Lazy Queueing and Direct Process Switch — Merit or Myths?

Kevin Elphinstone *
National ICT Australia
and
University of New South Wales
Sydney, Australia
kevine@cse.unsw.edu.au

David Greenaway
National ICT Australia
and
University of New South Wales
Sydney, Australia
davidg@cse.unsw.edu.au

Sergio Ruocco
Nomadis Lab., DISCo,
Università di Milano-Bicocca
Milano, Italy
ruocco@disco.unimib.it

ABSTRACT

The L4 microkernel, like many first and second generation microkernels, was designed to maximise best-effort performance. One component of its functionality critical to overall system performance is its interprocess communication primitive. L4 uses two techniques to minimise communication costs: direct process switching and lazy queue management. These techniques improve performance at the expense of real-time predictability of the scheduler. Now that L4 is being adopted in the embedded space, which features real-time requirements, we must determine if there is continued merit in using the optimisations. In this paper we quantitatively analyse the two optimisations using different kernel implementations and measure the performance improvements of the optimisations directly, and indirectly using the Re-aim benchmark suite. We find that the system-level performance improvements are marginal for this Unix-like workload.

1. INTRODUCTION

The functionality and the overall system complexity of high-end embedded systems is rapidly approaching, and in some cases surpassing, those of desktop systems. At the same time, they are expected to be much more reliable and robust than desktop systems as in most cases embedded systems cannot be managed by their users, and often they cannot be physically serviced.

However, traditional operating systems that are cut down to run in the embedded space usually struggle to provide strong real-time guarantees as their original design aimed at best-effort system performance, and kernel components such as interrupt handlers run outside the scheduler's control.

*National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

While timeliness can be addressed in part by running the OS on top of a realtime executive, this does not help with the reliability and complexity issues. In fact, an already large desktop operating system is expanded further by a real-time executive, and potentially real-time tasks that run without any isolation next to the desktop kernel.

To address these requirements embedded systems are moving, on the hardware side, towards processors featuring full memory management (i.e., translation and protection), and, on the software side, towards microkernel-based systems, where operating system services run as separated user-level applications, safely isolated by hardware protection.

In principle, compared to a monolithic system, in a microkernel-based system it should be easier to tame complexity and provide timeliness for high-end embedded systems. System services are decomposed into user-level services that contain most of the system functionality (and hence complexity). These user-level services execute under microkernel enforced protection boundaries (processes), which should result in improved system reliability through well defined modular structuring, and better fault isolation and fault identification.

Timeliness benefits from the smaller kernel in two ways. Firstly, being smaller in size, the kernel should be more amenable to whole kernel analysis and carefully targeted modifications to provide or improve real-time behaviour, as there are significantly less lines of privileged code to analyse or modify. Secondly, a real-time capable microkernel provides its real-time guarantees to higher-level services including interrupt handlers, device drivers and other traditional kernel services. Kernel activities that are difficult to account for (or are ignored completely) in monolithic systems, become user-level applications under control of the scheduler and the guarantees it provides.

In practice, like traditional monolithic systems, general-purpose microkernels stem from performance-driven designs, and have ingrained in their design or implementation many optimisations that aim to improve best-effort system performance at the expense of the predictability in scheduling required for real-time systems [18].

L4 [11] is a general-purpose microkernel well-known in academic circles for its contributions to low overhead commu-

nication between processes [13]. Recently L4 is gaining an industrial foothold as a basis for high-end embedded and mobile systems and as a virtualisation platform.

In this paper we focus on two performance optimisations performed in the L4 microkernel interprocess communication primitive, commonly known in the L4 community as *the IPC path*. *Direct process switching*, that avoids running the scheduler along the kernel's critical paths, and *lazy queueing*, that defers the updating of its ready queue. These optimisations decrease the cost of interprocess communication (IPC), but, as a side-effect, the first can temporarily violate the scheduling policy of the system, while the second, in pathological cases, may increase its latency to external events. We describe these two optimisations in detail, provide qualitative arguments both for and against their use, and quantify their performance benefits to allow kernel engineers and users to weigh their pros and cons in both best-effort and real-time scenarios.

2. INTERPROCESS COMMUNICATION OPTIMISATION

2.1 The Pursuit of IPC Performance

The intended structure of microkernel-based systems puts heavy demands on the performance of IPC. In microkernel-based systems, traditional operating system services — such as device drivers, filesystems and network stacks — are provided by processes (servers) running at user-level. Thus, instead of a system call to a traditional monolithic operating system, in a microkernel-based system all interactions between applications and system services involve IPC to and from servers implementing those services.

In the L4 microkernel the basic IPC mechanism is used not only to transfer messages between user-level threads, but also to deliver interrupts, asynchronous notifications, memory mappings, thread startups, thread preemptions, exceptions and page faults. Because of its pervasiveness, IPC is likely to be used very frequently. It is also evident that any kernel change that increases IPC costs will increase overhead.

It is then clear why IPC performance in L4 (and in general) has received so much attention (including, but not limited to [2,10,13,19]), with achieved performance being sufficient to support near-monolithic system performance when all system-call-like invocations are implemented by IPC to a system server, which in this case was Linux [3]. Härtig *et al.* also directly compared their system with MkLinux (a directly comparable version of Linux based on a microkernel with slower IPC performance) and demonstrated that the version of Linux running on the slower microkernel exhibited a 25% performance penalty. IPC performance was critical to overall system performance.

The requirement for high IPC performance is further motivated when device drivers are run as user-level servers to improve robustness and reduce kernel complexity [7,12]. In L4, hardware interrupt delivery is via IPC to interrupt handling threads. Interrupt delivery overhead can be critical for hardware devices such as gigabit Ethernet where, to reduce the performance impact of high interrupt rates, hardware-based interrupt throttling is now common for even normal interrupt delivery.

IPC performance affects not only the overall performance of the system, but also its design space. It has been observed early in the evolution of microkernels that given poor IPC performance, system builders will work around it by either co-locating services back within the kernel, or by composing the system with much coarser granularity than they would otherwise [4].

In addition to supporting decomposed services and applications, microkernels can also support virtual machine monitors (VMMs) as an approach to supporting legacy operating systems [3,9,17], while concurrently providing isolated environments for microkernel-based applications and services that, to implement security or temporally critical services, rely only on the guarantees provided only by the microkernel [14].

VMMs require efficient exception handling to emulate privileged instructions present in the hosted operating system. Also relevant for VMMs is the vectoring of native system calls to a paravirtualised operating system server running on the microkernel [1].

In the case of a paravirtualised Linux running on top of L4, both exception and syscall delivery are again via IPC from the Linux applications to the Linux server, or from the Linux server to the virtual machine monitor. Again IPC performance plays a primary role in determining system performance for system call-intensive (or exception-intensive) applications.

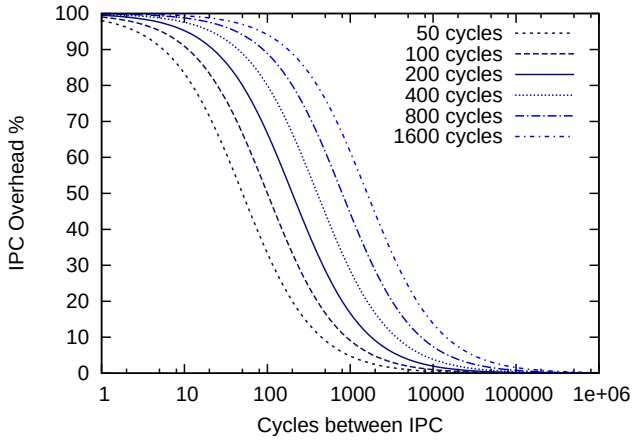

Figure 1: The IPC overhead for various IPC costs against the average cycles between successive IPC.

Figure 1 illustrates the sensitivity of IPC overhead to the raw cost of IPC, and the average cycles between successive IPCs. The x-axis represents the average number of cycles between each IPC, on a log scale. Each line is a plot of the percentage overhead attributable to IPC for hypothetical IPC costs of 50, 100, 200, 400, 800 and 1600 cycles, which are within the range of typical IPC warm-cache costs. It shows how the overhead is particularly sensitive to IPC costs in the range surrounding 5000 cycles. This corresponds broadly to

an hypothetical gigabit Ethernet interrupt delivery intervals (the time to receive a minimum sized packet on the wire) if hardware-based interrupt throttling is not used. Fine-grained synchronisation would also be in this range.

Summarising, there is strong motivation for minimising IPC costs if it is invoked frequently, and this is certainly the case for L4. In the remainder of this paper we will discuss two generally-applicable IPC optimisations used in L4 that reduce IPC cost, but affect in major and minor ways its realtime scheduling behaviour. In the following sections, to illustrate the two optimisations we will consider the case of a interprocess communication where one process *calls* another to request a service, resulting in the caller becoming blocked and the called becoming runnable; the reverse happens as a result of the response.

2.2 Direct Process Switch

In principle, every operating system, when the current process blocks, invokes the scheduler to choose the next process to run based on the specific scheduling policy it implements, e.g., the highest priority runnable process will run. However, if the process blocks in the IPC path, invoking the scheduler can be a costly operation that impacts IPC performance. Therefore L4 avoids it and switches directly to the newly runnable IPC destination, often disregarding relevant scheduling criteria, such as priority of other threads in the system.

The advantages of direct switch in the IPC path are threefold: (i) the overhead involved in calling the scheduler in the performance-critical IPC path is avoided, (ii) the latency of reaction to events delivered via IPC is reduced (also the interrupt fastpath performs a direct switch), and (iii) the cache working set may be reduced.

The first benefit does not warrant further explanation. The second benefit is advantageous (e.g. during interrupt handling) as it gives a process the opportunity to service the interrupt earlier, and therefore potentially request the next I/O operation earlier, improving I/O utilisation. The third benefit occurs as a client and server which interact closely can share the cache without the scheduler interfering by polluting the cache with the correct scheduling of a third process.

Direct process switch was first proposed by Liedke [10] to improve microkernel IPC performance. However, it makes the real-time schedulability analysis for any specific scheduling policy difficult, if not impossible, as the scheduler is not involved in the majority of scheduling decisions. In fact, scheduling decisions due to IPCs happen a few thousands of times per second, one or two orders of magnitude more frequently than those due to the scheduler running after timeslice preemptions, which happens a few hundreds of times per second.

Historically, direct switch has also been applied inconsistently in L4. To decide which process should run some L4 implementations consider the priorities of the communicating processes and the type of IPC performed, others do not, but all of them bypass the scheduler on the critical path. Ruocco [18] provides a detailed analysis of the direct switch behaviours in two recent kernels of the L4 family, and their implications for priority-driven real-time scheduling.

Notably, also the real-time OS QNX Neutrino seems to perform a direct switch in synchronous IPCs when data transfer is involved [16]:

> **Synchronous message passing**
>
> This inherent blocking synchronizes the execution of the sending thread, since the act of requesting that the data be sent also causes the sending thread to be blocked and the receiving thread to be scheduled for execution. This happens without requiring explicit work by the kernel to determine which thread to run next (as would be the case with most other forms of IPC). Execution and data move directly from one context to another.

2.3 Lazy Queueing

When performing a remote procedure call (RPC) over synchronous IPC, the sender thread blocks after sending the message, and the waiting receiver thread is unblocked after receiving the message. The blocking and unblocking of threads results in ready queue manipulation. The blocked thread must be removed from the ready queue, and the unblocked thread must be inserted in the ready queue. If two threads interact in a tight client-server loop, this happens continuously, undoing work just performed, and then performing it again.

Lazy queueing consists of the kernel deferring work in the hope that it is eventually unneeded. L4 performs lazy queue management with the following two techniques:

1. A blocking thread is not immediately removed from the ready queue. Its removal is deferred until the scheduler is called. The scheduler then removes any blocked thread(s) it encounters in the course of searching the next thread to run in the ready queue.

2. The kernel preserves the invariant that at least all ready threads not currently running must be in the ready queue. The currently running thread is not required to be in the ready queue.

   If the currently running thread is preempted (changes state from *running* to *ready*), it is added to the queue if it is not already present. Thus, switching briefly to a newly runnable thread does not require adding it to the ready queue.

In L4, the combination of these two techniques ensures that when IPC results in just one thread blocking and another running, short-lived updates that elide each other are avoided, and unavoidable queue maintenance is deferred as much as possible. The ready queue is finally updated when the messaging is preempted and the scheduler runs, typically as a result of timeslice exhaustion or a blocking IPC to a busy thread.

The pros of lazy queueing are saving the direct cost of the queue management, the indirect cost of an increased number of cache lines polluted by the queue manipulation, and avoiding the potential pollution of TLB entries, depending on the architecture and virtual memory mapping.

While lazy queueing cannot result in more overall processing performed compared to strict queue management, it does defer queue maintenance to the scheduler, where the scheduler may encounter (and remove) blocked threads in the ready queue. The number of blocked threads encountered is difficult to predict, resulting in latency of scheduling operations also being difficult to predict.

Finally, a note on terminology. Liedtke [10] calls lazy scheduling what in this paper we call lazy queueing. We use the latter term to avoid confusion with direct process switch, which can be considered a form of lazy scheduling. That said, in the L4 community and literature the term 'lazy scheduling' is sometimes used loosely to indicate a generic optimisation in scheduling, and thus can refer to direct process switch, lazy queueing, or even both optimisations.

2.4 Related Work

While there is a body of work on IPC performance, and on real-time kernels, besides the analysis mentioned above [18], there is little in the literature on the trade-off between the two IPC optimisations described, and a kernel's ability to support real-time workloads. The most relevant is Steinberg *et al.*, who proposed extending L4's IPC mechanism to donate scheduling context in order to support various classes of real-time scheduling disciplines, including priority inheritance, and reservation-based realtime systems on L4 [20].

They augment L4's IPC to do the book keeping required to track dependencies and time-slice donations. This work is complementary to our work in that they also demonstrate the need to modify their microkernel's IPC implementation to achieve their desired scheduling behaviour. They acknowledge that performance is an issue, and argue qualitatively that their system's approach adds little overhead. However, no quantitative results are given and their baseline IPC overhead is up to an order of magnitude higher than the costs we have quantified. Given a highly optimised IPC, on a favourable architecture, it is unclear that their changes would continue to be "little overhead".

2.5 Summary

In this section we have argued that IPC performance is important, and described two optimisations used in the L4 microkernel to reduce the direct and indirect costs of IPC, but with a detrimental effect on real-time workloads. Direct switch comes at the expense of the system no longer strictly adhering to its own scheduling policy — especially in the case of priority-driven scheduling — and precluding the schedulability analysis of a real-time system. Lazy queueing can increase latency in pathological cases.

In the remainder of the paper we quantify the cost in terms of performance of direct switch and lazy queueing by benchmarking the standard kernel, then removing selectively each optimisation, and finally both of them. We aim to clarify the trade-off between using and not using the two optimisations, to offer microkernel designers and users an educated choice between performance and predictability.

3. EXPERIMENTS

We performed three experiments to quantify performance differences between various optimisation configurations. The first was to instrument the kernel to collect statistics on number of IPCs, context switches, queueing operations, and scheduler invocations to determine how often queueing or scheduling is avoided. The second experiment microbenchmarks the IPC performance directly by timing repeated ping-pong messages. The third experiment measures throughput for individual components of the Re-aim benchmark suite running on a paravirtualised Linux, which, in turn, runs on the microkernel. Each experiment is described in more detail in the following sections.

For this paper we used L4-embedded N2 v1.3.0 [15], derived from L4::Ka Pistachio 0.4 [6], as a representative microkernel for experimentation. Both of them feature both the direct switch and lazy queueing optimisations described earlier. The hardware platform we used was a Gumstix Connex 400xm, which has an XScale PXA255 clocked at 400 MHz, and 64 MB of RAM. In addition to L4-embedded, we also use the Iguana operating system personality running on L4, together with Wombat (a version of Linux paravirtualised to run on Iguana on L4), to provide a system for higher-level benchmarking. Further details follow in Section 3.2.

3.1 Kernel Internal Scheduler Interface

To experiment with various combinations of scheduler optimisations, we constructed an internal scheduling interface within the L4 kernel that allows a compile-time selection of schedulers with different optimisations.

Scheduling in L4 is scattered through various parts of its source code, where scheduling decisions are made implicitly in the source each time two threads interact. For instance, when two threads communicate using IPC, the IPC code determines which thread should execute next at the conclusion of the operation without involving the scheduler, often — but not always — by directly comparing the priorities of the two threads.

The creation of an internal interface involved refactoring the code to remove the implicit scheduling decisions, which can then be centralised and performed explicitly according to a uniform and easily changeable policy.

One significant impact of this centralisation of scheduling was that the highly-optimised assembly language IPC path (known as the *fastpath*) was disabled. Instead, IPC is routed to a slower *C* language path, which uses the new internal interface.

Each of the new scheduler interface calls that involve a schedule operation also takes an additional parameter we termed a *scheduling hint*. Scheduling hints were introduced to allow the same interface to support both the behaviour of existing L4 implementations, which often dictate which thread is to be scheduled next, while also allowing other scheduling policies to be implemented, such as strict priority observance.

Listing 1 illustrates three different scheduling hints that were required to mimic the existing behaviour of L4: (i) a hint indicating that the highest priority thread in the system should be scheduled; (ii) a hint that the most recently enqueued thread should be scheduled (in the case of IPC, this emulates direct process switch); and (iii) a hint indicating that either the currently running thread or the most recently enqueued thread should be scheduled, whichever has the highest priority (used in the case of send-only IPC or interrupts).

```
/* 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);
```

Listing 1: The new L4 internal scheduling API

In addition to the hints, there are four functions which: *enqueue* or *dequeue* a thread in the ready queue, atomically block one thread and start another (*swap*), and *schedule* which chooses which thread to run next, and context switches to it. There are also three more interface functions which are clearly combinations of the previous four.

3.1.1 Measured Schedulers

In our experiments we investigated five variants of the L4-embedded kernel. The first variant was an unmodified L4-embedded kernel **Unmod**. As mentioned earlier, this kernel features an optimised assembly IPC path which was not used for the remainder of the measured variants, as we are yet to write assembly language versions of the measured schedulers that would be suitable for inclusion on an assembly language IPC path. **Unmod** simply represents a best case for comparison to gauge the effect the C internal kernel scheduling interface has on IPC performance.

The other four variants we investigated used the internal scheduling interface, described in detail in Section 3.1. These four variants implement combinations of either direct process switching (DS) or full scheduling (FS), and lazy queueing (LQ) or eager queueing (EQ). The combinations are as follows:

DS/LQ L4 with the scheduler bypassed during IPC (direct switch) with lazy queue management;

FS/EQ L4 with a full scheduler call in the IPC path together with eager queue management;

FS/LQ L4 with a full scheduler call in the IPC path together with lazy queue management;

DS/EQ L4 with the scheduler bypassed during IPC (direct switch) with eager queue management.

Note that the **DS/LQ** kernel variant reproduces the scheduling behaviour of the **Unmod** kernel using the new internal scheduling API. Therefore, the difference between **DS/LQ** and **Unmod** reflects the overhead of the C-based scheduler interface, and the lack of an optimised assembly IPC path. Obviously excluding **Unmod**, the remaining four variants are similarly implemented and are directly comparable.

3.2 Benchmarks

We compared the four scheduler variants (together with the unmodified kernel) using three approaches. Firstly, we measured the number of raw operations to determine how much queue and scheduling avoidance occurs when the optimisations are applied. Secondly, we directly measured the cost of raw IPC. Thirdly, we measured how the scheduler variants impact on the throughput of a para-virtualised version of Linux. A more detailed description of the specific benchmarks begins in Section 3.2.2, but before that we describe in more detail Wombat, our para-virtualised Linux environment, together with the Re-aim benchmark suite.

3.2.1 Wombat and Re-aim

To measure the effect on overall system performance of the scheduler variants we use *Wombat* [8], a paravirtualised version of Linux running on top of L4/Iguana [5]. A Wombat system is structured as depicted in Figure 2. The Iguana embedded OS acts as a virtual machine monitor for Wombat. Iguana provides services such as address spaces, threads, and some services (such as device drivers) that run as Iguana applications. Linux is modified by providing an L4/Iguana CPU architecture which, instead of performing direct low-level, privileged CPU operations, it uses IPC to request Iguana to provide threads and provide and manipulate address spaces for Linux processes running on Wombat, and to Wombat itself.

Overall system performance will depend on (i) the cost of propagating native system-call exceptions as IPCs from Linux applications to the Wombat instance acting as a Linux server for the Linux applications, and (ii) the cost of IPC to the Iguana virtual machine monitor when Wombat requires changes to the low-level hardware artifacts. Thus any variation in raw IPC costs may be visible depending on the level of interaction between Iguana, Wombat, and Linux (applications) processes.

To determine the relative performance of Wombat, we used the Re-aim benchmark suite [21]. Re-aim provides two modes of determining system performance. First, Re-aim has a *single-user* mode which measures the throughput of a series of operations, such as the number of TCP/IP operations

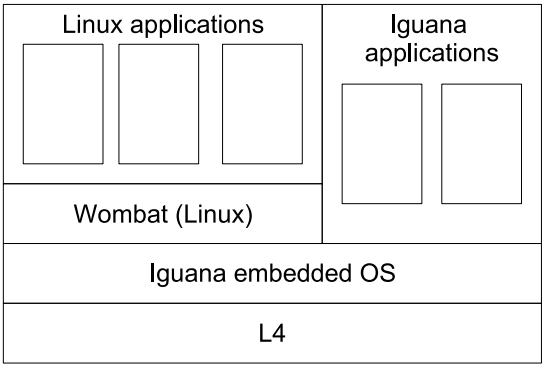

Figure 2: Wombat, Iguana OS and L4.

performed per second, number of processes created per second, the number of floating point operations per second, and so on.

Second, Re-aim has a *multi-user* mode which attempts to simulate real-world workloads. We ran the full Re-aim multi-user benchmark with five processes, each of which carries out a series of tasks in a pseudo-random order exercising both the CPU and kernel. In order to ensure the results were reproducible, we modified the Re-aim benchmark source code to seed its random number generator on the child-number of the benchmark processes, instead of the normally used Linux process-id.

Note that Wombat and Re-aim use a RAM disk to back the file system, so no real I/O occurs in the benchmarks.

3.2.2 Avoiding Work

This specific experiment uses the *multi-user* Re-aim benchmark described above. The scheduler variant is **DS/LQ**, but it has been modified to keep statistics such as number of IPCs, total actual enqueue and dequeue operations, and also total number of operations that would have occurred with eager queueing. This instrumentation is only included for this particular experiment, and only counts events. The instrumentation is not used in cases where we measure performance elsewhere in the paper.

The statistics can be used to determine how many enqueue, dequeue and scheduler operations are avoided, to illustrate the effectiveness of the technique.

3.2.3 Ping Pong

This microbenchmark consists of *ping pong*, where a low-priority client sends a message of a fixed length to a high-priority server, which then in turn responds immediately back with a message of the same length. The benchmark directly measures L4 and thus is independent of Wombat and Iguana.

We benchmark 1 000 000 iterations of ping pong using the cycle counter in the performance monitoring unit of the PXA255, and the average number of cycles of a single IPC is determined. The process is repeated for messages of various lengths. The final number of cycles counted includes both the time required for the user-level threads to call the kernel and the time spent in the kernel performing the IPC operation. We run this benckmark for each of the kernel configurations we have.

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_1`	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_1`	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.
`shell_rtns`	Execute simple shell scripts in a loop.
`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 1: Descriptions of the Re-aim single-user benchmark tasks tested.

3.2.4 Re-aim Throughput

Our last experiment takes selected single-user throughput benchmarks from the Re-aim suite. The throughput is determined by counting the number of completed activities in an interval of approximately 10 seconds (the cycle-counter on the PXA255 is used to get an accurate measurement of the length of the interval). The specific activity counted was dependant on the actual benchmark component under test. For example, the UDP test counts the number of packets sent and received. Each benchmark specific throughput was an average of 4 runs. Each runs was closely consistent with the others, the standard deviation was always less than 0.5 percent , and the average standard deviation was 0.08 percent.

Table 1 briefly describes the selected benchmarks. The selection excludes the CPU oriented benchmarks whose performance is largely independent of the underlying operating system architecture and implementation and thus not relevant for this paper.

4. RESULTS

Operation	Result
Benchmark Length (seconds)	222.16
IPCs	1450474
Average IPC Length (32-bit words)	6.26
Eager enqueue operations	1509011
Actual enqueue operations	62482
At a deferred time	19719
Enqueue operations avoided	95.86%
Eager dequeue operations	1509056
Actual dequeue operations	62482
At a deferred time	40289
Dequeue operations avoided	95.86%
Context switches	1571609
Scheduling queue lookups	80749
Queue lookups avoided	94.86%

Table 2: Breakdown of L4 operations for the Re-aim multi-user 'all tests' benchmark

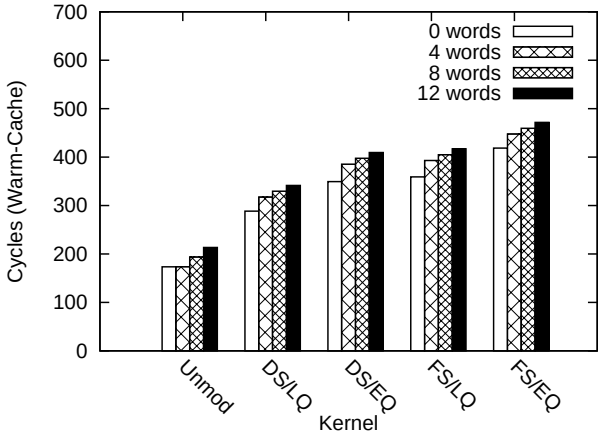

Figure 3: Raw IPC costs for various scheduler implementations.

4.1 Work Avoidance

Table 2 summarises the results of our experiment on work avoidance. The table is divided into 4 sections: general statistics of the Re-aim multi-user benchmark, enqueue operations, dequeue operations, and scheduler invocations. We see that the overall benchmark takes 222 seconds to run, averaging one IPC per 150 microseconds, which is every 62000 cycles. As Figure 1 suggests, in this inter-IPC cycle range, small variations in IPC duration should have a very small effect on the overall run-time of Re-aim, unless IPC costs become substantially greater than 1000 cycles.

Looking at the queueing results, we see that the application of lazy queue management reduces the number of queue operations substantially. We see that 96% of queue operations are avoided altogether with the technique, even when we include queueing operations that are not avoided entirely and are only deferred to a later point in time. The scheduling results show a similar percentage (95%) of scheduler invocations are avoided by the direct process switch technique.

Summarising, we see that direct process switching and lazy queue management are very effective in avoiding scheduling and queueing costs. However, given the infrequency of IPC in the Re-aim multi-user benchmark, we don't expect to see IPC overheads greater that a percentage point or two on average. However, we will see later that some of the individual benchmarks do vary significantly.

4.2 Ping Pong

Figure 3 shows the results of the ping pong benchmark for the 5 kernels. We see that **Unmod** kernel with its assembly IPC path is significantly faster (174 cycles for a zero-sized message) than **DS/LQ** (289 cycles) despite implementing the same algorithm.

This difference is attributable to two factors. The first factor is the assembly only IPC path in **Unmod** avoids preparing the kernel stack to call C, and avoids preserving the C compiler function calling convention on a context switch. The second factor is that the C path used in **DS/LQ** is a modified version of a slower IPC path (the IPC *slowpath*), also written in C. While the fastpath can handle only a frequently-used subset of all IPC cases, the slowpath can handle all of them.

Examining the four directly comparable results for the kernels with different combinations of direct switch and lazy queueing, we have the following results for zero-sized messages: **DS/LQ** 289, **DS/EQ** 350, **FS/LQ** 359, and **FS/EQ** 419. We see that direct switching saves 70 cycles off the IPC path, and lazy queue management saves 60 cycles off the IPC path. We see that there are comparatively large savings to be made to the raw cost of IPC by using both optimisation techniques.

4.3 Re-aim performance

The results for the single-user Re-aim benchmarks are shown in Figure 4. We see throughput results for the individual benchmark tests within the suite, normalised to the throughput of **Unmod**. The influence IPC performance has over the individual benchmarks varies from virtually no influence in the case of `dir_rtns_1` and `shell_rtns_2`, to a significant difference of a 17% reduction in throughput for the `shared_memory`, when comparing the assembly path kernel **Unmod** to the slowest C path kernel **FS/EQ**.

Now examining comparable results, we see the biggest differences (between **DS/LQ** and **FS/EQ**) is in the `shared_memory` benchmark, with a reduction of 5% in throughput. The average reduction in throughput was 2.5% for all the individual benchmarks.

5. CONCLUSIONS

We have described and motivated two general IPC optimisations that have historically been used in the L4 microkernel: direct process switching and lazy queueing. We have argued that the optimisations have negative consequences on real-time predictability as they undermine the scheduling policy,

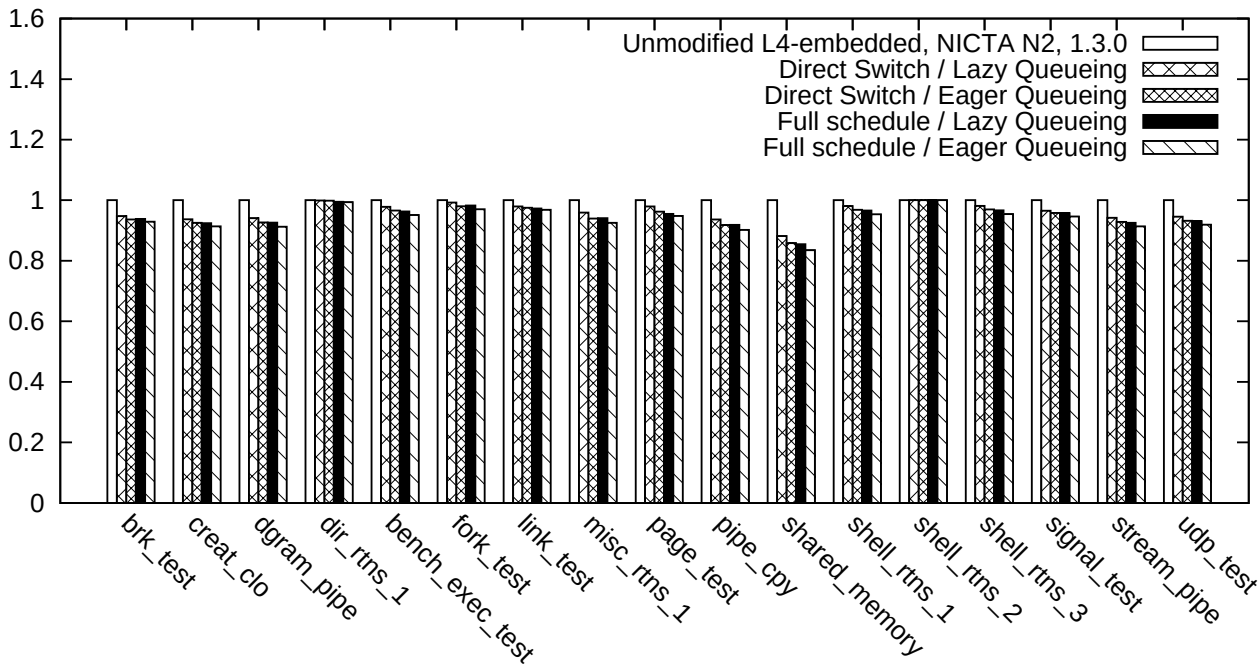

Figure 4: Individual Re-aim benchmark components normalised to standard fastpath L4

and defer a difficult-to-predict amount of work for when the scheduler is eventually invoked.

However, we also determined via measurement that these optimisations avoided scheduling related activity from IPC in over 95% of IPC invocations. The consequent improvement in best-effort system performance was dependent of the relative costs of IPC and scheduling activity, and the frequency of IPC invocation.

When we quantified their effect on the overall system performance, we found that the performance gains are modest. As expected, the overhead of IPC depends on its frequency. Removing the optimisations reduced system throughput by 2.5% on average, 5% in the worst case. Thus the case for including the optimisations at the expense of real-time predictability is weak for the cases we examined. For much higher IPC rate applications, it might still be worthwhile.

We acknowledge two weak points in our comparison that we intend to address in future work. Firstly, the work was done in C, which for IPC incurs a substantial overhead compared to the optimised assembly version. The sensitivity of IPC overhead to scheduler implementation may change in a faster assembler-only implementation. Secondly, we only examine the PXA255: other processor architectures and implementations have much larger or smaller relative IPC cost to which the results may be sensitive to.

However, our results confirm that the trade-off between performance and real-time predictability exists. For the cases we investigated, which are designed to model Unix system use, the performance gain is small, and unjustified when considering the loss of predictable scheduler behaviour.

6. REFERENCES

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on OS Principles*, pages 164–177, Bolton Landing, NY, USA, October 2003.

[2] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang, and Gernot Heiser. Itanium — a system implementor's tale. In *Proceedings of the 2005 Annual USENIX Technical Conference*, pages 264–278, Anaheim, CA, USA, April 2005.

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

[4] Wilson C. Hsieh, M. Frans Kaashoek, and William E. Weihl. The persistent relevance of IPC performance: New techniques for reducing the IPC penalty. In *Workshop on Workstation Operating Systems*, pages 186–190, 1993.

[5] Iguana OS. URL http://www.ertos.nicta.com.au/iguana/.

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

[7] Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting (Rita) Shen, Kevin Elphinstone, and

Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, September 2005.

[8] 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.

[9] Joshua LeVasseur and Volkmar Uhlig. A sledgehammer approach to reuse of legacy device drivers. In *Proceedings of the 11th SIGOPS European Workshop*, 2004.

[10] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 175–188, Asheville, NC, USA, December 1993.

[11] 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.

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

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

[14] Frank Mehnert, Michael Hohmuth, and Hermann Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, Austin, TX, USA, 2002.

[15] National ICT Australia. *NICTA L4-embedded Kernel Version N2 v. 1.3.0*. http://www.ertos.nicta.com.au/software/kenge/pistachio/latest/.

[16] QNX Neutrino IPC. URL http://www.qnx.com/developers/docs/momentics621_docs/neutrino/sys_arch/kernel.html#NTOIPC.

[17] Timothy Roscoe, Kevin Elphinstone, and Gernot Heiser. Hype and virtue. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, May 2007.

[18] Sergio Ruocco. Real-Time Programming and L4 Microkernels. In *Proceedings of the 2006 Workshop on Operating System Platforms for Embedded Real-Time Applications*, Dresden, Germany, July 2006.

[19] Jonathan S. Shapiro, David F. Faber, and Jonathan M. Smith. The measured performance of fast local IPC. In *Proceedings of the 5th IEEE International Workshop on Object Orientation in Operating Systems*, pages 89–94, Seattle, WA, USA, October 1996. IEEE.

[20] U. Steinberg, J. Wolter, and H. Härtig. Fast component interaction for real-time systems. In *Proc. 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, Palma de Mallorca, Spain, July 2005.

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