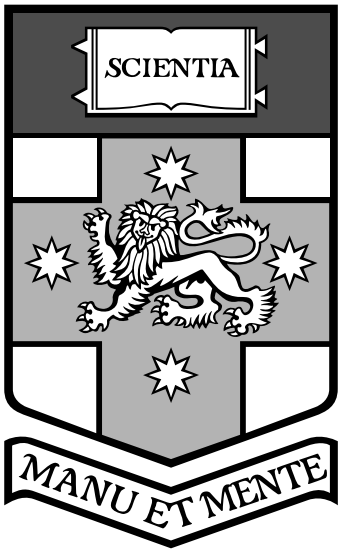
The University of New South Wales
School of Computer Science and Engineering



From 'Real Fast' to Real-Time : Quantifying the Effects of Scheduling on IPC Performance

David Greenaway

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

5 June, 2007

Supervisor: Dr. Kevin Elphinstone
Assessor: Dr. Stefan M. Petters

Acknowledgements

A countless number of people have assisted me in many different ways over the last twelve months, only a tiny few of whom I can recognise in this short space.

I thank my supervisor, Kevin Elphinstone, for his his invaluable guidance and wisdom; all the people from ERTOS and Open Kernel Labs, for providing a friendly environment to work, and many of whom kindly donated cycles of their time when really they had none to spare; my parents, for their understanding and (vast) patience; and finally, Hui Yee Chin, not only for staying up until the early hours of the morning to proofread, but more importantly for her constant encouragement, support and friendship over these last twelve months.

Contents

Chapter 1

Introduction

L4 is increasingly being utilised as a software platform for commercial embedded systems, and has already been deployed in mobile phones with soft real-time requirements. While L4's design has traditionally been optimised for best-effort systems where throughput is the primary concern, support for real-time applications is increasingly being required of the kernel.

In this thesis, we look at three scheduling optimisations that have traditionally been used in L4 to increase best-effort system performance: lazy-queueing, direct process switching and FIFO IPC send-queue ordering. Although all three optimisations have served in the past to provide L4 with blazingly fast performance, they have also come at the price of reducing L4's theoretical real-time capabilities.

We qualitatively describe the trade-offs associated with these three optimisations and quantitatively measure the actual benefits and costs of the optimisations, both in terms of throughput and latency. We perform these measurements by benchmarking several modified versions of L4, each implementing a different combination of the three optimisations.

1.1 Document Overview

Chapter 2 of this document gives an overview of L4 and describes its current scheduling algorithms and policies in detail. The implementation of the three optimisations of lazy-queueing, direct process switching and FIFO IPC queueing are described in detail, along with the theoretical benefits and costs of each.

Chapter 3 describes the different L4 kernel implementations used for our experiments, as well as the custom data structures and micro-optimisations implemented on each. This

chapter also describes an in-kernel scheduling API that we developed to assist development of each of the kernel variants.

Chapter 4 describes the methodology we used to evaluate each of the kernels. Four methods were used: (i) gathering statistics on how workloads interact with the kernel; (ii) measuring the throughput of each kernel with a macro-benchmark; (iii) measuring the latency and speed of specific kernel operations using micro-benchmarks; and finally (iv) profiling each of the kernels to determine the time spent in the scheduler and IPC paths.

Chapter 5 analyses the results of each of our benchmarks, providing quantitative measurements of the advantages and costs of each of the optimisations.

Finally, Chapter 6 draws conclusions from the results, and discusses possible future directions of work.

Chapter 2

Background and Related Work

2.1 The L4 Microkernel

A *kernel* is the part of an operating system running in the processor's privileged mode that is responsible for providing services that allow processes on the system to carry out their tasks effectively. Traditionally, kernels have attempted to provide all the services required for a fully functioning operating system, such as device drivers, file systems and networking services. While such *monolithic* kernels are flexible and convenient to construct, these advantages come at the cost of stability and security: a bug in any part of the large kernel may cause the entire operating system to become corrupt or the system's security to be compromised.

In contrast to these fully-featured monolithic kernels, the primary goal of a *microkernel* is to have the minimal amount of functionality running in the processor's privileged mode that still allows the implementation of a system's required functionality [9, 7]. Many of the services that would ordinarily be provided by a monolithic kernel (for instance device drivers) are instead implemented as user-level servers. By reducing the quantity of privileged code and increasing the isolation of such system services, a greater level of security and stability can be achieved. A bug in an isolated device driver may require the driver to be restarted, but will avoid corrupting the entire system [5].

The NICTA L4-embedded N2 microkernel [12] is a second-generation microkernel designed from the ground-up for high-performance. The kernel's API has been designed with a focus on use with embedded systems. L4-embedded provides three primary abstractions to user-level tasks:

Threads : A *thread* represents a single flow of execution within the system. Every thread has its own program counter, stack and set of registers. The kernel's *scheduler*

determines which thread should be executing on the processor at each point in time.

Address spaces : Each *address space* in L4 provides a virtual address space with mappings to physical memory. Each address space may contain one or more threads. Threads can be protected from each other by isolating the threads in different address spaces.

Inter-process communication : *Inter-process communication* (IPC) allows threads, potentially in different address spaces, to communicate with each other. L4 uses *synchronous* IPC, in that a thread attempting to send a message is unable to continue execution until either the message has been received by its destination thread or the send is aborted.

Three types of IPC operations are possible: (i) *send*, which sends a message to another thread; (ii) *receive*, which receives a message from another thread; and (iii) *send/receive*, which sends a message to another thread and then, once the message has been successfully received, receives from another thread. A *call* IPC operation is a special case of send/receive where the thread being sent to is the same as the thread being received from.

2.2 The Importance of IPC Performance

Two primary advantages of microkernels are increased security and stability. For these advantages to be realised, system services must be modularised into separate isolated servers. These modular operating system components communicate to each other through IPC thousands of times per second, each potentially causing a context switch. IPC is also used in L4 for each interrupt delivery [10] and is also used by para-virtualised Linux and Darwin for each syscall and trap into the kernel [6, 16].

As IPC is such a frequent operation (with thousands of messages taking place each second), it has a significant potential to introduce high overheads into the system. Figure 2.1 shows the theoretical performance overhead of IPC as a function how frequently IPCs are performed. As the frequency of IPC increases to the 1000 to 10 000 cycle range, system performance becomes quite sensitive to IPC times. These calculations fail to take into account secondary effects such as cache and TLB usage by the IPC path, which may contribute even more to overhead. Such secondary effects were found to be significant factors in the poor performance of the first-generation Mach microkernel [1].

If IPC performance is slow, system designers will be forced to recombine system components back together in order to restore performance, defeating the primary advantages

gained by using the microkernel in the first place. For these reasons, significant effort has gone into ensuring that IPC remains as fast as possible. In the construction of the L3 microkernel, a predecessor to L4, Liedtke [8] asserted that "anything which may lead to higher IPC performance has to be discussed. In case of doubt, decisions in favour of IPC have to be taken."

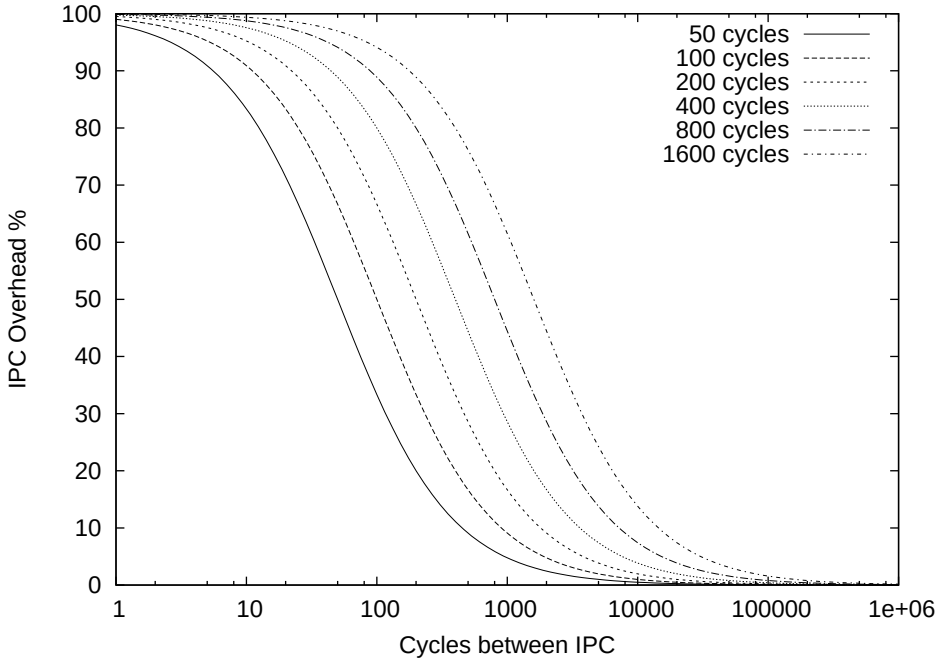


Figure 2.1: *Total system overhead attributable to IPC as a function of the number of cycles between each IPC. Each curve represents a different length IPC call, measured in cycles. The range of values shown are typical IPC times of a modern microkernel. Reproduced from Elphinstone et al [2].*

2.3 Thread Scheduling

The *scheduler* of a kernel is responsible for determining which thread on the system should execute at each point in time. Such decisions tend to be made by the scheduler each time a new thread becomes ready to run (such as when a thread is woken by an IPC message), when the currently running thread blocks (such as by calling another thread with IPC), and also at fixed time intervals multiple times a second (allowing the scheduler to share processor time between multiple threads).

Schedulers tend to be designed to make decisions according to a defined policy, such as always choosing to schedule high-priority threads over low-priority threads, or giving each

thread in the system a fair share of processor time.

A microkernel scheduler can be evaluated using the following metrics:

Scheduling Overhead The *overhead* of a scheduler is the percentage of system resources (most often measured in processor time) the scheduler requires to perform its function. Simple schedulers will often be able to make scheduling decisions with very little overhead, while more sophisticated schedulers may require more time and memory to make their (potentially 'better') decisions. Because IPC often involves the interaction of more than one thread, the scheduler may need to be involved in IPC operations. As such, inefficient schedulers may cause significant increases in IPC times.

Interrupt Latency A scheduler's *interrupt latency* is the amount of time between an external event occurring (in the form of an interrupt) and the thread responsible for handling the event being scheduled, as shown in Figure 2.2. Long interrupt latencies result in the system being unable to respond to events in a timely fashion. Short and predictable interrupt latencies are particularly important to the class of systems known as 'real time systems', described further in Section 2.3.1.

The minimum interrupt latency of a kernel is determined by the amount of time the kernel takes to switch from the current thread to the interrupt handling thread. Interrupt latencies may be longer if an interrupt occurs while the kernel is in the middle of a long-running operation that cannot be interrupted. For this reason, the maximum interrupt latency in a system is directly proportional to the length of the longest non-preemptable operation in the kernel.

For example, older versions of L4 took significant amounts of time to delete address spaces with complex page mappings associated with them. Any interrupt that fired towards the beginning of such a delete operation would be forced to wait for the entire operation to complete before being serviced.

Accuracy The *accuracy* of a scheduler is its ability to ensure that threads are only given the resources they are entitled to, as dictated by the scheduling policy. For instance, a strict priority-observing scheduler should not allow low-priority threads to be executed while high-priority threads are waiting. Similarly, a round-robin scheduler should give two equal-priority threads an equal amount of time on the processor.

These scheduling requirements are often in conflict: optimisations that help reduce scheduler overhead may come at the cost of higher interrupt latencies (such as the 'lazy queueing' optimisation, described in Section 2.4.3). Others may come at the cost of reducing accuracy (such as the 'direct process switch' optimisation, described in Section 2.4.2).

The design of a scheduler needs to be based on the precise requirements of the processes running on the system, with scheduler trade-offs made accordingly.

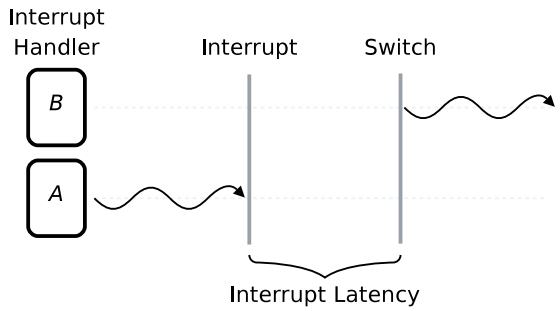


Figure 2.2: *Interrupt latency is the time taken between an interrupt firing and the first instruction of the interrupt handler executing.*

2.3.1 Real-Time Scheduling

An increasingly large proportion of embedded systems require strict timing guarantees from the operating system. Such systems, known as *real-time* systems, not only rely on functional correctness, but also temporal correctness (i.e., the correctness of the system depends on *when* the system finishes its calculation).

Real time systems fall into two broad categories: *hard* real-time systems and *soft* real-time systems. The former category requires strict guarantees that deadlines will be met; missing a deadline may result in injury or loss of life. An embedded system monitoring sensors in a car to determine if and when an airbag should be released would be considered an instance of a hard real-time system.

The latter class of soft real-time systems requires only statistical guarantees that deadlines will be met. Failure to meet a single deadline reduces the effectiveness of the system, but will not cause a complete failure. A mobile phone that suffers a degradation of sound quality when deadlines are missed would be considered an instance of a soft real-time system.

In order to provide support for real-time systems, kernels must provide guaranteed upper bounds on both interrupt latencies and the time taken by system calls. Real-time scheduling algorithms are also required to ensure that threads are given sufficient processor time at the time they require it. Two popular real-time scheduling algorithms for real-time systems with threads that have periodic deadlines are the *rate-monotonic scheduling* (RMS) algorithm and the *earliest deadline first* (EDF) algorithm [11].

RMS gives a unique priority to each thread in the system, sorted by the frequency of deadlines that each thread must meet. Threads with a high rate of deadlines are given highest priority, while threads with a lower rate have lower priorities. EDF also gives unique priorities to each thread, but dynamically modifies the priorities such that the thread with the closest deadline is given the highest priority. Both algorithms provide guarantees that all deadlines will be met regardless of the number of real-time threads in the system so long as system utilisation remains below 69% for RMS or 100% for EDF.

2.4 Thread Scheduling in L4

The current implementation of L4-embedded N2 uses a priority-based scheduling scheme, carrying out round-robin scheduling on threads with equal priorities. Conceptually, at any point in time each thread in the system is either *running*, *ready* to run, or *blocked* waiting for an external event.

Each thread in the system is also assigned a *priority*, from 0 to 255 inclusive. A thread will only be scheduled to run if there are no ready threads of a higher priority. If multiple threads of the same priority are all ready, each thread will run for a fixed amount of time (known as its *timeslice*) before control is passed to the next thread of the same priority in a round-robin fashion.

Timeslice lengths in L4 are enforced by the kernel periodically probing the system to determine if the current thread's timeslice has expired. Each of these probes is known as a *timer tick*, and typically occurs once every 1 ms to 10 ms, depending on the system architecture [13].

2.4.1 Scheduling Implementation

The L4 scheduler is implemented by having a *ready-queue*, which keeps track of all threads currently in a state ready to run. Each time a new thread needs to be selected to be scheduled, the scheduler searches the ready-queue for the highest priority thread, which then begins execution.

The L4 ready-queue data-structure is implemented as an array of 256 queues, one queue for the ready threads of each priority level as shown in Figure 2.3. Each thread control block (TCB) is linked to the next and previous TCB of the same priority level. A thread can be added to the scheduling queue in O(1) time by adding it to the tail of the queue

associated with its priority. Threads can similarly be removed from the queue in O(1) time by updating the pointers of the previous TCB and the next TCB in the queue.

As is, the ready-queue with p priority levels would require O(p) operations to determine the highest priority thread, found by iterating over the array of queues from highest to lowest priority. To avoid the scan, a two-level tree of bitmaps is used to improve search speed, depicted in Figure 2.4. The lower level consists of p bits, bit i being set if a thread is queued at priority i. Each bit in the upper level corresponds to a word in the lower level. The upper-level bit is set if any bit in the lower level word is active. This bit-tree structure can be generalised to support any arbitrary number of p threads by having $\lceil \log_w p \rceil$ levels, assuming w-bit words.

Some modern processors, such as the ARM family of processors [3], are able determine the highest bit set of a word in a single operation. This allows the scheduler to determine the highest priority thread with just a single operation on each of the $\lceil \log_w p \rceil$ levels of the bitmap (a total of two operations in the current L4 scheduler). Processors without hardware support for finding the highest priority bit must do a little more work, but can still find the highest order bit of a w-bit word in O($\log_2 w$) operations, making the entire lookup operation take O($\lceil \log_w p \rceil \cdot \log_2 w$) operations (approximately ten operations in the current L4 scheduler implementation, though each operation is likely to be more involved than those on processors with hardware support for lookups).

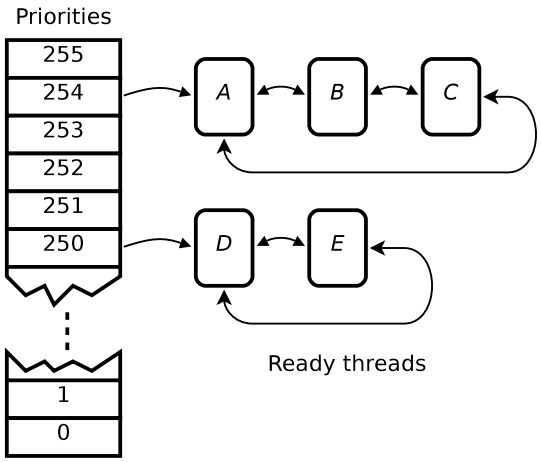


Figure 2.3: *The scheduling queue data structure. The next thread to be selected by the scheduler will be thread A, which will then be moved to the end of the queue. Threads D and E will not be considered by the scheduler while ever ready threads of a higher priority are in the queue.*

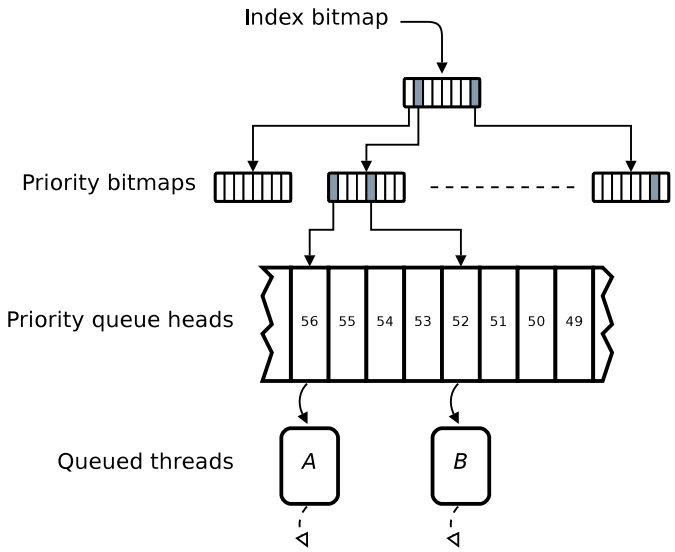


Figure 2.4: *The scheduling queue bitmap tree data structure. Two levels of bitmaps form a simple tree. Each queue that contains a thread will set the corresponding bit in the tree.*

2.4.2 Direct Process Switch

In a kernel that implements a strict scheduling policy, each time a thread sends an IPC to another thread, the scheduler must be invoked to determine which thread in the system has the highest priority so that flow of control can be given to it. In the majority of cases, the scheduler's decision will be merely to keep letting the sender execute or to switch to the destination of the IPC. Involving a heavy-weight scheduler each time a message passes between two threads introduces a significant level of overhead to each IPC call, often unnecessarily.

To avoid the scheduler overhead in this situation, Liedtke [8] proposed an optimisation termed a *direct process switch*, which allows the IPC path to decide which thread should be scheduled at the completion of an IPC operation without consulting the scheduler. Assuming thread A is currently executing, thread B is ready to receive from A, and thread C is ready to send to A, then:

- if thread A **sends** to thread B, either A or B will be switched to (whichever has highest priority);
- if thread A **calls** thread B, thread B will be switched to;
- if thread A **receives** from thread C, thread A or C will be switched to (whichever has higher priority); and finally

- if thread A performs a **send/receive** to B and from C, thread B or C will be switched to (again, whichever has higher priority).

The scheduler's ready-queue is not consulted in any of these cases to determine if the thread being switched to is the highest-priority thread in the system. Instead, the sending thread 'donates' the time between the IPC taking place and the next timer tick to the destination thread, a process misleadingly termed *timeslice donation*[1]. When the next timer tick takes place, the scheduler will be invoked and will restore control to highest priority thread ready to execute.

The primary consequence of using direct process switching is that thread priorities are no longer strictly observed by the kernel: if a high-priority thread calls a low-priority thread, the low-priority thread will be scheduled, even if other threads are ready to execute. This time that a thread executes while a higher priority thread is ready to run is known as *temporary priority inversion*, and is depicted in Figure 2.5.

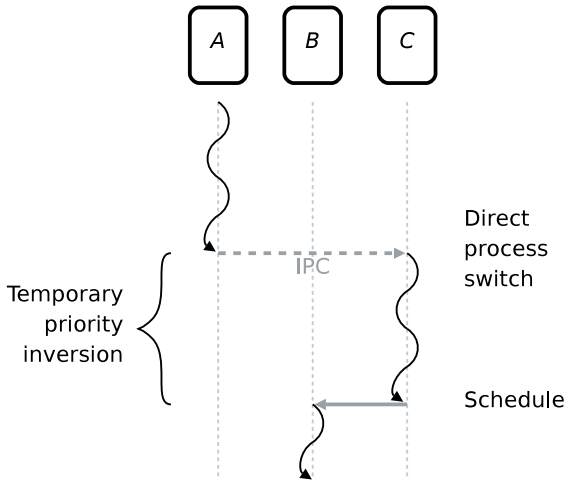


Figure 2.5: *Temporary priority inversion caused by the direct process switch optimisation. High-priority thread A performs an IPC to low-priority thread C, causing it to begin execution. When the remainder of A's timeslice runs out thread C will be preempted, and the intermediate-priority thread B will be scheduled.*

[1]The name 'timeslice donation' originates from previous versions of L4 which actually did donate the remainder of the current thread's timeslice to the destination of the IPC [14]. In contrast, modern versions of L4-embedded only donate the time up until the next timer tick, which may be a significantly shorter length than the remainder of the thread's timeslice.

2.4.3 Lazy Queueing

Each time an IPC operation in L4 takes place, multiple manipulations of the scheduler's ready-queue are required. A standard call between two threads A and B would involve four queue operations: (i) A is removed from the ready-queue as it passes control to B; (ii) B is added to the ready-queue as it begins to execute; (iii) B is removed from the ready-queue as it returns control back to A; and finally (iv) A is added back on to the ready-queue again.

In most cases it is expected that thread A will be calling B to perform a simple operation which will then immediately return control back to A. The scheduling queue is modified four times only to return back to its original state.

In an attempt to avoid the scheduling operations altogether in the common case of one thread calling another, *lazy queueing* [8] was introduced into L4. Two forms of lazy queueing are possible: *lazy enqueueing* and *lazy dequeueing*, both of which are implemented in L4.

Lazy Dequeueing

A kernel implementing lazy dequeueing no longer guarantees that its scheduler queue only contains runnable threads. Instead, it will contain *at least* every runnable thread, but may also contain non-runnable threads, as shown in Figure 2.6. When a thread in the ready queue becomes blocked, no action is taken to dequeue it. If the thread becomes runnable again while still on the scheduling queue, the time that would have been spent dequeueing the thread (only to enqueue it later) is saved.

While searching the ready-queue to find the highest priority thread ready to execute, the scheduler may discover that the first thread on the scheduling queue is actually in a blocked state. In this case the scheduler simply dequeues this thread and moves on to the next thread in the queue. Such an operation is termed a *deferred dequeue*—the dequeue operation that should have taken place when the thread lost its runnable status was deferred to a later time. Theoretically, lazy queueing will never add overhead to the scheduler: in the best case, queue operations are avoided. In the worst case, queue operations are merely deferred to a later time, but precisely the same number of operations are performed.

Lazy-dequeueing may, however, cause a potentially large amount of work to be deferred at a later time. If a large number of blocked threads build up on the ready-queue, the

scheduler may need to dequeue them all in a single non-preemptable operation. This has the potential to cause significant delays in interrupt delivery.

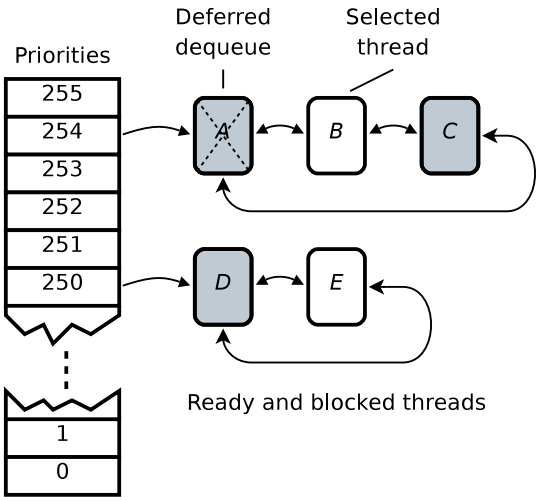


Figure 2.6: *The scheduling queue with lazy queueing enabled. The shaded boxes represent blocked threads. In the queue's current state, the scheduler will determine that A is blocked, remove it from the queue and then choose B. Threads C and D will remain on the queue despite being in a blocked state as the scheduler will not reach them prior to selecting B.*

Lazy Enqueueing

A kernel implementing lazy enqueueing removes the restriction that the scheduler's ready-queue must contain the currently executing thread. With this optimisation, a thread that executes for a short burst of time may be able to complete is execution without ever being placed on the ready-queue, saving two queue manipulations.

For example, if the currently running thread A calls thread B (which is not currently on the scheduling queue), B will execute directly without any queue operations taking place. If thread B then returns control back to thread A, two enqueue operations will have been avoided (in addition to the two operations avoided from lazily dequeueing thread A).

If thread B is interrupted prior to entering a blocked state again, only then will it need to be added to the ready-queue. Such an operation is referred to as a *deferred enqueue*. This may occur, for instance, if thread B is preempted by an interrupt causing a context switch, or if thread B performs a send IPC to third thread C leaving both B and C ready

to run, but C executing. As with lazy dequeueing, lazy enqueueing will not cause any additional work to be performed, but in the worst case merely defers *when* the queueing operations occur.

2.4.4 FIFO IPC Queueing

Threads attempting to send an IPC to a thread not ready to receive can choose to either abort the IPC operation or block, waiting until the destination becomes ready to accept the IPC. Sending threads that are forced to block are added to the *IPC send queue* of the destination thread. When the destination thread finally becomes ready to receive, it will determine if any threads are currently waiting on its queue and, if so, will take one of those threads off and complete its IPC operation.

The L4 kernel implements the IPC send queue as a simple first-in-first-out (FIFO) queue: threads are processed from the queue in the same order that they are added. The primary benefit of using FIFO queues is that both enqueue and dequeue operations can occur in O(1) time, helping to reduce IPC times.

The use of FIFO send queues may introduce an indirect form of priority inversion, as high-priority threads can become queued behind lower-priority threads. Such high-priority threads will be forced to wait for the lower-priority threads to be served before being able to continue.

2.5 Real-Time Scheduling in L4

As L4 is utilised more and more in embedded systems with soft real-time requirements, the importance of providing timing guarantees has increased. Unfortunately, the use of the three scheduling optimisations described in Section 2.4.1 all hinder L4's suitability for use in real-time systems.

Both the RMS and EDF scheduling algorithms use priorities to ensure that threads are executed at the correct time so that deadlines are met. Both algorithms assume that the priorities of threads will be strictly adhered to. The direct process switching and FIFO queueing optimisations of L4 unfortunately both violate this assumption.

Additionally, to be used as a real-time system, L4 must ensure that interrupt latencies have a predictable upper bound. The introduction of lazy-queueing has the potential to introduce large latencies into the kernel when the number of active threads is large. In a real-time system where the number of threads in the cannot be determined in advance,

the length of such latencies are unpredictable. Alternatively, if a fixed upper bound on the number of threads can be determined during system construction, the maximum latency caused by lazy-queueing can be measured and accounted for, and presents less of a problem.

2.6 Related Work

Liedtke [8] originally proposed both the direct process switch and lazy-queueing optimisations for the L3 operating system, a predecessor to L4. While Liedtke's tests showed that removing lazy-queueing slowed down IPC by up to 23%, he did not consider the costs of increased interrupt latency nor quantify the performance advantages of direct process switching. Fourteen years of development and a change of CPU architecture has also taken place between L3 and L4-embedded N2, which may have also had an effect on the trade-offs of the optimisations.

Ruocco [13] investigated the direct process switch scheduling behaviour in L4, concluding that "the extreme performance optimisations that L4-embedded inherited from previous implementations, especially those performed in the critical IPC path, are, to a large degree, the main sources of complexity for real-time scheduling" and that a "review of the current trade-offs between performance and predictability would ease priority-driven real-time programming." This thesis attempts to perform such a review.

Krishnapura [4] investigated how the EDF scheduling scheme could be implemented at user-level on top of the existing fixed-priority round-robin scheduler, by ensuring that only one thread was runnable at any point in time. By having only one thread runnable at any time, the problem of priority inversion from direct process switching is overcome, but care must still be taken to prevent priority inversion from IPC FIFO queueing.

Steinberg [14] modified a variant of L4 named 'Fiasco' to give threads the option of donating both their current timeslice and all future timeslices when performing an IPC. Such semantics form a basic priority inheritance protocol and allow the kernel to implement the direct process switch optimisation whenever threads elect to carry out this donation. Steinberg did not quantify the performance costs of the changes, however, despite the implementation being quite intricate.

2.7 Summary

L4 uses three optimisations that, while potentially reducing scheduling overhead in the system and hence increase system throughput, come at the cost of increased latencies and decreased scheduler accuracy. These trade-offs are summarised in Table 2.1.

These problems hinder L4's suitability for being used as a platform for real-time systems. In this thesis, we attempt to quantify both the advantages and costs of these three optimisations so that system implementers can make an informed choice when deciding whether such optimisations should be used.

	Effect on Scheduler		
Optimisation	**Overhead**	**Latency**	**Accuracy**
Direct Process Switch	✓		✗
Lazy Queueing	✓	✗	
FIFO IPC Queueing	✓		✗

Table 2.1: *Summary of the optimisations used in the IPC path. Ticks indicate an improvement of the named attribute, while crosses indicate a degradation. The three optimisations all help to decrease scheduler overhead, but come at the cost of latency or accuracy.*

Chapter 3

Kernel Implementations

To quantify the performance trade-offs of the three scheduler optimisations described in Chapter 2 we created multiple variants of the L4-embedded N2 kernel, each with a different set of optimisations applied.

In particular, we created four variants of the kernel with different combinations of the direct process switch and lazy-queueing optimisations, and three variants of the kernel with different methods of IPC queueing that ensure thread priorities are observed. These kernel variants are described in detail below.

3.1 NICTA L4-embedded N2 1.3.0

All of our tests are based on the NICTA L4-embedded N2 1.3.0 kernel [12], which we modified to implement different scheduling optimisations and data structures. We also ran all tests on the original unchanged kernel to provide a performance baseline and to allow our modified implementations to be compared to the unmodified kernel.

One feature of the L4-embedded N2 kernel designed to reduce the cost of IPC is the *IPC fastpath*. The IPC fastpath is a highly-optimised assembly implementation of the routines in L4 carrying out IPC between threads. The fastpath gains its speed by paying careful attention to avoiding unnecessary cache misses, avoiding C calling conventions, and by avoiding complex checks required for less common IPC cases.

By avoiding these checks, the IPC fastpath is unable to fulfill the more complex IPC calls, which would be too time consuming to optimise in assembly. Any call that the IPC fastpath can not handle is redirected to the *IPC slowpath*, a C-based implementation of the IPC routines. The IPC slowpath is able to handle all IPC cases, but comes at a performance cost.

In designing the fastpath, the L4 designers assume that it will be able to handle the majority of IPC calls, with only an occasional call needing to be demoted to the slowpath. This provides the best of both worlds: most calls are able to be carried out at high speed without sacrificing advanced functionality.

For all our experiments, we implemented our data structures and modifications to the IPC path entirely in C. Unfortunately, it becomes difficult to make meaningful conclusions by comparing the unmodified assembly-optimised L4 kernel with our modified C-based kernels. For this reason, we tested two versions of the unmodified L4 kernel. The first version, the Fastpath kernel, is the unmodified L4 kernel with all optimisations enabled. The second version, the Slowpath kernel, is the L4 kernel with the IPC fastpath disabled, forcing all IPC calls to pass through the C-based IPC routines.

Any performance differences between the Fastpath and Slowpath kernel can be attributed to the advantages of optimising the IPC path in assembly. Differences between the Slowpath and other kernels in our experiments should be closely related to the changes in algorithms, optimisations and data structures made by our experiments.

3.2 Internal Scheduling Interface

As a result of the direct process switching optimisation, L4 has been designed in such a way that a scheduling decision is made locally each time two threads interact. For instance, if one thread sends an IPC message to another, the code in the IPC path itself determines which thread should next be executed, enqueues the alternate thread onto the scheduler's ready list, and then switches execution to the first thread without any other calls being made to the scheduler. Examples of this implicit scheduling code style are shown in Figure 3.1.

While such implicit scheduling decisions allow the lazy queueing and direct process switching optimisations to be implemented more easily, the make it difficult to introduce different scheduling policies or implementations into the kernel without making significant modifications to large amounts of the kernel. Any change in scheduling policy would require that modifications be made to every one of the many implicit scheduling points scattered throughout the code.

For this reason, we introduced an in-kernel abstract scheduling interface into L4 to allow scheduling policies and implementations to be switched with relative ease. All points in the code that make localised scheduling decisions or performed ready-queue manipulations were rewritten to use a new scheduler API.

```
/* Execute an IPC call between 'current' and 'dest' */
void sys_ipc(tcb_t * current, tcb_t * dest, ...)
{
    ...

    /* Switch to the destination thread if they are higher
     * priority than us, otherwise keep running */
    dest->set_state(STATE_READY);
    if (dest->priority > current->priority) {
        enqueue(current);  /* Lazy enqueue of dest */
        switch_to(dest);
    } else {
        enqueue(dest);
    }
    ...

    /* Call the destination thread and wait for a reply */
    dest->set_state(STATE_READY);
    current->set_state(STATE_WAITING);
    switch_to(dest);  /* Lazy dequeue of current */
    ...
}
```

Figure 3.1: *Examples of the implicit scheduling decisions made in the L4 code. The first code segment shows the actions that take place on a 'send-only' IPC call. The second shows the actions for a 'send-wait' IPC call.*

The original process of introducing the API took quite some time, approximately 60 engineering hours. Once the API was in place, however, changes to the scheduler could be made with relative ease. Modifying the scheduler from using direct process switching to a strict priority-observing scheduler took approximately 4 engineering hours for a basic yet fully-functional implementation.

3.2.1 Interface API

At a basic level, a scheduler only needs to know through its interface *which* threads are ready to be scheduled and *when* a scheduling decision needs to be made. Details such as the actual thread that should be scheduled at any given time or how long that thread should be allowed to run for should be left to the scheduler implementation.

The scheduler API, shown in Figure 3.2, introduces three basic mechanisms: an '`enqueue`' operation that informs the scheduler that a given thread is ready to be scheduled, a '`dequeue`' operation that informs the scheduler that a thread can no longer be scheduled, and a '`schedule`' operation that requests the scheduler to make a decision about which

```
/* Hints describing traditional L4 behaviour */
typedef enum {

    /* Schedule highest priority thread */
    HINT_HIGHEST_PRIORITY,

    /* Schedule most recently enqueued thread */
    HINT_NEW,

    /* Schedule the current or just-enqueued thread */
    HINT_CURRENT_OR_NEW,

} hint_t;

/* Ready queue manipulations */
void enqueue (tcb_t *);
void dequeue (tcb_t *);
void swap    (tcb_t *, tcb_t *);

/* Request scheduler to perform a context switch */
void sched (hint_t);

/* Manipulate ready-queues and perform a switch */
void enqueue_sched (tcb_t *,          hint_t);
void dequeue_sched (tcb_t *,          hint_t);
void swap_sched    (tcb_t *, tcb_t *, hint_t);
```

Figure 3.2: *The new L4 internal scheduling API*

thread should next execute and (if it is different from the currently executing thread) switch control to it.

The scheduling API also introduced methods that allow multiple commands to be issued to the scheduler at once, such as 'swap', which performs an enqueue and dequeue atomically, and also 'enqueue/schedule', 'dequeue/schedule' and 'swap/schedule' which perform a ready-queue manipulation followed directly by a scheduling operation. More advanced schedulers can optimise these combined calls, potentially saving work.

Each of the API calls involving a 'schedule' operation also takes an additional parameter that we termed a *scheduling hint*. Scheduling hints are passed to the scheduler calls and inform the scheduler which thread the direct process switching optimisation would have chosen to schedule. A scheduler that does not implement direct process switching is free to ignore the hints without further consequence, while a scheduler that *does* implement the behaviour can use the hints to avoid having to search the ready-queue and allow it to precisely emulate the behaviour of L4 prior to the introduction of the scheduling API.

Three scheduling hints were required to specify the traditional scheduling behaviour of L4: (i) a hint indicating that the highest priority thread in the system should be scheduled, (ii) a hint indicating that the most recently enqueued thread should be scheduled (used

```
/* Execute a IPC call between 'current' and 'dest' */
void sys_ipc(tcb_t * current, tcb_t * dest, ...)
{
    ...

    /* Enqueue 'dest', and perform a reschedule. */
    dest->set_state(STATE_READY);
    enqueue_sched(dest, HINT_CURRENT_OR_NEW);
    ...

    /* Dequeue 'current', enqueue 'dest', and perform
     * a reschedule. */
    dest->set_state(STATE_READY);
    current->set_state(STATE_WAITING);
    swap_sched(dest, current, HINT_NEW);
    ...
}
```

Figure 3.3: *The code from Figure 3.1, rewritten using the internal scheduler API.*

in the IPC path when a thread calls a second thread), and (iii) a hint indicating that the most recently enqueued thread or the currently executing thread should be scheduled, whichever has the highest priority (used in the IPC path when a thread sends to a second thread without waiting for reply.)

One significant limitation of the interface is that it still requires programmers to manually set a 'thread state' variable, as shown in Figure 3.3. Ideally, thread state information should be managed entirely by the scheduler, being set to 'ready' on an enqueue and reset to 'waiting' on a dequeue. Unfortunately, the thread state variable in L4 is overloaded to store more information than just that relevant to scheduling. For instance, the SMP implementation of L4 uses the thread state variable to mark threads that are to be scheduled on another CPU, to indicate when a TCB is locked for writing, and to facilitate IPC between threads on different CPU cores. In the future it would be ideal if these two separate functions (scheduling status and SMP functionality) were separated, allowing for a cleaner scheduling API.

3.2.2 Reducing Abstraction Overhead

One significant problem in introducing an abstraction layer inside the kernel is the possibility of adding overhead to all scheduling-related functions. Such overhead may arise from the additional function calls required to call the scheduler (and the associated stack-maintenance costs of maintaining the C calling convention involved in such calls), or

having code behind the API that is over-generalised (which may prevent optimisations that do not apply to *all* locations being applied at *any* location).

The first issue was addressed by enabling aggressive compiler inlining, allowing the compiler to avoid the costs of carrying out function calls. Additionally, we were able to write the scheduling code in such a way that unnecessary code (such as the code for scheduling hints that do not apply to a given scheduler call) can be optimised away by the compiler's dead-code removal pass.

The second issue was addressed through the use of the combined action API calls (which allow the scheduler to optimise for cases where multiple actions are taking place at once), and also by optimising for each scheduling hint individually.

Further, the centralisation of the scheduling code allowed us to optimise the scheduling code at one central location, benefiting the entire kernel. Time could be spent on micro-optimisations that would have otherwise been too time consuming had they required changes to multiple different locations in the kernel.

3.3 Measuring Scheduling Optimisations

To determine the effect that the direct process switch and the lazy-queueing optimisation had on performance and latency, we implemented four kernels with different combinations of the two optimisations. The four kernel implementations are described in detail below.

3.3.1 Direct Process Switch / Lazy Queueing

The Direct/Lazy kernel implementation uses both the direct process switch and lazy-queueing scheduler optimisations, a behaviour mimicking that of the Slowpath kernel.

We used the kernel to quantify the performance impact of introducing the in-kernel scheduling API. Differences between the Slowpath and Direct/Lazy kernels can be attributed to the changes in code required by the scheduling API as both kernels use the same scheduling optimisations.

One impact of using the lazy-queueing scheduler optimisation with the scheduler API is that calls which involve dequeue operations need not be carried out immediately, but can take place at a deferred time if necessary. This causes the scheduler API calls `swap`, `dequeue_sched`, and `swap_sched` to become equivalent to `enqueue`, `sched` and `enqueue_sched` respectively.

While `dequeue` could potentially be ignored completely by the scheduler, we chose to treat the operation normally in our implementation because of its limited use in the kernel and to allow routines to safely be able to dequeue threads about to be destroyed. An alternate solution would have been to add an additional scheduler hint to all scheduler calls involving a dequeue allowing a caller to force a thread to be dequeued when lazy-queueing is in effect.

3.3.2 Direct Process Switch / Eager Queueing

The Direct/Eager kernel implementation uses the direct process switch optimisation like the Direct/Lazy kernel, but removes the lazy-queueing optimisation of L4, instead 'eagerly' taking threads off the ready-queue.

While the use of eager-queueing theoretically requires that more enqueue and dequeue operations take place, removing lazy-queueing allows the code to find the next priority thread to be simplified, no longer needing to handle deferred dequeues. Similarly, when threads that have been lazily-enqueued are preempted, they no longer need to be added to the ready-queue, slightly simplifying the scheduling logic.

3.3.3 Full Scheduling / Lazy Queueing

The Strict/Lazy kernel implementation disables the direct process switch optimisation while still utilising lazy-queueing. Removing direct process switching ensures that in almost all cases no thread will ever execute while a higher priority thread is ready to execute. The one exception arises from the L4 `ThreadSwitch` system call, which allows one thread to donate the remainder its tick to another. Because the call was used extensively in the initialisation routines in our testing platforms described in Chapter 4, this call could not be removed from the API nor replaced with different semantics that observe priorities.[1]

Even with strict scheduling, many micro-optimisations can be employed by the scheduler to avoid having to perform a full search of the ready-queue prior to each context switch. For instance, if we assume that no prior enqueue or dequeue operations have taken place since the last schedule, we can take the following shortcuts:

[1] The concept of a `ThreadSwitch` call is incompatible with strict priority-observing scheduling. As such, we did not attempt to strictly define the semantics of the call, which becomes troublesome when the destination of a `ThreadSwitch` call itself performs a `ThreadSwitch`, performs an IPC, or page faults requiring its pager to intervene. Instead, we cheat and merely claim that the executing thread is undefined after an `ThreadSwitch` call until the next timer tick.

- If an 'enqueue/schedule' call is made, either the current thread or the newly enqueued thread is going to have the highest priority in the system, so the higher of the two may be scheduled.
- If a 'swap/schedule' call is made which dequeues the current thread, the newly enqueued thread can be scheduled if its priority is greater than the current thread's priority (which is the highest priority thread in the system). If a strict FIFO ordering of threads is not required, this optimisation can also take place if the thread priorities are equal.

The question now becomes how to determine if previous enqueue or dequeue operations have taken place since the last schedule. One simple method would be to keep a flag that is set if a enqueue or dequeue call is made, and is cleared after each schedule. If the flag is clear, the above-mentioned optimisations can take place. If the flag is set, the ready-queue must be searched.

A more sophisticated method is to instead keep track of an upper bound on the highest priority thread. When a schedule call is made, either the current thread is the highest priority (in which case no context switch takes place), a recently enqueued thread is the highest priority (in which case execution switches to it), or a recently dequeued thread is currently cached as the highest priority (in which case a full ready-queue lookup must take place.) In any case, no more additional work is required by the scheduler compared to not using these optimisations, other than that of caching the highest thread priority. This takes place by increasing the currently cached priority if necessary each time an enqueue takes place, and resetting the cached value after each lookup.

These optimisations also allow lazy-queueing to become feasible. If a full ready-queue lookup was required each time a schedule took place, all ready threads would have to be enqueued at all times to ensure that the scheduler considered them (preventing lazy-enqueueing), and any high-priority blocked threads would be immediately dequeued (significantly reducing the effectiveness of lazy-dequeueing.)

3.3.4 Full Scheduling / Eager Queueing

The Strict/Eager kernel implementation removes both the lazy-queueing and direct process switch optimisations. This implementation uses the implementation optimisations described for both the Direct/Eager and Strict/Lazy kernels, providing the same benefits as both.

Since the two optimisations are mostly independent of each other, we would expect the both the performance decreases between the Direct/Lazy and Strict/Lazy kernels and the

decrease between the Direct/Lazy and Direct/Eager kernels to be present in the Strict/Eager kernel.

Optimistically, the performance decrease of removing the two optimisations might be less than the sum of removing each one. This would be the case if there is an overlap in the additional work required to be carried out by removing each of the individual operations. Pessimistically, the performance decrease may be worse if the two individual optimisations have 'synergistic' effects, such as those observed by Liedtke [8] when performing similar experiments on scheduler and IPC optimisations.

3.4 IPC Queueing Behaviour

The current L4 implementation uses a FIFO queue ordering, as described in Section 2.4.4. While FIFO queueing allows each queue operation to take place in O(1) time, it may cause a high-priority thread to become queued behind a lower-priority thread.

We attempted to determine the costs of processing queued IPC messages in priority order using three different kernel implementations. Each implementation is based on the Direct/Lazy kernel, such that any benchmark results should be directly comparable to the Direct/Lazy kernel.

3.4.1 Full IPC Queue Scan

We modified the Direct/Lazy kernel to perform a full scan of the IPC send queue each time a thread performs a receive operation. During the scan, the thread with the highest priority is determined and is used as the next thread to receive from. This implementation is called the IPC-Scan kernel.

Such an implementation allows fast O(1) enqueues of threads onto an IPC send queue, but requires $O(n)$ operations in order to scan the IPC send queue before a thread can be processed. In a pathological case, it may require $O(n^2)$ operations to complete n IPC operations.

3.4.2 Sorted IPC Queue

We modified the Direct/Lazy kernel again to insert new threads in IPC send queues such that threads are always sorted in priority order, forming the IPC-Ordered kernel. Such a method requires $O(n)$ operations to add a new thread to a send queue, but allows the

next thread to be processed to be determined in O(1) time. Like the IPC-Scan kernel implementation, it may still require $O(n^2)$ operations to complete n IPC operations in the worst case.

3.4.3 Heap IPC Queue

Our final implementation, IPC-Heap, converted the IPC send queues for each thread into a heap data structure. Heaps require $O(\log n)$ time to both add a new element onto the queue and remove a single element from the queue.

While using a heap is a significantly better choice of data structure in terms of time complexity, the operations for manipulating a heap are more involved than those involved in scanning a list for the highest priority thread or inserting a thread at the appropriate location in a list. The trade-off is that while using a heap data structure guarantees that n IPC operations will only take $O(n \log n)$ time, it may come at the cost of significantly slowing down other operations due to the housekeeping involved in maintaining the heap.

The heap implementation used in this implementation is a rough prototype and, due to time limitations, was not significantly optimised. However, the results of the implementation should still provide a general understanding of the costs of using a more complex data structure in a critical path.

3.5 Summary of Kernel Implementations

Kernel Name	Description
Fastpath	Unmodified L4-embedded N2 kernel implementation, with an assembly-optimised IPC code path.
Slowpath	Unmodified L4-embedded N2 kernel implementation, with no assembly IPC optimisations.
Direct/Lazy	Modified L4-embedded N2 kernel, with an added interface to allow the scheduler implementation to be easily modified.
Direct/Eager	As with Direct/Lazy, with eager queueing (i.e., lazy queueing optimisations disabled).
Strict/Lazy	As with Direct/Lazy, with full scheduling on context switches (i.e., direct process switch optimisations disabled).
Strict/Eager	As with Direct/Lazy, with both eager queueing and full scheduling on context switches (i.e., both direct process switch and lazy queueing optimisations disabled).
IPC-Scan	As with Direct/Lazy, with a full scan of IPC send queues carried out each time an IPC receive is performed to ensure the highest thread is processed first.
IPC-Ordered	As with Direct/Lazy, with the IPC send queues of each thread always kept in priority-order to ensure the highest thread is processed first when an IPC receive is performed.
IPC-Heap	As with Direct/Lazy, with the IPC send queues of each thread replaced with a heap data structure, ensuring that the highest thread is processed first when an IPC receive is performed.

Table 3.1: *Summary of the different kernel implementations evaluated.*

Chapter 4

Scheduler Evaluation Methods

To determine the strengths and weaknesses of each of the variants of the kernels described in Chapter 3, we constructed a series of benchmarks designed to exercise different aspects of the kernels. The benchmarks fall into two broad categories: *macro-benchmarks*, designed to exercise the system as a whole and be representative of workloads the system is likely to experience in real-world usage, and *micro-benchmarks*, designed to focus on a particular aspect of the system to evaluate performance of particular aspects of the kernels. These benchmarks are described in more detail below.

4.1 Testing Platform

4.1.1 Hardware Platform

We ran our tests on the Gumstix Connex 400xm board. The processor has an X-Scale PXA255 clocked at 400 MHz, with 64 MB of RAM.

The PXA255 has on-chip performance counters that allow processor events such as clock cycles, cache misses and TLB misses to be counted. We used the cycle counter in all our benchmarks to gain our timing results, except where noted otherwise.

4.1.2 Software Platforms

L4 Workbench

To perform micro-benchmarks, we constructed a custom framework named 'L4 Workbench' to allow benchmarks to be quickly constructed and tested.

The platform is carefully designed to ensure that no system activity takes place when a benchmark is executing other than the benchmark itself. The framework also runs benchmarks multiple times before timing results are taken to ensure that code and data pages associated with the benchmark are paged in and that the processor's caches are primed. Each test is then run five additional times, and the average result of each test is taken. The standard deviation of all test results was less than 1%.

Wombat and Re-aim

Wombat [6] is a port of Linux modified to run para-virtualised on top of L4. Wombat was constructed by adding a new L4 'architecture' to the Linux kernel. The port utilises L4 for thread creation and management, and uses L4 IPC for communication between the different components of the system such as syscall and exception delivery.

Re-aim [15] is a full-system benchmark designed to simulate a variety of workloads on POSIX-compliant operating systems. For our tests, we ran Re-aim on top of Wombat. We used the benchmark not only for its output benchmark results directly, but also as sample workload allowing us to gather statistics about how user-level processes interact with L4 (Section 4.2), and finally to act as a profile target to determine how much system time individual components of L4 consume (Section 4.5).

The Re-aim benchmark has two separate testing modes:

Single-user Mode : This benchmark mode runs a series of micro-benchmarks, each measuring the performance of a particular aspect of the system such as the number of UDP operations per second, the number of floating point operations per second, or the disk throughput of the system. We used the Re-aim single user benchmark to test system throughput for the different kernels. The different single user tests we used in our benchmarks are outlined in Table 4.1.

Multi-user Mode : This benchmark mode attempts to simulate workloads experienced by 'real world' systems. In our tests, we ran five processes that simultaneously execute a list of tasks in a pseudo-random order.

The tasks carried out by each thread are determined based on the benchmark 'profile' used. Each profile attempts to simulate a different workload, such as a database server, a file server or a web server. The three profiles we tested are described in Table 4.2. We used the Re-aim multiuser benchmarks to gather statistics on how userspace interacts with the kernel, and also to act as a profile target.

Benchmark	Task
`brk_test`	Carry out the `brk` syscall in a loop.
`creat_clo`	Create and then close files in a loop.
`dgram_pipe`	Send and receive random-length datagram packets.
`dir_rtns`	Carry out various directory querying syscalls.
`exec_test`	Create children with `fork`, which in turn carry out an `exec`.
`fork_test`	Create and wait for child processes using `fork` and `wait`.
`link_test`	Create and destroy hard links to individual files.
`misc_rtns`	Carry out miscellaneous Unix query syscalls.
`page_test`	Allocate and deallocate memory with `sbrk`.
`pipe_cpy`	Send and receive random-length packets over a Unix pipe.
`shared_memory`	Perform semaphore operations and read/write operations on shared memory.
`signal_test`	Send and catch Unix signals in a loop.
`stream_pipe`	Send and receive random amounts of data of a Unix stream.
`udp_test`	Send and receive random-length UDP packets over loopback.

Table 4.1: *Descriptions of the Re-aim single-user benchmark tasks tested.*

For both benchmarking modes, a number of the tests attempt to read or write to disk. The hardware we used for testing runs Wombat entirely from a RAM-disk, so no real I/O occurs.

In order to ensure that the results gained were reproducible, we modified the Re-aim benchmark source-code to seed its pseudo-random number generator on the child-number of each benchmark process instead of the Linux-generated process-id that Re-aim was configured to use.

Finally, in our initial trials of Re-aim we discovered a bug in Wombat that caused it to lose track of significant amounts of time under high system activity. This bug caused the Re-aim throughput results to become artificially inflated. Unable to fully determine the cause of the bug, we modified Re-aim to query the cycle-counter on the PXA255 to gain accurate timing results.

Benchmark	Task
`complete`	The complete suite of Re-aim tests, both CPU-bound and those with a high level of system activity.
`database`	A workload simulating that of a database server. Involves a some CPU-bound work, and a significant level of work with high system activity.
`udpserver`	A workload simulating a network server communicating over UDP, such as a DNS server.

Table 4.2: *The Re-aim multiuser system profiles tested.*

4.2 Kernel/Userspace Interaction Statistics

To gain an understanding of how userspace workloads interact with the kernel, we modified a version of the Direct/Lazy kernel to collect statistics about how the three different Wombat workloads listed in Table 4.2 interact with the kernel. We use these values to calculate theoretical performance qualities of the kernels described in Chapter 3, such as the number of queue operations saved by lazy-queueing, or the number of ready-queue lookups saved by direct process switching.

The following values are measured:

Eager Enqueues / Dequeues : The number of times a kernel that implements eager-scheduling would need to enqueue or dequeue a thread from the ready-queue.

Only one counter is used to record the number of enqueues and dequeues because the two values are the same: every thread that would be enqueued onto the ready-queue by an eager scheduler must eventually be dequeued again. (The one exception is that the two counters would differ from each other if there are threads on the ready-queue that have not yet been dequeued.) The final value of the counter is hence equal to the number of enqueues and also the number of dequeues.

The counter value is obtained by recording the number of transitions a thread makes from a ready state to a blocked state. A kernel implementing eager-queueing would be required to remove the thread from the ready-queue at this time, but a lazy-queueing kernel may avoid the operation completely.

Actual Enqueues / Dequeues : The number of times a thread is actually enqueued in the scheduling queue data structure by a kernel implementing lazy-queueing.

Similarly to the eager enqueue/dequeue counter, only one counter is used to record both actual enqueues and actual dequeues because the two values are expected to

be the same: the number of times a thread is placed on the ready-queue must be equal to the number of times a thread is removed from the ready-queue, disregarding changes in the number of threads currently on the ready-queue.

By inspecting the differences in the number of eager enqueues/dequeues and the number of actual enqueues/dequeues, the effectiveness of lazy-queueing in avoiding ready-queue operations can be determined.

Deferred Enqueues / Dequeues : The number of times an enqueue or dequeue operation is avoided by lazy-queueing, only to occur again at a later time.

Deferred dequeues occur when the scheduler is searching for a thread to run and stumbles across a thread not in a ready state, which must then be removed before continuing. Deferred enqueues occur when an executing thread is not on the scheduling queue and is interrupted by a second thread, sends to another thread, or has its timeslice expire. The original thread must then be placed on the ready-queue to ensure that the scheduler comes back to it at some point in the future.

Context Switches : The number of context switches from one user thread to another user thread, caused for example by IPC transferring control to a different thread or timeslice expiration.

Ready Queue Lookups : The number of times the ready-queue is searched to find the highest priority thread in the system ready to execute.

Ready Queue Length : The number of threads currently on the ready-queue, sampled each time a ready-queue operation (i.e., an enqueue, a dequeue or a lookup) takes place. The value sampled is the number of items on the queue *prior to* the queue operation taking place.

Total IPC Calls : The number of IPC messages successfully transferred from one thread to another.

IPCs Requiring Queueing : The number of IPC calls that require the sending thread to be placed on the receiver's IPC send queue prior to the IPC being carried out. This situation occurs if the destination thread is not currently ready to receive an IPC, for example if it is not listening for IPCs or if it is only ready to accept an IPC from a specific thread other than the sender.

4.3 Micro-Benchmarks

4.3.1 Ping Pong

A common benchmark used in the L4 community to measure the performance of IPC is the 'ping pong' benchmark. This benchmark attempts to determine the best-case number of cycles required for a thread to send a single IPC message with a fixed amount of data to another thread.

The benchmark is performed by having a client thread make an IPC call to a server thread. The server immediately replies back to the client, simultaneously becoming ready to receive the next IPC. We measured the number of cycles required to perform 1 000 000 such operations to determine the number of cycles required for the IPC operation.

4.3.2 Hot Potato

The 'hot potato' benchmark measures the time it takes for a ring of 250 threads to relay an IPC message (the 'potato') among them. Each thread is given a unique priority between 1 and 250 inclusive, and the order of threads in the ring shuffled.

When only one IPC is being passed between the threads, the hot potato benchmark is similar to the ping pong benchmark, with two exceptions: (i) the thread being sent to and the thread being received from are different, which exercises different areas of the IPC code; and (ii) IPC messages are sent to and from threads with different combinations of priorities, each combination producing a different scheduling outcome. This contrasts ping pong where each thread both sends to and receives from a thread of a fixed priority.

The hot potato benchmark also tests the time taken for the ring of threads to relay the IPC when multiple IPCs are active within the ring. When only one IPC is being relayed, only a single thread will ever be placed on the ready-queue at any point in time, while all others will blocked waiting for the IPC. When multiple IPCs are active in the ring, multiple threads will be ready to be scheduled at any point in time: one per active IPC. By increasing the number of active IPCs, we test how the ready-queue data structure responds to a greater number of threads being placed on it at any point in time.

The hot potato benchmark is a more realistic benchmark for determining average-case IPC times than the ping-pong benchmark as it exercises a greater amount of the IPC handling code of each kernel. The benchmark also uses a large number of threads, preventing the entire working set of the benchmark from fitting into the processor's cache, as would likely be the case for IPC usage in real-life systems.

4.3.3 IPC Queueing

The four kernels, Direct/Lazy, IPC-Scan, IPC-Ordered and IPC-Heap all use different methods of enqueueing and dequeueing threads on IPC send queues. This benchmark tests the performance of these methods as the number of threads on a single IPC queue increases.

The test executes by starting a number of threads with random priorities. The threads are each woken one by one in a random order. When a thread wakes, it calls the server, causing it to becomes queued on the server's IPC send queue. Only after all threads are enqueued does the server process the threads waiting on its send queue. This process is repeated and the average time is determined for each thread to send its IPC, be enqueued, and then be processed by the server.

The test allows us to measure how long it takes to enqueue and dequeue threads from IPC send queues. Because we expect the time taken to enqueue and dequeue to change depending on the number of other threads on the queue, we repeat the test for varying numbers of threads.

4.4 System Latency

We measured latency in the system by setting up a high-priority 'latency measurement' thread which periodically takes latency measurements while other system activity takes place. The thread utilises a 3.6864 MHz timer on the PXA255 to trigger an interrupt at a known time in the future, and then blocks waiting for the interrupt. When the thread is woken again, it records the current time and determines the latency by calculating the difference between the time the thread was scheduled to wake up and the time the thread was actually woken. The length of time in the future the latency thread schedules its wake-up interrupt is chosen randomly to avoid systematic errors in the latency samples.

To measure worst-case latencies introduced by the different scheduling algorithms, we constructed benchmarks designed to act as pathological testcases for each algorithm. While none of these pathological cases are intended in any way to be representative of expected workloads in L4, they are still relevant when considering hard real-time systems (which require guarantees about the upper-bounds on interrupt latencies, for example), when there is a possibility of malicious threads being present on the system (which may attempt to exploit these worst-case scenarios), or if the system designers are just feeling particularly pessimistic (and hence do not wish to assume that the scenarios described in these contrived benchmarks will not actually occur in their real-life workloads).

4.4.1 Lazy-Dequeueing Latencies

The use of lazy-dequeueing in a kernel has the potential to introduce large latencies into the kernel if a large number of blocked threads build up on the ready-queue and must later be removed in a single non-preemptable operation. These two tests attempt to highlight these problems by deliberately triggering these long chains of deferred dequeues.

IPC Elevator Test

A chain of 250 threads in increasing priority is formed, each thread initially blocked waiting for an IPC from the previous thread in the chain. Starting from the lowest priority thread, each thread, when woken, carries out the following sequence of actions:

- The thread performs a send IPC to a high-priority server currently ready to receive.
- Control flows to the server, which then performs a blocking receive, sending it to sleep again. The scheduler then returns control back to the original thread.
- The thread then performs a blocking call to the next thread in the chain, causing the next thread to wake and the current thread to become blocked.

This process is depicted in Figure 4.1. The first step in the sequence of actions forces each thread in the chain onto the ready-queue, while the third step causes each thread, now on the ready queue, to become blocked. If lazy-dequeueing is being used by a kernel implementation, the thread will not be removed from the ready-queue. When the end of the chain is reached, all 250 threads (now in a blocked state) will be on the ready-queue. The next time the kernel attempts to find a thread ready to be scheduled, all 250 threads will be removed from the queue, a long non-preemptable operation potentially causing large latencies. The process is then repeated.

Thread Start/Stop Test

An alternative way of showing the large potential latencies caused by lazy-dequeueing is through L4's suspend/resume thread functionality. When a thread is suspended, lazy-queueing kernels may not remove the suspended threads from the ready-queue.

This test sets up 250 intermediate-priority threads, each of which is then suspended by a high-priority controller thread. Once all threads have been suspended, the controller thread changes to a low priority, forcing the L4 scheduler to search for the next ready-ready thread. During this search, all 250 suspended threads are lazily dequeued from the

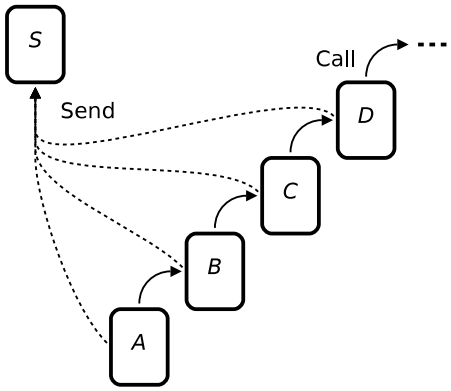


Figure 4.1: *The IPC elevator test. A series of threads in increasing priority each send to a high-priority server S, causing S to be scheduled and the thread placed on the ready-queue. Each thread, when next scheduled, then calls the next thread up in the chain, becoming blocked. The process repeats, forcing a large number of blocked threads onto the ready-queue, which will all have to be dequeued the next time a ready-queue lookup occurs.*

ready-queue in a non-preemptable operation. L4 will eventually reschedule the control thread, which will then revert to a high priority, resume the 250 threads, and repeat the process.

4.4.2 Chained IPC Latencies

Another source of high latencies in the L4 kernel independent of the scheduler optimisations presented in Chapter 2 are *chained* IPCs. Chained IPCs involve a sequence of successive IPCs that are carried out one after the other in a single non-preemptable operation.

Such chains may form if multiple threads perform a 'send and receive' IPC to each other. None of the IPC operations can take place until the first thread in the chain becomes ready to receive. When this happens it is possible, if the priorities of threads in the chain are in increasing order, that the entire chain of IPCs will be carried out in a single non-preemptable operation, potentially introducing high latencies. This benchmark sets up 250 threads in such a chain, releases the chain, and then repeats the process in a loop.

We examine this source of latency in order that we may be able to compare the latency caused by lazy-queueing with it. This will assist in gaining an understanding of the relative magnitude of the lazy-queueing latencies.

4.5 Scheduler Overhead

The final metric we tested was the amount of processor time consumed by the scheduler, as a percentage of the total system time. We would expect that the performance of each of the kernels would be inversely proportional to the amount of time taken by the scheduler to make its decisions. Profiling can help reveal the cause of performance bottlenecks, and also acts as a method to double-check the results measured in previous benchmarks (for instance, by ensuring that unexpected idle time does not occur).

To determine the amount of time taken by the scheduler, we profiled a system running the Re-aim `udpserver` multiuser benchmark for each kernel.

4.5.1 Statistical Profiling

One light-weight method of profiling a complete system is *statistical profiling*. This profiling method periodically takes a sample of the instruction the processor is currently executing. The collected samples are analysed off-line, matching each sample with its corresponding function. Over long periods of sampling, functions in the system will be represented by approximately the same proportion of samples as the processor time they consume.

Statistical profiling has the benefit that every component of the system—both components running in the processor's privileged mode and those in standard mode—can be simultaneously profiled. Additionally, statistical profiling can take place without needing to recompile any part of the system, and is capable of profiling any type of code regardless of the language it was written in (whether that be C, C++, or hand-crafted assembler).

Statistical methods of profiling do, however, suffer from a few flaws. They rely on random sampling, and as such are not appropriate for profiling code that runs a small, constant number of times. Such code will not be sampled a sufficient number of times to confidently determine what percentage of time each of the functions are taking.

Further, the method of statistical sampling used assumes that a program counter value can be mapped uniquely to a single function. Because program counter samples contain virtual addresses (which may be shared by multiple different processes), this assumption may not always hold.

For the Re-aim benchmarks, L4, Iguana and Wombat each occupy a different region of the system's virtual address space, allowing program counter samples to be correctly

attributed to each. The user applications running on top of Wombat (such as *init*, the system's shell and Re-aim itself) share virtual addresses, as depicted in Figure 4.2.

For our profiling results, we make every effort to ensure that the minimal number of Wombat processes are running during the profiling process, and assume that those that are running will be relatively inactive, and thus will only be sampled a negligible number of times. With this assumption, we attribute all samples in the user-level virtual address area to the single application being profiled.

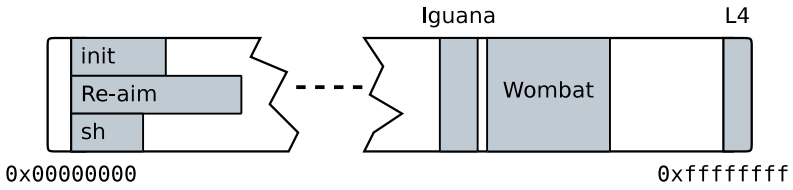


Figure 4.2: *Address space layout of a system running Re-aim on Wombat. L4, Iguana and Wombat each occupy unique locations in the virtual address space, allowing program counter samples to be uniquely mapped to compiled functions. User-space applications running on Wombat, however, share the same set of virtual addresses making this mapping difficult.*

4.5.2 Sampling the program counter on the ARM

The basic idea of gathering program counter samples is fairly straight-forward: the profiler sets up an interrupt to periodically fire. When the interrupt fires, the profiler is trapped into. The profiler then records the address that was executing when the interrupt fired and resumes execution of the system.

The PXA255 chipset has an on-chip read/write 32-bit cycle counter, which also has the ability to fire an interrupt when the counter overflows. By setting the value of the counter to $2^{32}-t$, we can choose to fire an interrupt t cycles in the future. For our tests we sampled the processor's program counter at a random time within every 25 µs block, as shown in Figure 4.3.

In our experiments we ran multiple profiling runs on the same workloads taking a sample once every 25 µs in half the runs and once every 1600 µs in the other half. We were unable to detect any significant differences in profiling results of the two runs. This suggests that the overhead of sampling once every 25 µs is not significant enough to modify the profiling results.

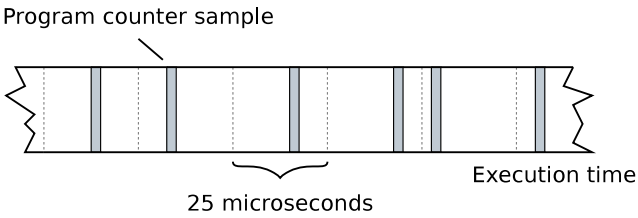


Figure 4.3: *Random sampling of the processor's program counter. Time is divided into even length blocks. An interrupt is scheduled to fire at a random point of time in each block, where the program counter is recorded for later off-line analysis.*

Sampling kernel code

A significant difficulty in profiling code is that while the L4 microkernel is executing, interrupts are disabled. The effect of this is that any timer-based interrupt would be delayed until after interrupts were re-enabled (causing a significant number of samples to be captured on the 're-enable interrupt' instruction, making it appear as if that instruction itself was taking the majority of the processor's time).

The ARM processor has two methods of generating an interrupt, a standard interrupt and a 'fast' interrupt. Each interrupt type has a different method of trapping into the kernel. Further, when interrupts are disabled, the kernel is capable of leaving fast interrupts enabled.

We modified the kernel to leave fast-interrupts enabled while in kernel mode, and designed the profiler to use fast interrupts as its method of taking a sample. This allows the profiler to be able to take samples at almost all times.

There are still a few periods of time that the profiler is unable to sample the system: when the kernel page faults, for example, even fast interrupts are disabled until the fault is resolved. From the profiler's point of view, it appears that faulting instructions take many hundreds of cycles to execute, and hence these instructions are more likely to be sampled by the profiler.

4.5.3 Analysing Results

To analyse the profiling data, we group program counter samples into their individual functions. This is done by dumping the symbols from compiled executables which gives the compiled-code addresses of each source-level function. Functions that have been

compiled from C and C++ have both start and end locations recorded in their executable files, making program counter samples easy to account for. Functions written in assembly (such as `memcopy` and `memmove`) only have start symbols listed for them, making it harder to determine if a given program counter sample belongs to them. We use a heuristic that assigns program counter samples to an assembly function if the previous symbol known in the address space was the start of an assembly function and the starting location is not more than a small constant number of bytes earlier than the sample.

Dynamic libraries are trickier to map samples to, as the libraries may be located at different virtual addresses for each program execution. We could not find a reliable method of determining where a given library was located in memory that did not involve making modifications to either Wombat itself or the applications running under Wombat. As a consequence, our implementation does not track program time spent in dynamic libraries, but instead marks it 'unknown'.

4.5.4 System cache profiling

The system profiler is configured to use the processor's clock cycle performance counter as a trigger for when to take a sample: when a certain number of cycles have occurred (representing the passage of time), a sample is taken. These samples tell us where the most time is being spent.

The PXA255 offers a variety of different performance counters, all of which can be set up to trigger an interrupt at overflow. If instead of sampling after a number of clock cycles we take a sample after a specific number of data-cache misses, we can create a very rudimentary system cache-usage analyser. This same technique can be used on any of the other performance counters offered by the processor: TLB misses, branch mispredictions, data stalls, etc.

Due to time limitations, we were unable to collect data relating to the scheduler's contribution to cache misses, but will consider it in future work.

4.6 Summary

We took four different approaches to evaluate the different kernels described in Chapter 3: (i) collecting statistics regarding how the Re-aim multiuser benchmarks interact with the kernel scheduler and IPC paths; (ii) performing micro-benchmarks on L4 Workbench, each testing a particular aspect of system performance for each modified kernel; (iii) using the

Re-aim single user benchmarks to determine the performance of a variety of different tasks for each modified kernel; and finally (iv) profiling each modified kernel to determine the amount of time spent in the scheduler and IPC paths.

The results of these tests are described in the next chapter.

Chapter 5

Results

In this chapter, we explore the results of the benchmarks described in Chapter 4. We start by examining statistics showing how userspace workloads interact with the kernel and calculating the theoretical overheads associated with the scheduler optimisations and IPC. We then move on to looking at empirical results and compare them against our theoretical results.

5.1 Kernel/Userspace Interaction Statistics

5.1.1 Scheduler Usage

Scheduler Operations

The results in Table 5.1 show the number of different scheduler operations performed per second for three Re-aim workloads described in Section 4.1.2. Comparing the number of eager enqueues/dequeues to the number of actual enqueues/dequeues, we can see that the number of queue operations avoided by lazy-queueing is significant: between 96% and 99% of operations need not be carried out, depending on the workload.

Of the enqueue and dequeue operations actually carried out, a large fraction occur at a deferred time; roughly one-third of enqueues and two-thirds of dequeues. These results suggest that it is important for lazy-queueing kernels to ensure that deferred queue operations are able to be performed quickly, as all kernels explored in this thesis currently do.

The results also show that the number of scheduler lookups is significantly less than the number of context switches. If no optimisations are applied, we would expect that a priority-observing kernel implementation would be required to perform one ready-queue

Operation	complete	database	udpserver
Eager enqueues/dequeues	6757.60	10187.48	41605.22
Actual enqueues/dequeues	280.96	233.79	213.45
Enqueues/dequeues avoided	95.84%	97.71%	99.49%
Deferred enqueues	88.88	83.27	99.02
(percent of actual)	31.63%	35.62%	46.39%
Deferred dequeues	181.85	140.88	128.81
(percent of actual)	64.73%	60.26%	60.35%
Context switches	7040.25	10464.57	41905.92
Ready-queue lookups	363.94	308.16	305.38
Lookups avoided	94.83%	97.06%	99.27%

Table 5.1: *Counts of scheduling operations required by three different Re-aim workloads. All values are operations per second.*

lookup before each context switch to determine the next thread to switch to. The direct process switch optimisation saves a large fraction of the lookups required, with between 95% and 99% of the lookups being avoided, depending on the workload.

Ready Queue Usage

Figure 5.1 shows the number of threads present on the ready-queue prior to each queue operation taking place (i.e., prior to each enqueue, dequeue or queue-lookup taking place). The results show that for these workloads, the majority of queue operations take place with only two or three threads actually being present on the ready-queue. Further, a maximum of six threads are ever simultaneously on the ready-queue for these particular workloads.

These results are not unexpected. The tested Re-aim workloads only run five processes at any point in time. The worst-case scenario would correspond to all five processes being ready to execute at the same time as the Wombat syscall thread.

Lazy-dequeueing is likely to inflate the number of ready threads on the ready-queue, as blocked threads remain on the queue for longer than necessary. Similarly, lazy-enqueueing may artificially reduce the number of threads on the ready-queue, but only by at most a single thread, unlike lazy-dequeueing which may cause an arbitrary increase.

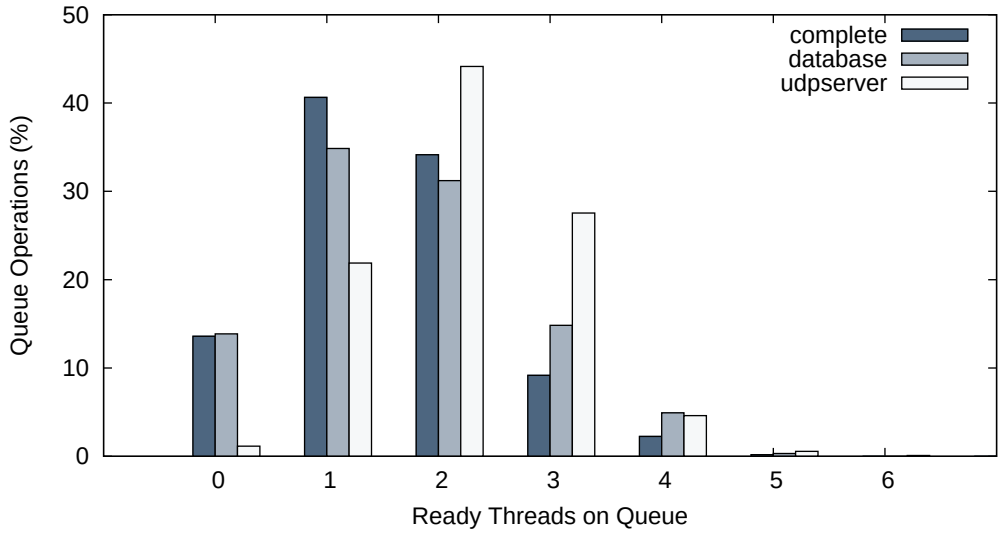


Figure 5.1: *Number of threads enqueued on the ready-queue for Re-aim workloads as a percentage of queue operations. We can observe that for the majority of queue operations, only two or three threads are actually present on the ready-queue.*

Because these results are sensitive to the number of threads running in the system (which was limited to only five for the Re-aim tests), it is hard to draw any generalised conclusions. Future work that carries out benchmarks utilising a greater number of threads may provide greater insight to how the ready-queue is utilised by system workloads.

5.1.2 IPC Usage

All three workloads have a significant level of IPC activity, ranging from thousands to tens-of-thousands of IPCs per second. The `complete` and `database` workloads tend to

Operation	`complete`	`database`	`udpserver`
IPC calls	6494.19	9979.19	41402.03
Average cycles between successive IPC calls	61244	39733	9311
Expected IPC overhead (350-cycle IPC)	0.57%	0.87%	3.62%
IPCs requiring delivery queueing	0.64	0.89	0.56
Maximum IPC send-queue length	1	1	1

Table 5.2: *Counts of IPC operations required by three different Re-aim workloads. All values are operations per second.*

have a lower number of IPCs, as both workloads involve tasks that are primarily CPU bound. In contrast, the `udpserver` workload primarily involves communication back and forth to Wombat, so we would expect a higher level of IPC activity.

We calculated the approximate number of cycles of work performed between successive IPCs, using the fact that the PXA255 has a clock speed of 400MHz and assuming that the instrumented kernel takes approximately 350 cycles to perform each IPC.

Both the `complete` and `database` workloads have a significant number of cycles between each IPC. Using the calculations shown in Figure 2.1 on page 10, we would expect these workloads to have approximately 0.57% to 0.87% of their time attributable to IPC overheads. The `udpserver` workload has a much greater rate of IPCs, averaging approximately one IPC each 9000 cycles of work performed. With this high level of IPC activity, we would expect system time attributable to IPC to be approximately 3.62%.

For all three workloads, the number of IPCs that are placed into their sender's IPC send queue is very small: less than one per second. Further, for each of these queueing operations, the enqueued thread is the only thread in the receiver's send-queue. As IPC send-queueing is such a rare operation, we would not expect any measurable difference for the IPC-Scan, IPC-Ordered or IPC-Heap kernel implementations. This allows a guarantee of thread ordering to be provided by the kernel with minimal impact on these workloads.

5.2 Micro-Benchmarks

5.2.1 Ping-Pong

Figure 5.2 shows the number of cycles taken for the different kernel implementations to perform a warm-cache IPC operation. All standard deviations for the results were less than 0.1%.

The Fastpath implementation is by far the fastest IPC implementation, taking 173 cycles to perform an IPC, which is 121 cycles faster than the C-based Slowpath implementation using the same algorithms, taking 294 cycles (70% slower).

The Direct/Lazy kernel, mimicking the behaviour of the Slowpath kernel, has an almost identical speed, taking 297 cycles. Thus the introduction of the abstract scheduling interface has a minimal impact on warm-cache IPC times between two threads.

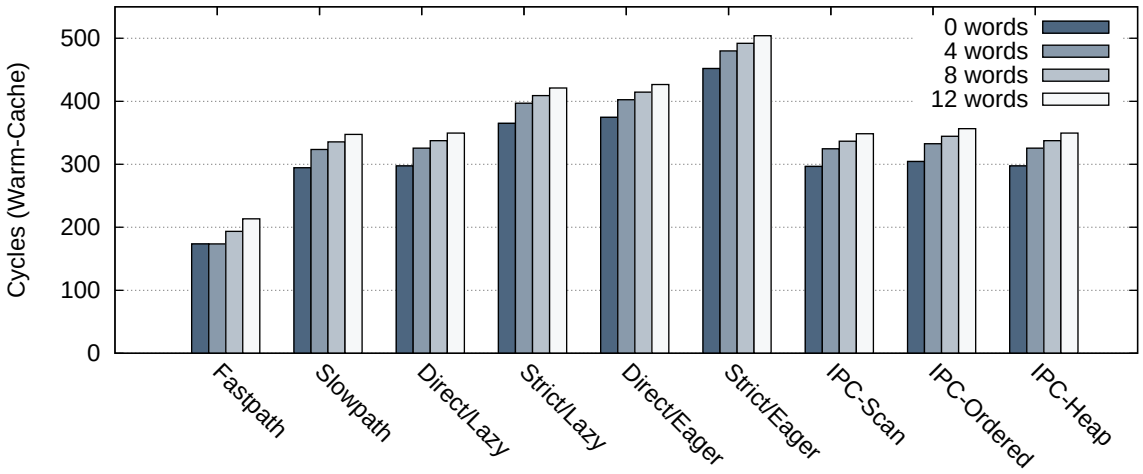


Figure 5.2: *Warm-cache ping-pong times for the different kernel implementations. Times are shown for different numbers of 32-bit words being transferred in the IPC call.*

The Direct/Lazy, Strict/Lazy, Direct/Eager and Strict/Eager kernel implementations take 297, 365, 374, and 452 cycles respectively. For raw IPC times, the direct process switch optimisation saves 68 cycles while lazy-queueing saves 77 cycles. Removing these optimisations would respectively cause a 23% and 26% time increase to warm-cache IPC times over the base Direct/Lazy IPC times. Despite over 13 years of development having taken place on the L4 kernel and a change of architecture from IA32 to ARM, the former result closely matches Liedtke's [8] results, which also determined that removing lazy-queueing would increase IPC times by 23%.

Comparing the Direct/Lazy, IPC-Scan, IPC-Ordered and IPC-Heap kernels we see that all four have very similar raw IPC times, taking 297, 296, 304 and 297 cycles respectively. No change should be expected between the four kernels, as the ping-pong benchmark does not require any IPC queue manipulations to take place. The small differences between the four kernels can be attributed to small changes in the compiler's code generation.

5.2.2 Hot Potato

Figure 5.3 shows the results of the hot-potato benchmark, described in Section 4.3.2. The x-axis shows the number of IPC messages (or 'potatoes') active in the ring, while the y-axis shows the average time taken for each thread to process a single IPC message and pass it on to the next thread. The number of cycles required for each IPC operation

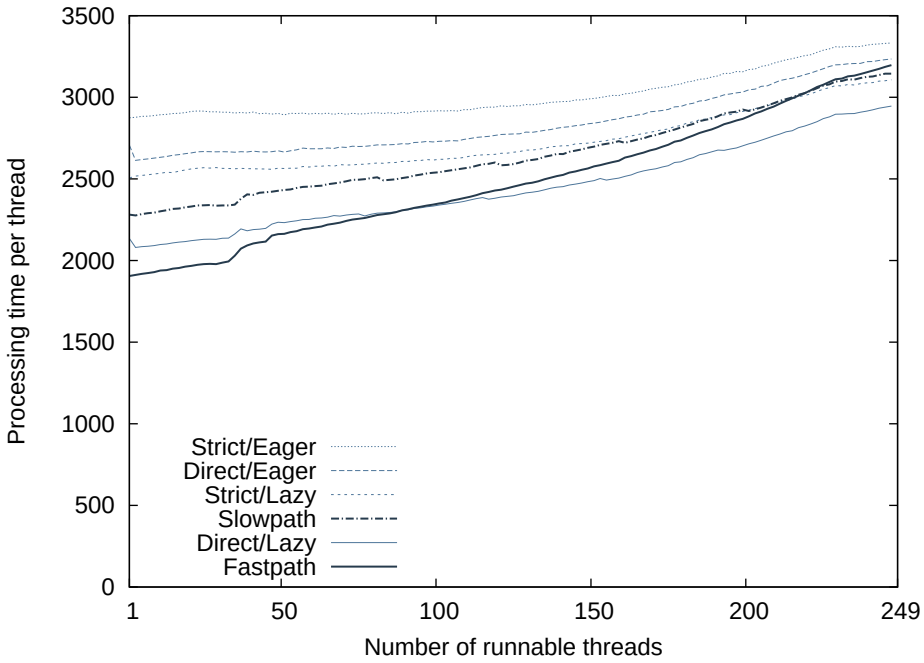


Figure 5.3: *Results of the 'hot potato' benchmark, described in Section 5.2.2. The x-axis shows the number of IPC messages active in the ring, while the y-axis shows the time taken for each thread to see a constant number of IPC messages.*

is significantly higher than in the ping-pong benchmark because the working set of the benchmark is too large to fit entirely in the processor's cache.

When the number of potatoes is small, the Fastpath kernel is significantly faster than the other kernel implementations, benefiting from its assembly-optimised IPC fastpath. As the number of potatoes increases, however, the performance of of the kernel slowly deteriorates, eventually exceeding the speed of the Slowpath kernel by a small margin. As the number of active IPCs in the ring approaches the number of threads, threads increasingly are forced to wait for their destination to be ready before they can send. Such waiting will require the source thread to be placed on the destination thread's IPC send-queue, and later removed when the IPC finally takes place. Neither of these operations are supported by the assembly-optimised IPC path of Fastpath, forcing it to to use the same C-language IPC path as the Slowpath kernel implementation.

For large numbers of active IPCs, the Fastpath kernel eventually becomes slower than the Slowpath kernel, because the fastpath offers no benefits to the kernel for these testcases but is still invoked each IPC forcing the kernel to perform extra checks only for the IPC to be demoted to the slowpath.

The Slowpath kernel and the Direct/Lazy kernel both implement the same scheduling algo-

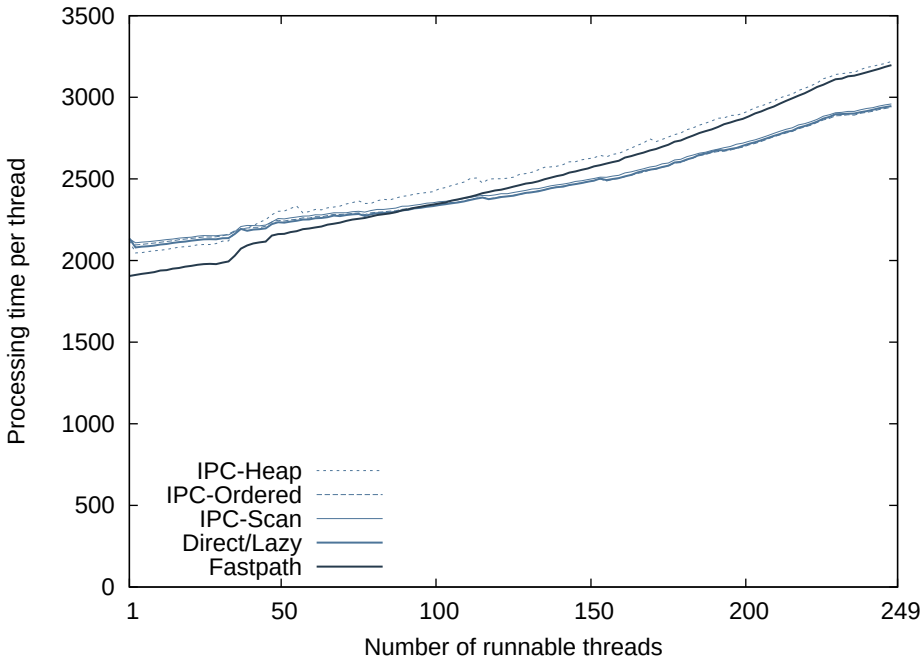


Figure 5.4: *Results of the 'hot potato' benchmark comparing the* IPC-Scan*,* IPC-Ordered *and* IPC-Heap *kernels against the* Direct/Lazy *kernel. The* Fastpath *kernel is also shown as a baseline.*

rithm and are both implemented in the same language. All else being equal, we would expect a similar performance curve for the two kernels. The Direct/Lazy kernel, however, is around 10% faster for this benchmark. The difference in performance can be attributed to a minor implementation difference in the lazy-queueing optimisation of the two kernels: the Slowpath kernel eagerly dequeues threads from the scheduler's ready-queue when a thread carries out a blocking receive, unless the thread being received from is the same as the thread being sent to. This latter exception is why the performance difference is not visible in the ping-pong benchmark results. In contrast, the Direct/Lazy kernel (like the Fastpath kernel's IPC path) lazily-dequeues the threads.

The ordering of the Direct/Lazy, Strict/Lazy, Direct/Eager and Strict/Eager kernels are similar to the warm-cache IPC times in Section 5.2.1. The notable exception is that the performance difference between Strict/Lazy and Direct/Eager kernels is magnified from the ping-pong results, the former kernel having an 4% speed advantage over the latter.

Comparing the three IPC strict queue ordering kernels shown in Figure 5.4, both the IPC-Scan and IPC-Ordered kernels have very similar performance to that of the Direct/Lazy kernel. The hot-potato benchmark does not require IPC queues to ever hold more than a single thread at any point in time; this means that any disadvantages that the two kernels with $O(n)$ send-queue operations have dealing with a larger number of threads do

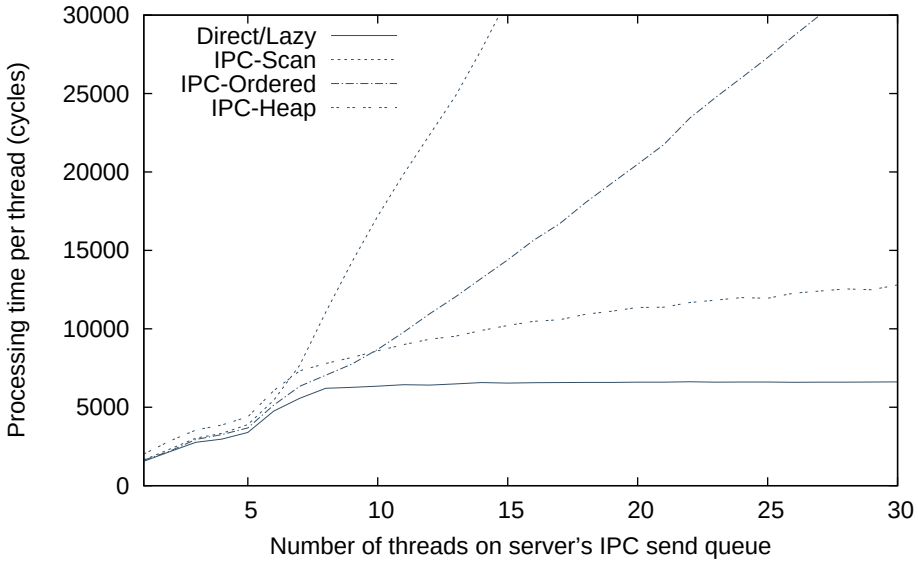


Figure 5.5: *IPC queueing benchmark results. The x-axis shows the number of threads enqueued onto a single server thread's IPC send-queue before the server starts processing the threads; the y-axis shows the amount of time taken for each thread to be processed.*

not apply for this benchmark. The IPC-Heap kernel, meanwhile, has slower performance than the other three kernels. While this kernel only has to ever enqueue a single thread on any one IPC send-queue like the others, the heap data structure used by the kernel has more complex enqueue and dequeue operations, causing the performance decrease seen in the results as the number of queued IPCs increases.

5.2.3 IPC Queueing

The results in Figure 5.5 show the IPC queueing benchmark described in Section 4.3.3 executed for Direct/Lazy, IPC-Scan, IPC-Ordered, and IPC-Heap. The Direct/Lazy implementation makes no attempt to process threads in priority order, but can be considered as a baseline for an "ideal" implementation.

The IPC-Scan and IPC-Ordered kernel implementations require $O(n)$ time to process each thread, as would be expected of the implementation. When a larger number of threads are present on the IPC send-queues, the IPC-Ordered kernel tends to be twice as fast as the IPC-Scan kernel. This occurs because when threads are inserted into a sorted list, on average only half the list needs to be searched before the correct location can be found. In contrast, the IPC-Scan algorithm requires the *entire* list to be scanned every time a thread is taken off the queue.

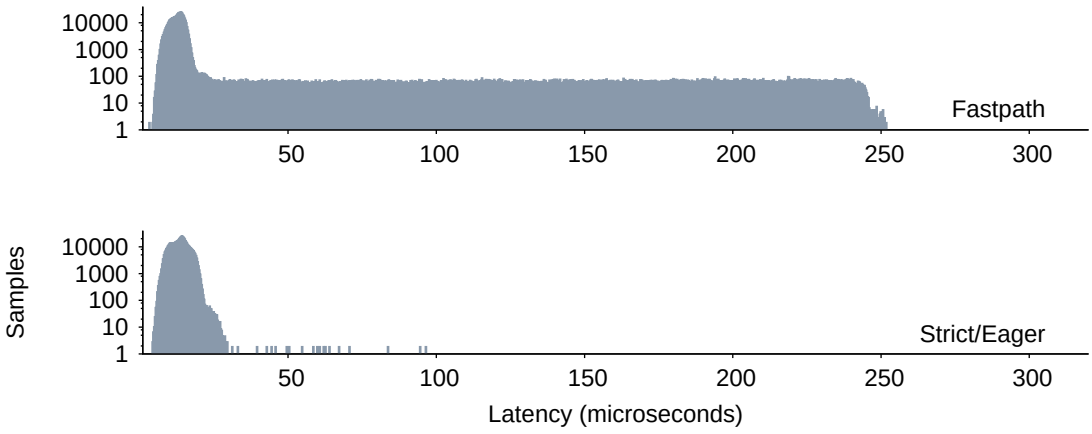


Figure 5.6: *Latency measures of a pathological case for lazy-queueing. 250 threads are placed on the ready-queue which then move into a blocked state. A kernel implementing lazy-queueing may have to dequeue all 250 blocked threads before a ready thread can finally be found, introducing potentially large latencies into the system.*

The IPC-Heap kernel performs significantly better than both IPC-Scan and IPC-Ordered once the number of threads becomes larger, as we would expect of the $O(\log n)$ algorithm. This comes at the cost of slower performance than the other two implementations when the number of threads is smaller than ten. The operations involved in maintaining a heap data structure require more work than those for maintaining a sorted list or scanning a list, causing the scalability offered by the IPC-Heap kernel to come at the cost of higher overhead for each queued IPC operation. It is likely, however, that these overheads could be reduced if additional effort was spent optimising the heap implementation of this kernel.

5.3 System Latency

5.3.1 Lazy-Dequeueing Latencies

The two lazy-dequeueing latency tests show the effect of lazy dequeueing on worst-case latency times. Both latency benchmarks attempt to force the scheduler to place 250 different threads on the ready-queue and then lazily dequeue them all. A lazy-queueing kernel that leaves these threads on the ready-queue will possibly require that all 250 threads be removed from the ready-queue in a single non-preemptable operation.

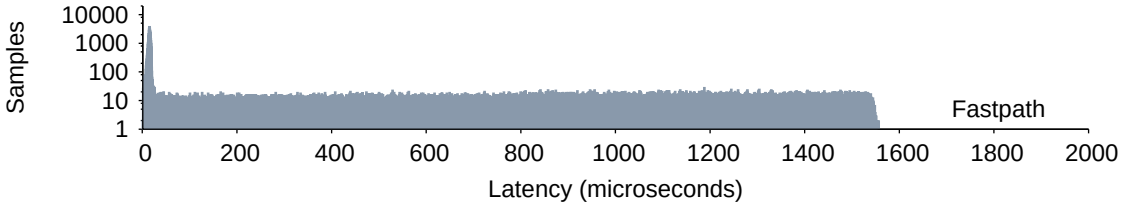


Figure 5.7: *Latency measures of a pathological case for IPC chaining, described in Section 4.4.2 on page 41. Full results are shown in Figure A.3 of Appendix A.*

By inspecting the assembly of the compiled Slowpath kernel, we expect a deferred dequeue to take approximately 20 clock cycles, with four data-cache lines accessed, one of which is shared with eight other threads. The PXA255, running at 400 MHz, would then be expected to take approximately 50 ns for each deferred dequeue (12.5 µs for all 250 threads) assuming warm caches, or 893 ns (223 µs for all 250 threads) if caches are cold, assuming a 108-cycle cache-miss penalty (measured for the PXA255) and that the penalty for the one shared cache line is only taken by one eighth of the threads.

Figure 5.6 shows the lazy-dequeueing benchmark latency results for two kernels. The lazy-queueing Fastpath kernel takes up to 250 µs for a significant proportion of samples taken. In contrast, the eager-queueing Strict/Eager kernel has the majority of its latency samples less than 30 µs. We were unable to determine the precise cause of the outliers for the Strict/Eager kernel, but may be due to unexpected cache-misses.

The full results in Figure A.1 of Appendix A show that all eager-queueing kernels have a similar profile to the Strict/Eager kernel, while almost all lazy-queueing kernels suffer the same level of latencies as the Fastpath kernel. The Slowpath kernel, however, does *not* suffer the high average latencies of the other kernels, despite employing lazy-queueing. This is because the Slowpath kernel eagerly-dequeues threads that block on IPC, as described earlier in Section 5.2.2.

Figure A.2 of Appendix A shows the results of the second lazy-queueing latency test, and shows that the Slowpath kernel still has the same latencies in pathological cases. The two kernels Direct/Lazy and Strict/Lazy, though using lazy-queueing, do not show the high latencies in this case as they both eagerly dequeue halted threads.

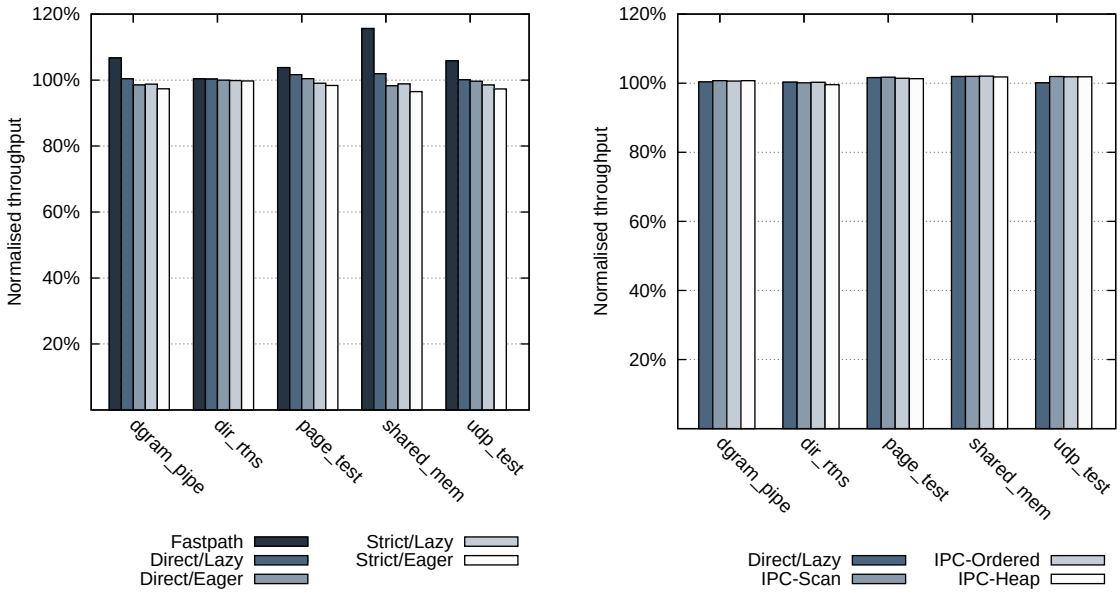


Figure 5.8: *Partial results of the Re-aim single-user benchmark results, normalised to the performance of the* Slowpath *kernel. The full results are presented in Figure A.4.*

5.3.2 IPC Chaining

Figure 5.7 shows the results of the IPC chaining latency test for the Fastpath kernel. The test shows that latencies of almost 1.6 ms can be triggered by userspace applications. All the kernels we tested showed similar results, which are shown in full detail in Figure A.3 of Appendix A.

The latencies arise because the kernel must complete all 250 IPC transfers in a single non-preemptable operation. The use of strict scheduling does not assist to reduce the latencies because the high-priority interrupt handling thread can not be placed onto the scheduler's ready-queue (and hence considered for scheduling) until the entire chain of operations is complete.

One method of reducing these latencies to lower levels would be to introduce a preemption point after each IPC transfer. Such a preemption point would check for pending interrupts each time an IPC transfer was completed and, if appropriate, would interrupt the chain of IPCs to allow the interrupt handler to be scheduled.

5.4 System Throughput

Figure 5.8 shows partial results of the Re-aim single user benchmarks for the different kernel implementations. (The full results are presented in Appendix A.) The clearest observation is that for almost all of the benchmarks the Fastpath kernel has the highest throughput, with a greatest improvement of 13% for the '`shared mem`' benchmark compared to the Slowpath kernel. This increase is due to the high number of IPCs performed in the benchmark, which continuously obtain and release semaphores, each requiring a call to the kernel. The high-speed IPC path in the Fastpath kernel benefits most by this.

Other benchmarks, such as the '`dir rtns`' benchmark, show little difference between the various kernel implementations due to very few IPC calls between threads in the system being required by the benchmarks. Operations such as interrupts and timeslice expirations still require the scheduler to be invoked regardless of what activity the benchmark is performing, but these occur so infrequently (only hundreds of times per second) that any differences among the kernel implementations are insignificant.

The kernels with C-based IPC paths show less significant differences for the different benchmarks, with all kernels performing within 5% of the Slowpath kernel.

5.5 Scheduler Overhead

We ran a full system profile on different kernels running the Re-aim multiuser `udpserver` benchmark. Table 5.3 shows a breakdown of the most time-consuming functions during the benchmark running on the Fastpath kernel, and Table B.3 shows the total amount of time spent by each of the different modules in the system for the same benchmark run.

The first observation is the large amount of time (15%) spent in the '`sys_cache_control`' function of L4. The function is used primarily for flushing ranges of memory from the processor's cache. After a brief investigation we were unable to determine the reason why such a large amount of time is spent in the function, but further investigation would certainly be warranted.

Table 5.4 shows the total amount of time spent in the scheduler and IPC paths by each kernel for the `udpserver` benchmark. The aggressive compiler inlining used in all the kernels means that much of the time that should be attributed to the scheduler is falsely attributed to IPC. For this reason, the split in time between the IPC and scheduler columns of the table is, at best, a rough approximation. The split still allows us to see that the two eager-queueing kernels, Direct/Eager and Strict/Eager, spend less time in the

Module	Function	Time Spent (%)
vmlinux	memmove	30.08
l4kernel	sys_cache_control	15.10
vmlinux	syscall_loop	3.38
l4kernel	vector_arm_swi_syscall	2.45
vmlinux	ip_push_pending_frames	1.75
l4kernel	vector_arm_swi_exception	1.64
reaim	read_write_close	1.63
vmlinux	udp_sendmsg	1.46
vmlinux	ip_append_data	1.37
vmlinux	flush_range_invalidate_kernel	0.97
l4kernel	kip_code	0.91
vmlinux	local_bh_enable	0.90
vmlinux	netif_rx	0.89

Table 5.3: *Functions using the most amount of processor time running the Re-aim* udpserver *multiuser benchmark, running on the* Fastpath *kernel. A more detailed profile is listed in Appendix B.*

scheduler than the two equivalent kernels with lazy-queueing, Direct/Lazy and Strict/Lazy. This occurs as the two eager kernels need not perform deferred-dequeues when searching for the next priority thread, instead paying a higher overhead in the IPC path. The Slowpath kernel similarly has a relatively low scheduler overhead. This is likely to be because of its use of eager dequeueing in the IPC path when threads become blocked waiting for IPC, giving it the same advantage as the other eager-queueing kernels at the cost of IPC slowdown times.

The IPC overheads listed are greater than estimated in Section 5.1.2. The Slowpath kernel which takes only 294 cycles for a basic IPC would be expected to have significantly less than 3.62% overhead, calculated for a hypothetical kernel with a 350-cycle IPC cost. The difference is likely to be because the cycles calculated in the ping-pong benchmark assume best case conditions: only two threads, sending small numbers of message registers, with all caches warm. By taking the system overheads shown in Table 5.4 and the IPC statistics in Table 5.2, we can estimate that an average IPC for the Slowpath kernel actually takes 561 cycles, almost twice the basic ping-pong time recorded.

	Time (%)		
Kernel	**IPC**	**Scheduler**	**Combined**
Fastpath	2.80	0.90	3.70
Slowpath	5.81	0.30	6.11
Direct/Lazy	5.68	0.60	6.28
Strict/Lazy	6.32	1.00	7.32
Direct/Eager	7.10	0.25	7.34
Strict/Eager	7.14	0.59	7.72

Table 5.4: *IPC and scheduler overhead for the different kernel implementations, running the Re-aim multiuser* `udpserver` *benchmark.*

Chapter 6

Conclusion

The goal of this thesis was to quantify the advantages and disadvantages of the three scheduling optimisations used in L4: direct process switch, lazy-queueing, and FIFO IPC send ordering. In order to carry out these experiments, we also constructed an in-kernel scheduling API allowing scheduling policies in the L4 kernel to be readily modified without significant changes to other parts of the kernel being required. Each of these goals are discussed below.

6.1 Kernel Optimisations

6.1.1 Direct Process Switch

The direct process switch optimisation brings clear performance advantages for all the benchmarks we tested due to the decreased overhead of IPC. As workloads increase their rates of IPC, these benefits will become increasingly significant in the level of throughput the kernel is able to provide. Workloads with more modest rates of IPC are less likely to see the performance improvements, however.

The optimisation comes at the cost of predictability, though. No guarantees about priority observance can be offered, making the construction of real-time systems difficult to impossible. Real-time systems that utilise a lower rate of IPCs would be best served disabling the optimisation for only a small increase in IPC overhead.

6.1.2 Lazy Queueing

The lazy-queueing optimisations also bring measurable performance increase, but cause the worst-case scheduling latencies present in the system to increase, as highlighted in Section 5.3.1.

These worst-case latencies are directly proportional to the number of threads active in the system, however. If this number is known at system construction time, these latencies can be accounted for.

Further, in the current implementation of L4, latencies arising from other sources in L4 such as those from IPC chaining shown in Section 5.3.2 are significantly worse than those caused by lazy queueing, and would have to be accounted for regardless of whether lazy-queueing was used.

While lazy-queueing introduces measurable latencies into the system, these predictable latencies are unlikely to be an immediate cause for concern for system builders until other (more serious) latencies in L4 are fully addressed.

6.2 FIFO IPC Queueing

For the workloads we tested, threads were rarely ever placed on an IPC send queue, and those that were placed on a queue were always alone. This means that the theoretical problems of FIFO IPC queueing discussed in Section 2.4.4 are rarely a concern in real-life systems.

If guarantees *are* required regarding thread ordering, changes can be made to the L4 kernel to implement priority-based IPC queueing with little effect on the performance of existing workloads. While simple algorithms such as using an ordered queue are fastest for IPC queueing in the workloads we looked at, using a more scalable data structure such as the heap implementation in IPC-Heap would ensure that IPC times do not get out of hand in pathological cases.

6.3 Kernel Scheduling API

The introduction of the kernel scheduling API simplified scheduling code throughout the kernel, and allowed different schedulers to be dropped into L4 with relative ease.

Rewriting the L4 scheduler to remove the direct process switch optimisation required only 4 engineering hours for an unoptimised yet fully functioning implementation.

The scheduling API was also implemented with almost no overhead: the performance differences between the Slowpath and Direct/Lazy kernel implementations are negligible.

The differences between the Fastpath and the Direct/Lazy kernel are more significant, though. The assembly-optimised IPC fastpath offers clear performance advantages that can not be overlooked, but is unable to take any advantage of the scheduling API. This does not prevent the use of the scheduling API in the rest of the kernel, however, nor reduce its usefulness. A custom IPC fastpath must still be constructed each time a scheduling policy changes or a new optimisation is used, regardless of whether the scheduling API is used or not.

6.4 Future Work

While we attempted to gain an understanding of the trade-offs associated with best-effort scheduling optimisations, more areas of work remain to be investigated.

The kernel implementations we investigated used both scheduler implementations and IPC paths coded in C, in contrast to the commercially deployed Fastpath kernel which uses an assembly-based IPC path. The relative performance of each of our kernels have the potential to change if all are coded in optimised assembly.

All of our kernels used the same ready-queue data-structure described in Section 2.4.1 on page 13. It would be insightful to investigate the effects of using alternate data structures for the ready-queue, such as a data structure that provides a O(1) highest priority thread lookup, perhaps with slower enqueues. The correct choice of data structure may assist in reducing scheduling overhead even further.

Finally, all our macro-benchmarks utilised the Re-aim benchmark running on Wombat. Ideally, a wider range of system workloads should be instrumented and benchmarked with the different kernels. This would allow our results to be more generalised, giving system implementers a greater understanding of the trade-offs involved in the optimisations we discussed in real-life workloads.

Appendix A

Further Benchmark Results

The following pages contain the full results of the experiments described in Chapter 3 and discussed in Chapter 5.

A.1 IPC Elevator Latency Test Results

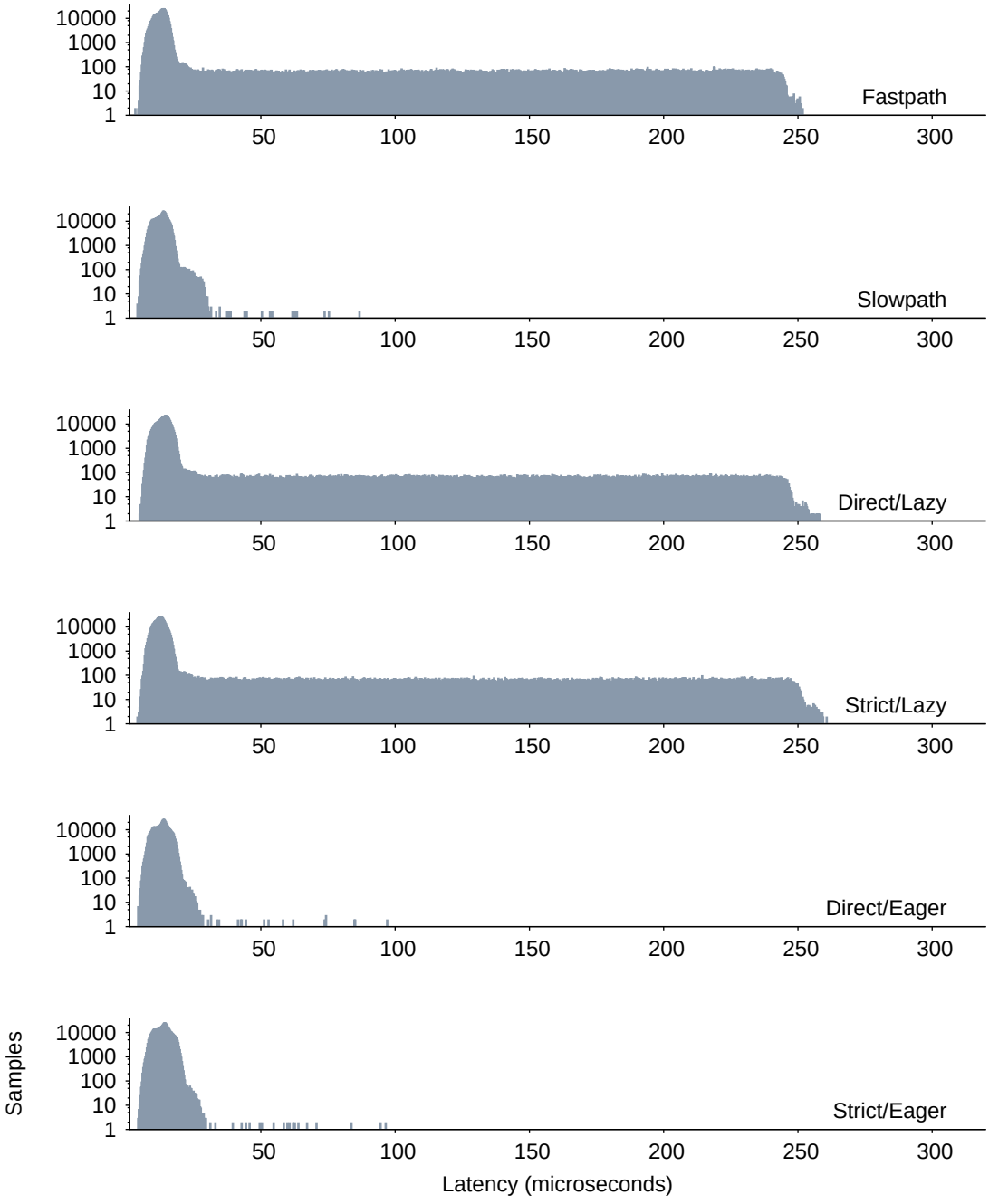


Figure A.1: *Latency measures for the IPC elevator test, described in Section 4.4.1 on page 40.*

A.2 Thread Start/Stop Latency Test Results

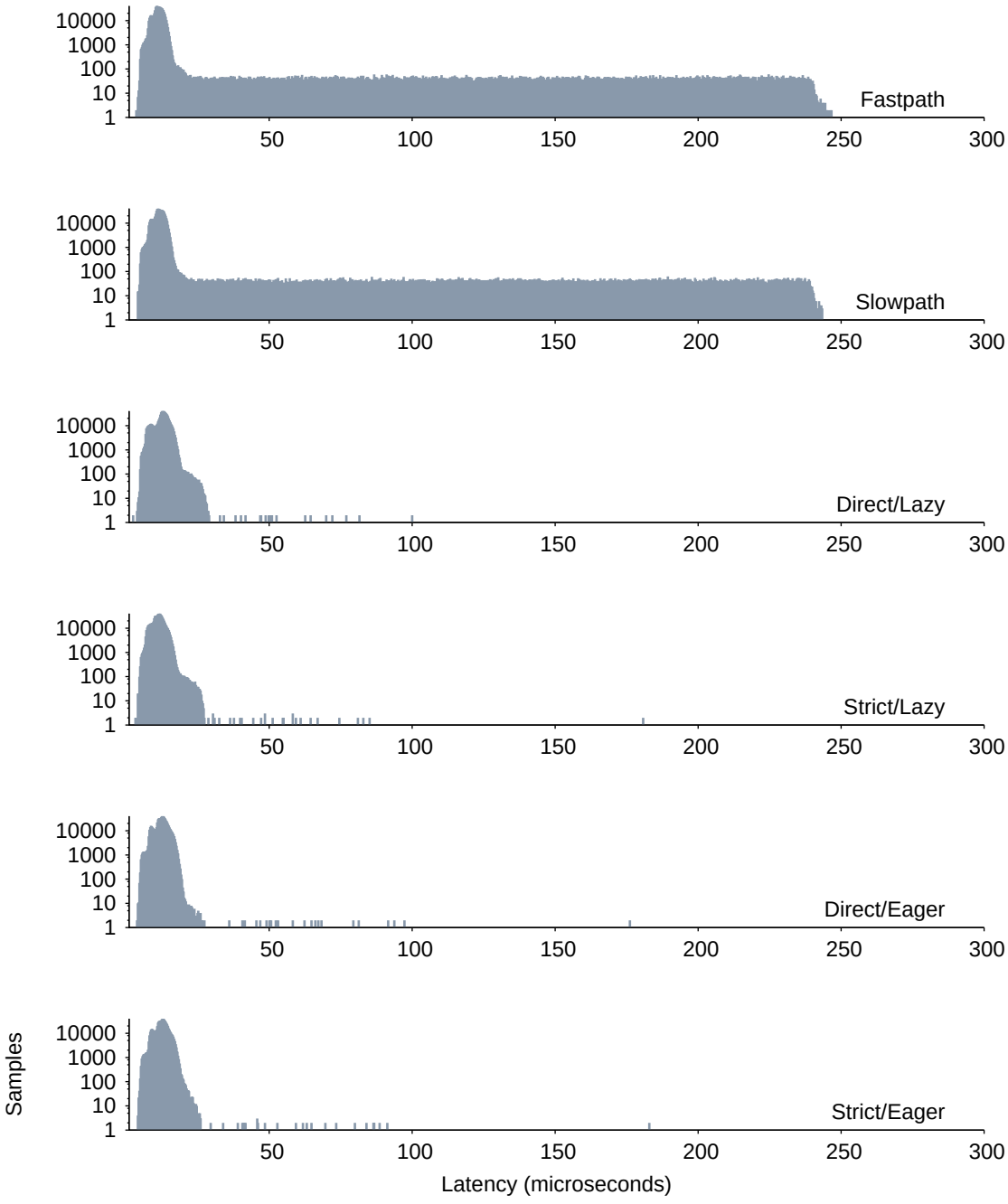


Figure A.2: *Latency measures for the thread start/stop test, described in Section 4.4.1 on page 40.*

A.3 IPC Chaining Latency Test Results

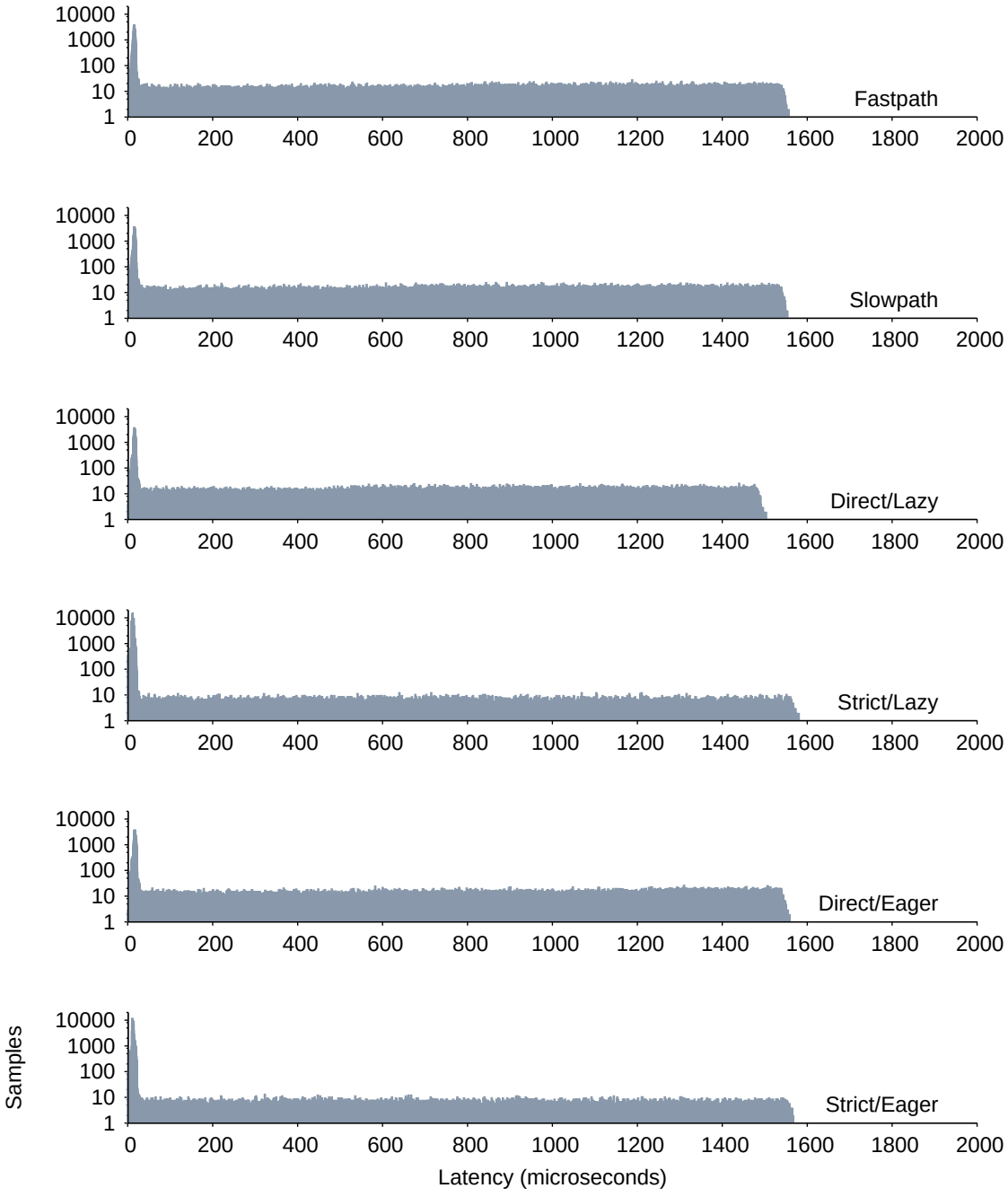


Figure A.3: *Latency measures for the IPC chaining latency test, described in Section 4.4.2 on page 41.*

A.4 Re-aim Single User Benchmarks

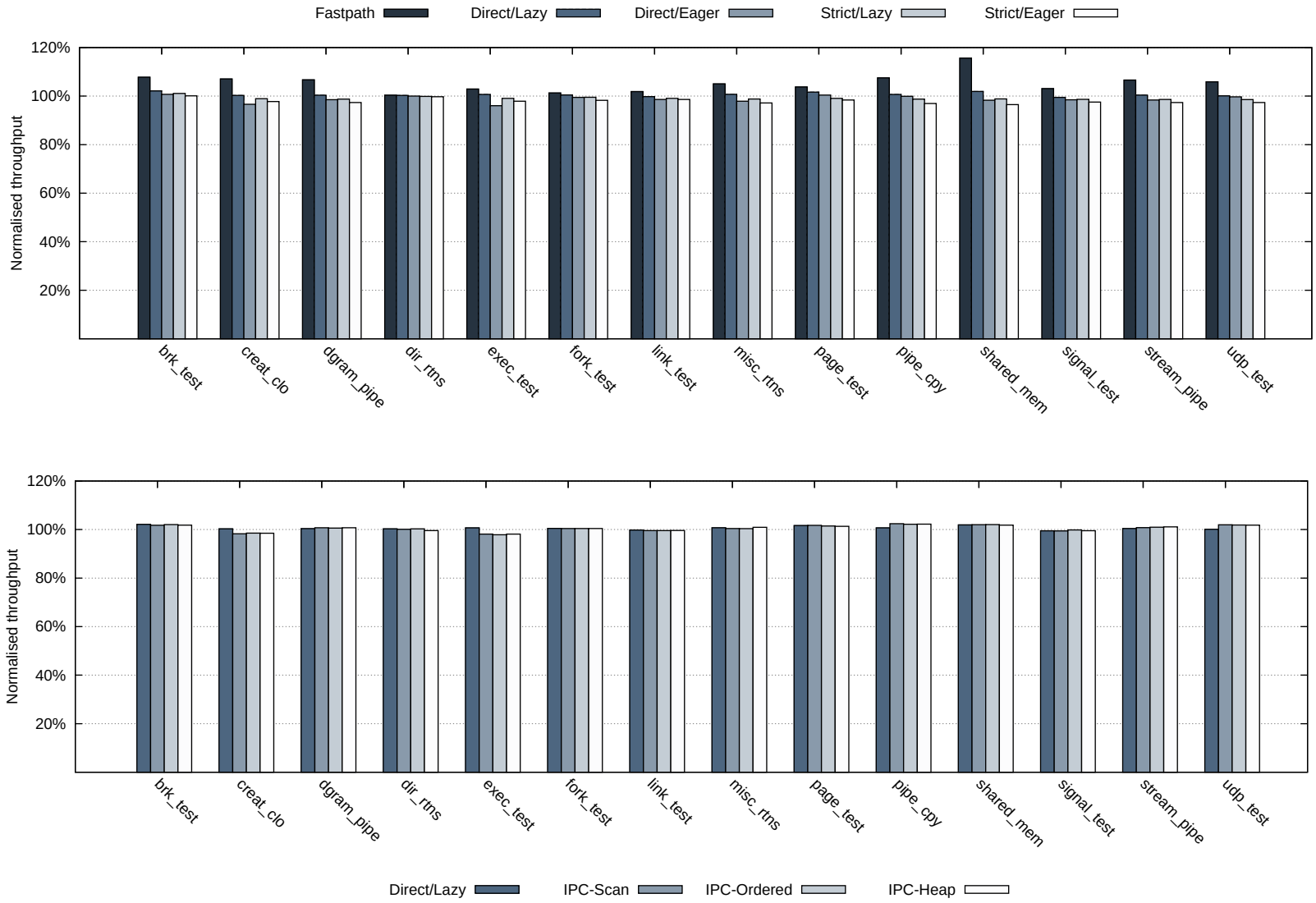


Figure A.4: *Re-aim single-user benchmark results, normalised to the performance of the* Slowpath *kernel.*

Appendix B

Fastpath Kernel Profile

Module	Function	Samples	Time (%)
vmlinux	memmove	2483775	30.075
l4kernel	sys_cache_control	1246782	15.097
vmlinux	syscall_loop	279264	3.381
l4kernel	vector_arm_swi_syscall	202029	2.446
vmlinux	ip_push_pending_frames	144646	1.751
l4kernel	vector_arm_swi_exception	135114	1.636
reaim	read_write_close	134309	1.626
vmlinux	udp_sendmsg	120695	1.461
vmlinux	ip_append_data	113373	1.373
vmlinux	flush_range_invalidate_kernel	79877	0.967
l4kernel	kip_code	75096	0.909
vmlinux	local_bh_enable	74369	0.900
vmlinux	netif_rx	73740	0.893
vmlinux	udp_queue_rcv_skb	67315	0.815
vmlinux	udp_rcv	66221	0.802
vmlinux	flush_range	64548	0.782
vmlinux	__kmalloc	61616	0.746
vmlinux	netif_receive_skb	58735	0.711
vmlinux	process_backlog	58646	0.710
vmlinux	ip_rcv	57829	0.700
vmlinux	udp_recvmsg	54195	0.656
vmlinux	dev_queue_xmit	50494	0.611
vmlinux	sock_alloc_send_pskb	48344	0.585
vmlinux	kfree	47331	0.573
vmlinux	parse_ptabs	47069	0.570

Table B.1: *Listing of the top 25 functions on a full system profile of the Re-aim multiuser benchmark '`udp`' running on the* Fastpath *kernel.*

Module	Function	Samples	Time (%)
l4kernel	sys_cache_control	1246782	15.097
l4kernel	vector_arm_swi_syscall	202029	2.446
l4kernel	vector_arm_swi_exception	135114	1.636
l4kernel	kip_code	75096	0.909
l4kernel	idle_thread	36057	0.437
l4kernel	schedule	29418	0.356
l4kernel	find_next_thread	26959	0.326
l4kernel	vector_arm_high_vector	19811	0.240
l4kernel	vector_syscall_return	19553	0.237
l4kernel	vector_common_syscall_return	18462	0.224
l4kernel	sys_ipc	17713	0.214
l4kernel	arm_page_fault	12464	0.151
l4kernel	handle_interrupt	9768	0.118
l4kernel	vector_arm_irq_exception	9023	0.109
l4kernel	vector_arm_abort_return	8323	0.101

Table B.2: *Listing of all functions in the L4 kernel measured to take more than 0.1% of system time, running the Re-aim multiuser benchmark '`udpserver`' running on the* Fastpath *kernel.*

Module	Time Spent (%)
vmlinux	69.15
l4kernel	23.62
reaim	3.40
ig_timer	0.60
ig_server	0.07
Unknown / Other	3.14

Table B.3: *Total time spent in each system module for the Re-aim '`udpserver` multiuser benchmark, running on the* Fastpath *kernel.*

Bibliography

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

[2] Kevin Elphinstone, David Greenaway, and Sergio Ruocco. Lazy scheduling and direct process switch—merit or myths? To appear, 2007.

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

[4] Rohini Krishnapura. L4 real-time issues. Technical report, 2004.

[5] Ben Leslie, Nicholas FitzRoy-Dale, and Gernot Heiser. Encapsulated user-level device drivers in the Mungi operating system. In *Proceedings of the Workshop on Object Systems and Software Architectures 2004*, Victor Harbor, South Australia, Australia, January 2004. `http://www.cs.adelaide.edu.au/ wossa2004/HTML/`.

[6] Ben Leslie, Carl van Schaik, and Gernot Heiser. Wombat: A portable user-mode Linux for embedded systems. In *Proceedings of the 6th Linux.Conf.Au*, Canberra, April 2005.

[7] R. Levin, E. S. Cohen, W. M. Corwin, F. J. Pollack, and W. A. Wulf. Policy/mechanism separation in HYDRA. In *ACM Symposium on OS Principles*, pages 132–40, 1975.

[8] 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.

[9] 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.

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

[11] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[12] National ICT Australia. *NICTA L4-embedded Kernel Reference Manual Version N2*, December 2006. `http://ertos.nicta.com.au/Software/systems/kenge/pistachio/-refman.pdf`.

[13] Sergio Ruocco. Real-Time Programming and L4 Microkernels. Dresden, Germany, July 2006.

[14] Udo A. Steinberg. Quality-assuring scheduling in the Fiasco microkernel. Diploma thesis, Dresden University of Technology, March 2004.

[15] Cliff White. Performance testing the Linux kernel. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2003.

[16] Ka-shu Wong. MacOS X on L4. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, December 2003.