipc_finish)

    lw     tmp6, OFS_TCB_SEND_HEAD(current)
    nop
    bnez   tmp6, _mips64_slowpath
    nop
    bne    tmp3, from_tid, check_other_tcb
    nop

continue_ipc:
    sw     tmp3, OFS_TCB_THREAD_STATE(current)

    /* This is the point of no return --- after this we cannot go to the slow path */

    sd     from_tid, OFS_TCB_PARTNER(current)
    blez   tmp5, switch_to
    sd     s8, PT_S8(sp)

    lw     tmp0, OFS_TCB_UTCB(current_tidtcb)
    sll    tmp2, tmp5, 3
    lw     tmp1, OFS_TCB_UTCB(to_tidtcb)

    andi   tmp6, tmp5, 1
    daddu  tmp2, tmp2, tmp0

    beqz   tmp6, 10f
    ld     tmp3, 200(tmp0)
    daddiu tmp0, 8
    daddiu tmp1, 8
    beq    tmp0, tmp2, switch_to
    sd     tmp3, 192(tmp1)
copy_loop:
    ld     tmp3, 200(tmp0)
10:
    ld     tmp6, 208(tmp0)
    daddiu tmp1, 16
    daddiu tmp0, 16
    sd     tmp3, 184(tmp1)
    bne    tmp0, tmp2, copy_loop
    sd    tmp6, 192(tmp1)
```

```
switch_to:
/* mips switch_to */
    /* At this point, we have set up the sending thread's TCB state.  We now setup the
     * stack so that when we are next switched to we do the right thing (set state to running
     * and return partner) --- this only happens in the generic send case.
     */
    dsubu  sp, sp,    MIPS64_SWITCH_STACK_SIZE

    lh     tmp1, OFS_TCB_ASID(to_tcb)    /* get: space->asid (assume no asid management) */

    sd     ra, 32(sp)

    lui    tmp5, %hi(K_STACK_BOTTOM)    /* Load kernel stack base address */

#ifdef CONFIG_PHYSICAL_KERNEL
    sw     sp, OFS_TCB_STACK(current)    /* Store current stack in old_stack */
#else
    sd     sp, OFS_TCB_STACK(current)    /* Store current stack in old_stack */
#endif

    .set at
    dmtc0  tmp1, CP0_ENTRYHI        /* Set new ASID */
    daddiu sp, to_tcb, KTCB_SIZE    /* STACK TOP CALC */
    dsll   tmp4, tmp4, 32
    sd     sp, %lo(K_STACK_BOTTOM)(tmp5)           /* Set current TCB */
#ifdef CONFIG_STORE_CURRENT_TIDTCB
    sd     to_tidtcb, %lo(K_CURRENT_TIDTCB)(tmp5)  /* store tidtcb pointer */
#endif
    dli    tmp0, TSTATE_RUNNING

    dmtc0  tmp4, CP0_CONTEXT    /* Save current Page Table */

    /* Mark self as runnable */
    sw     tmp0, OFS_TCB_THREAD_STATE(to_tcb)

    /* Set return value to sender's global ID (already in v0)*/

    mfc0   t6, CP0_STATUS
    ld     t7, PT_SP-PT_SIZE(sp)               /* load stack */
    ori    t6, t6, ST_EXL                      /* set Exception Level */
    ld     t0, OFS_TCB_MYSELF_LOCAL(to_tidtcb)  /* Load UTCB */

    /* Clean up mr0 (clear receive flags) */
    and    mr0, ~(0xe << 12)

    mtc0   t6, CP0_STATUS            /* to disable interrupts, we now can set EPC */
    ld     t4, PT_STATUS-PT_SIZE(sp) /* load status */
    ld     t5, PT_EPC-PT_SIZE(sp)    /* load epc */
    ld     ra, PT_RA-PT_SIZE(sp)     /* load ra */

    .set    reorder
    dmtc0  t5, CP0_EPC               /* restore EPC */
    ld     s8, PT_S8-PT_SIZE(sp)     /* restore s8 */

    dli    t3, CONFIG_MIPS64_STATUS_MASK
    move   sp, t7            /* restore stack */
    and    t6, t3, t6        /* compute new status register */
    nor    t3, zero, t3
    and    t4, t3, t4
    or     t7, t6, t4
    mtc0   t7, CP0_STATUS    /* new status value */
    move   k0, t0            /* Load UTCB into k0 */
    nop

    eret
```

```
    .set    reorder
ipc_finish:
    dli     tmp0, -KTCB_SIZE         /* tcb mask */
    dli     tmp1, TSTATE_RUNNING
    and     current, sp, tmp0        /* t5 = current tcb */

    daddu   sp, current, KTCB_SIZE-PT_SIZE

    sw      tmp1, OFS_TCB_THREAD_STATE(current)

    ld      v0, OFS_TCB_PARTNER(current)

    j       _mips64_l4sysipc_return

check_other_tcb:
    .set    noreorder
    beqz    from_tid, _mips64_slowpath

    tidtable_lookup2(from_tid, from_tcb, from_tidtcb, tmp0, tmp2, tmp6)

    /* Check global ID */
    ld      tmp0, OFS_TCB_MYSELF_GLOBAL(from_tidtcb)
    lw      tmp1, OFS_TCB_THREAD_STATE(from_tcb)
    /* abi : from_tidtcb is trashed now */
    bne     tmp0, from_tid, _mips64_slowpath

    /* Check if the thread is polling us --- if so, go to slow path */

    /* is_polling() */
    li      tmp2, TSTATE_POLLING
    bne     tmp1, tmp2, continue_ipc    /* from_tcb isn't polling */

    /* partner == current->global_id */
    ld      tmp1, OFS_TCB_PARTNER(from_tcb)
    beq     tmp1, current_global, _mips64_slowpath

    /* partner == current->local_id */
    ld      tmp2, OFS_TCB_MYSELF_LOCAL(current_tidtcb)
    bne     tmp1, tmp2, continue_ipc
    nop

    j       _mips64_slowpath
    nop
```

```
#if    defined(CONFIG_TIDTABLE_MPT_CACHED)

#ifndef CONFIG_TIDTABLE_MERGE_TCB

#define ASM_TIDTABLE_CACHE_WAYS        (1 << ASM_TIDTABLE_CACHE_LOG2WAYS)


#if    (ASM_TIDTABLE_CACHE_LOG2WAYS == 0)
#define    load_tcb_and_pick_bucket(_tcb_r, _mpt, _bucket, _tmp)                    \
    lw    _tcb_r, 0(_mpt);
#else
/* abi : non-direct-mapped cache picks a bucket randomly (using CP0_COUNT register) */
#define    load_tcb_and_pick_bucket(_tcb_r, _mpt, _bucket, _tmp)                    \
    mfc0   _tmp, CP0_COUNT;                             \
    lw     _tcb_r, 0(_mpt);                             \
    andi   _tmp, ASM_TIDTABLE_CACHE_WAYS - 1;           \
    dsll   _tmp, 3;                                     \
    daddu  _bucket, _bucket, _tmp;
#endif

#define refill_tidtable_cache(tid_r, tcb_r, bucket, tmpB, tmpC, refill, resume)         \
refill:                                                 \
    dla     tmpB, ASM_START_ADDRESS_TIDTABLE + (1 << (ASM_TIDTABLE_CACHE_LOG2SIZE + 3)); \
    /* tmpB now contains root directory */                      \
    dsrl32  tmpC, tid_r, 20;                                    \
    andi    tmpC, tmpC, 1023;                                   \
    dsll    tmpC, tmpC, 2;                                      \
    daddu   tmpB, tmpB, tmpC;                                   \
    lw      tmpB, 0(tmpB);                                      \
    /* tmpB now contains 2nd level pointer */                   \
    dsrl32  tmpC, tid_r, 10;                                    \
    andi    tmpC, tmpC, 1023;                                   \
    dsll    tmpC, tmpC, 2;                                      \
    daddu   tmpB, tmpB, tmpC;                                   \
    lw      tmpB, 0(tmpB);                                      \
    /* tmpB now contains 3rd level pointer */                   \
    dsrl32  tmpC, tid_r, 0;                                     \
    andi    tmpC, tmpC, 1023;                                   \
    dsll    tmpC, tmpC, 2;                                      \
    daddu   tmpB, tmpB, tmpC;                                   \
    /* tmpB now contains pointer to tidtable entry */           \
    load_tcb_and_pick_bucket(tcb_r, tmpB, bucket, tmpC);        \
    dsrl    tmpC, tid_r, 32;                                    \
    sw      tcb_r, %lo(ASM_START_ADDRESS_TIDTABLE + 4)(bucket);  \
    b       resume;                                             \
    sw      tmpC, %lo(ASM_START_ADDRESS_TIDTABLE + 0)(bucket);   \


/* abi : arguments need to match tidtable_lookup() arguments */
refill_tidtable_cache(to_tid, to_tcb, tmp0, tmp2, tmp5,     \
            tidtable_cache_miss, continue_after_lookup)
refill_tidtable_cache(from_tid, from_tcb, tmp0, tmp2, tmp6, \
            tidtable_cache_miss2, continue_after_lookup2)
```

```
#else    // CONFIG_TIDTABLE_MERGE_TCB

#define refill_tidtable_cache(tid_r, tcb_r, tidtcb_r, tmpA, tmpB, tmpC, refill, resume, clash) \
refill:                                                                 \
    dla     tmpB, ASM_START_ADDRESS_TIDTABLE + (1 << (ASM_TIDTABLE_CACHE_LOG2SIZE + 5));    \
    /* tmpB now contains root directory */                              \
    dsrl32  tmpC, tid_r, 17;                                            \
    andi    tmpC, tmpC, 1023;                                           \
    dsll    tmpC, tmpC, 2;                                              \
    daddu   tmpB, tmpB, tmpC;                                           \
    lw      tmpB, 0(tmpB);                                              \
    /* tmpB now contains 2nd level pointer */                          \
    dsrl32  tmpC, tid_r, 7;                                             \
    andi    tmpC, tmpC, 1023;                                           \
    dsll    tmpC, tmpC, 2;                                              \
    daddu   tmpB, tmpB, tmpC;                                           \
    lw      tmpB, 0(tmpB);                                              \
    /* tmpB now contains 3rd level pointer */                          \
    dsrl32  tmpC, tid_r, 0;                                             \
    andi    tmpC, tmpC, 127;                                            \
    dsll    tmpC, tmpC, 5;                                              \
    daddu   tmpB, tmpB, tmpC;                                           \
    /* tmpB now contains pointer to tidtable entry */                  \
    clash;                                                              \
    lw      tcb_r, 28(tmpB);                                            \
    ld      tmpA, 8(tmpB);                                              \
    ld      tmpC, 16(tmpB);                                             \
    sd      tmpA, 8(tidtcb_r);                                          \
    sd      tmpC, 16(tidtcb_r);                                         \
    ld      tmpA, 24(tmpB);                                             \
    sd      tid_r, 0(tidtcb_r);                                         \
    b       resume;                                                     \
    sd      tmpA, 24(tidtcb_r);                                         \


refill_tidtable_cache(to_tid, to_tcb, to_tidtcb, tmp0, tmp2, tmp5,       \
            tidtable_cache_miss, continue_after_lookup, )

#define check_clash     beq to_tidtcb, from_tidtcb, to_from_clash
refill_tidtable_cache(from_tid, from_tcb, from_tidtcb, tmp0, tmp2, tmp6, \
            tidtable_cache_miss2, continue_after_lookup2, check_clash)

/* abi : if to_tidtcb and from_tidtcb point to the same bucket, we have a problem :) .
 * abi : we resolve this by not inserting from_* into the cache and making from_tidtcb
 * abi : point to the tidtable entry (rather than the cache entry).
 */

/* abi : tmp2 needs to match with tmpB parameter to refill_tidtable_cache */
to_from_clash:
    b    continue_after_lookup2
    move    from_tidtcb, tmp2

#endif

#endif


#if    defined(CONFIG_STORE_CURRENT_TIDTCB) && defined(CONFIG_TIDTABLE_MPT_CACHED)
resolve_tidtcb_clash:
    move    current_tidtcb, current
    b    continue_after_clash
    nop
#endif

END_PROC(_mips64_fastpath)
```

```
    .set reorder
BEGIN_PROC(_mips64_slowpath)
    lui    ra, %hi(_mips64_l4sysipc_return)
    lw     t5, OFS_TCB_UTCB(current)
    daddiu ra, %lo(_mips64_l4sysipc_return)
    sd     s8, PT_S8(sp)
    sd     mr0, 128(t5)
    sd     mr1, 136(t5)
    sd     mr2, 144(t5)
    sd     mr3, 152(t5)
    sd     mr4, 160(t5)
    sd     mr5, 168(t5)
    sd     mr6, 176(t5)
    sd     mr7, 184(t5)
    sd     mr8, 192(t5)
    j      sys_ipc      /* C++ implementation of the IPC syscall (slowpath) */
    nop
END_PROC(_mips64_slowpath)
```

Bibliography

[1] Aim benchmarks. `http://sourceforge.net/projects/aimbench`.

[2] Allan Bricker, Michel Gien, Marc Guillemont, Jim Lipkis, Douglas Orr, and Marc Rozier. A new look at microkernel-based UNIX operating systems. Technical report, Chorus systemes, Paris, France, 1991.

[3] 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.

[4] Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 translation buffer: Simulation and measurement. *ACM Transactions on Computer Systems*, 3:31–62, 1985.

[5] Michael Condict, Don Bolinger, Dave Mitchell, and Eamonn McManus. Microkernel modularity with integrated kernel performance. Technical report, OSF Research Institute, Cambridge, 1994.

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

[7] Kevin Elphinstone. *Virtual Memory in a 64-bit Microkernel*. PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, March 1999. Available from publications page at `http://www.disy.cse.unsw.edu.au/`.

[8] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. Page tables for 64-bit computer systems. Technical Report UNSW-CSE-TR-9804, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, August 1998.

[9] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. *L4 Reference Manual: MIPS R4x00, Version 1.11, Kernel Version 79*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, May 1999. Available from `http://www.disy.cse.unsw.edu.au/Software/L4`.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Joseph Heinrich. *MIPS R4000 User's Manual*. Prentice Hall, 1993.

[15] Gernot Heiser. *Inside L4/MIPS: Anatomy of a High-Performance Microkernel*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, January 2001. Available from `http://www.disy.cse.unsw.edu.au/Software/L4`.

[16] 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, April 1992.

[17] Integrated Device Technology. *IDT79R4600 and IDT79R4700 RISC Processor Hardware User's Manual*, April 1995.

[18] Intel Corp. *IA-32 Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2001. URL `ftp://download.intel.com/design/Pentium4/manuals/245472.htm`.

[19] Intel Corp. *IA-32 Architecture Software Developer's Manual Volume 1: Basic Architecture*, 2002. URL `http://developer.intel.com/design/pentium4/manuals/245470.htm`.

[20] Intel Corp., http://developer.intel.com. *Intel Xscale Microarchitecture for the PXA255 Processor*, March 2003.

[21] Intel Corp. *Intel Itanium 2 Processor Reference Manual*, May 2004. `http://developer.intel.com/design/itanium/family`.

[22] Bruce Jacob and Trevor Mudge. Virtual memory: Issues of implementation. *IEEE Computer*, 31:33–43, June 1998.

[23] Bruce L. Jacob and Trevor N. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organisations. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 295–306, San Jose, CA, USA, October 1998. ACM.

[24] Dave Jagger, editor. *Advanced RISC Machines Architecture Reference Manual*. Prentice Hall, July 1995.

[25] L4/Alpha. `http://l4alpha.sourceforge.net/`, March 2001. Source of L4 for SMP-Alpha from UNSW.

[26] L4Ka Team. L4Ka::Pistachio kernel. `http://l4ka.org/projects/pistachio/`.

[27] L4Ka Team. *L4* e*X*perimental *Kernel Reference Manual Version X.2*. University of Karlsruhe, October 2001. `http://l4ka.org/projects/version4/l4-x2.pdf`.

[28] L4/MIPS source code, kernel version 79. Available from `http://www.disy.cse.unsw.edu.au/Software/L4`, February 1999.

[29] 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.

[30] 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.

[31] Jochen Liedtke. *L4 Reference Manual — 486/Pentium/PentiumPro, Version 2.0*. GMD, Schloß Birlighofen, Germany, September 1996. Working Paper 1021.

[32] Jochen Liedtke. μ-Kernels must and can be small. In *Proceedings of the 5th IEEE International Workshop on Object Orientation in Operating Systems*, pages 152–161, Seattle, WA, USA, October 1996. IEEE.

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

[34] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for extensibility). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 28–31, Cape Cod, MA, USA, May 1997.

[35] Jochen Liedtke and Horst Wenske. Lazy process switching. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, Schloss Elmau, Germany, May 2001.

[36] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann, 1994.

[37] David Nagle, Richard Uhlig, Tim Stanely, Stuart Sechrest, Trevor Mudge, and Richard Brown. Design tradeoffs for software-managed TLBs. In *Proceedings of the 20th International Symposium on Computer Architecture*. ACM, 1993.

[38] David A. Patterson and John L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., 2nd edition, 1998.

[39] M. Rozier, V. Abrossimov, F. Armand, L. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*, pages 39–69, Seattle, WA, USA, April 1992.

[40] Curt Schimmel. *UNIX Systems for Modern Architectures*. Addison Wesley, 1994.

[41] Jonathan S. Shapiro. Vulnerabilities in synchronous IPC designs. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, April 2003.

[42] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[43] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 3rd edition, 1998.

[44] Cristan Szmajda. A new virtual memory implementation for L4/MIPS. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 1999. Available from `http://www.cse.unsw.edu.au/~disy/papers/`.

[45] Madhusudhan Talluri. *Use of Superpages and Subblocking in the Address Translation Hierarchy*. PhD thesis, University of Wisconsin-Madison Computer Sciences, 1995. Technical Report #1277.

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