Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 234710, 14 pages
doi:10.1155/2008/234710

Research Article

A Real-Time Programmer's Tour of General-Purpose L4 Microkernels

Sergio Ruocco

Laboratorio Nomadis, Dipartimento di Informatica, Sistemistica e Comunicazione (DISCo), Università degli Studi di Milano-Bicocca, 20126 Milano, Italy

Correspondence should be addressed to Sergio Ruocco, ruocco@disco.unimib.it

Received 20 February 2007; Revised 26 June 2007; Accepted 1 October 2007

Recommended by Alfons Crespo

L4-embedded is a microkernel successfully deployed in mobile devices with soft real-time requirements. It now faces the challenges of tightly integrated systems, in which user interface, multimedia, OS, wireless protocols, and even software-defined radios must run on a single CPU. In this paper we discuss the pros and cons of L4-embedded for real-time systems design, focusing on the issues caused by the extreme speed optimisations it inherited from its general-purpose ancestors. Since these issues can be addressed with a minimal performance loss, we conclude that, overall, the design of real-time systems based on L4-embedded is possible, and facilitated by a number of design features unique to microkernels and the L4 family.

1. INTRODUCTION

Mobile embedded systems are the most challenging front of real-time computing today. They run full-featured operating systems, complex multimedia applications, and multiple communication protocols at the same time. As networked systems, they are exposed to security threats; moreover, their (inexperienced) users run untrusted code, like games, which pose both security and real-time challenges. Therefore, complete isolation from untrusted applications is indispensable for user data confidentiality, proper system functioning, and content-providers and manufacturer's IP protection.

In practice, today's mobile systems must provide functionalities equivalent to desktop and server ones, but with severely limited resources and strict real-time constraints. Conventional RTOSes are not well suited to meet these requirements: simpler ones are not secure, and even those with memory protection are generally conceived as embedded software platforms, not as operating system foundations.

L4-embedded [1] is an embedded variant of the general-purpose microkernel L4Ka::Pistachio (L4Ka) [2] that meets the above-mentioned requirements, and has been successfully deployed in mobile phones with soft real-time constraints. However, it is now facing the challenges of next-generation mobile phones, where applications, user interface, multimedia, OS, wireless protocols, and even software-defined radios must run on a single CPU.

Can L4-embedded meet such strict real-time constraints? It is thoroughly optimized and is certainly fast, but "real fast is not real-time" [3]. Is an entirely new implementation necessary, or are small changes sufficient? What are these changes, and what are the tradeoffs involved? In other words, can L4-embedded be real fast *and* real-time?

The aim of this paper is to shed some light on these issues with a thorough analysis of the L4Ka and L4-embedded internals that determine their temporal behaviour, to assess them as strengths or weaknesses with respect to real-time, and finally to indicate where research and development are currently focusing, or should probably focus, towards their improvement.

We found that (i) general-purpose L4 microkernels contain in their IPC path extreme optimisations which complicate real-time scheduling; however these optimisations can be removed with a minimal performance loss; (ii) aspects of the L4 design provide clear advantages for real-time applications. For example, thanks to the unified user-level scheduling for both interrupt and application threads, interrupt handlers and device drivers cannot impact system timeliness. Moreover, the interrupt subsystem provides a good foundation for user-level real-time scheduling.

Overall, although there is still work ahead, we believe that with few well-thought-out changes, general-purpose L4 microkernels can be used successfully as the basis of a significant class of real-time systems.

The rest of the paper is structured as follows. Section 2 introduces microkernels and the basic principles of their design, singling out the relevant ones for real-time systems. Section 3 describes the design of L4 and its API. Section 4 analyses L4-embedded and L4Ka internals in detail, their implications for real-time system design, and sketches future work. Finally, Section 5 concludes the paper.

2. MICROKERNELS

Microkernels are minimalist operating system kernels structured according to specific design principles. They implement only the smallest set of abstractions and operations that require privileges, typically address spaces, threads with basic scheduling, and message-based interprocess communication (IPC). All the other features which may be found in ordinary monolithic kernels (such as drivers, filesystems, paging, networking, etc.) but can run in user mode are implemented in user-level servers. Servers run in separate protected address spaces and communicate via IPC and shared memory using well-defined protocols.

The touted benefits of building an operating system on top of a microkernel are better modularity, flexibility, reliability, trustworthiness, and viability for multimedia and real-time applications than those possible with traditional monolithic kernels [4]. Yet operating systems based on first-generation microkernels like Mach [5] did not deliver the promised benefits: they were significantly slower than their monolithic counterparts, casting doubts on the whole approach. In order to regain some performance, Mach and other microkernels brought back some critical servers and drivers into the kernel protection domain, compromising the benefits of microkernel-based design.

A careful analysis of the real causes of Mach's lacklustre performance showed that the fault was not in the microkernel approach, but in its initial implementation [6]. The first-generation microkernels were derived by scaling down monolithic kernels, rather than from clean-slate designs. As a consequence, they suffered from poorly performing IPC and excessive footprint that thrashed CPU caches and translation lookaside buffers (TLBs). This led to a second generation of microkernels designed from scratch with a minimal and clean architecture, and strong emphasis on performance. Among them are Exokernels [7], L4 [6], and Nemesis [8].

Exokernels, developed at MIT in 1994-95, are based on the idea that kernel abstractions restrict flexibility and performance, and hence they must be *eliminated* [9]. The role of the exokernel is to securely multiplex hardware, and export *primitives* for applications to freely implement the abstractions that best satisfy their requirements.

L4, developed at GMD in 1995 as a successor of L3 [10], is based on a design philosophy less extreme than exokernels, but equally aggressive with respect to performance. L4 aims to provide flexibility and performance to an operating system via the least set of privileged abstractions.

Nemesis, developed at the University of Cambridge in 1993–95, has the aim of providing quality-of-service (QoS) guarantees on resources like CPU, memory, disk, and network bandwidth to multimedia applications.

Besides academic research, since the early 1980s the embedded software industry developed and deployed a number of microkernel-based RTOSes. Two prominent ones are QNX and GreenHills Integrity. QNX was developed in the early 1980s for the 80x86 family of CPUs [11]. Since then it evolved and has been ported to a number of different architectures. GreenHills Integrity is a highly optimised commercial embedded RTOS with a preemptable kernel and low-interrupt latency, and is available for a number of architectures.

Like all microkernels, QNX and Integrity as well as many other RTOSes rely on user-level servers to provide OS functionality (filesystems, drivers, and communication stacks) and are characterised by a small size.[1] However, they are generally conceived as a basis to run embedded applications, not as a foundation for operating systems.

2.1. Microkernels and real-time systems

On the one hand, microkernels are often associated with real-time systems, probably due to the fact that multimedia and embedded real-time applications running on resource-constrained platforms benefit from their small footprint, low-interrupt latency, and fast interprocess communication compared to monolithic kernels. On the other hand, the general-purpose microkernels designed to serve as a basis for workstation and server Unices in the 1990s were apparently meant to address real-time issues of a different nature and a coarser scale, as real-time applications on general-purpose systems (typically multimedia) had to compete with many other processes and to deal with large kernel latency, memory protection, and swapping.

Being a microkernel, L4 has intrinsic provisions for real-time. For example, user-level memory pagers enable application-specific paging policies. A real-time application can explicitly pin the logical pages that contain time-sensitive code and data in physical memory, in order to avoid page faults (also TLB entries should be pinned, though).

The microkernel design principle that is more helpful for real-time is user-level device drivers [12]. In-kernel drivers can disrupt time-critical scheduling by disabling interrupts at arbitrary points in time for an arbitrary amount of time, or create deferred workqueues that the kernel will execute at unpredictable times. Both situations can easily occur, for example, in the Linux kernel, and only very recently they have started to be tackled [13]. Interrupt disabling is just one of the many critical issues for real-time in monolithic kernels. As we will see in Section 4.7, the user-level device driver model of L4 avoids this and other problems. Two other L4 features intended for real-time support are IPC timeouts, used for time-based activation of threads (on timeouts see

[1] Recall that the "micro" in microkernel refers to its economy of concepts compared to monolithic kernels, not to its memory footprint.

Sections 3.5 and 4.1), and *preempters*, handlers for time faults that receive preemption notification messages.

In general, however, it still remains unclear whether the above-mentioned second-generation microkernels are well suited for all types of real-time applications. A first examination of exokernel and Nemesis scheduling APIs reveals, for example, that both hardwire scheduling policies that are disastrous for at least some classes of real-time systems and cannot be avoided from the user level. Exokernel's primitives for CPU sharing achieve "fairness [by having] applications pay for each excess time slice consumed by forfeiting a subsequent time slice" (see [14], page 32). Similarly, Nemesis' CPU allocation is based on a "simple QoS specification" where applications "specify neither priorities nor deadlines" but are provided with a "particular *share* of the processor over some short time frame" according to a (replaceable) scheduling algorithm. The standard Nemesis scheduling algorithm, named Atropos, "internally uses an earliest deadline first algorithm to provide this share guarantee. However, the deadlines on which it operates are *not* available to or specified by the application" [8].

Like many RTOSes, L4 contains a priority-based scheduler hardwired in the kernel. While this limitation can be circumvented with some ingenuity via user-level scheduling [15] at the cost of additional context-switches, "all that is wired in the kernel cannot be modified by higher levels" [16]. As we will see in Section 4, this is exactly the problem with some L4Ka optimisations inherited by L4-embedded, which, while being functionally correct, trade predictability and freedom from policies for performance and simplicity of implementation, thus creating additional issues that designers must be aware of, and which time-sensitive systems must address.

3. THE L4 MICROKERNEL

L4 is a second-generation microkernel that aims at high flexibility and maximum performance, but without compromising security. In order to be fast, L4 strives to be small by design [16], and thus provides only the least set of fundamental abstractions and the mechanisms to control them: address spaces with memory-mapping operations, threads with basic scheduling, and synchronous IPC.

The emphasis of L4 design on smallness and flexibility is apparent in the implementation of IPC and its use by the microkernel itself. 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, but especially its impact on OS performance experienced with first-generation microkernels, L4 IPC has received a great deal of attention since the very first designs [17] and continues to be carefully optimised today [18].

3.1. The L4 microkernel specification

In high-performance implementations of system software there is an inherent contrast between maximising the performance of a feature on a specific implementation of an architecture and its portability to other implementations or across architectures. L4 faced these problems when transitioning from 80486 to the Pentium, and then from Intel to various RISC, CISC, and VLIW 32/64 bit architectures.

L4 addresses this problem by relying on a specification of the microkernel. The specification is crafted to meet two apparently conflicting objectives. The first is to guarantee full compatibility and portability of user-level software across the matrix of microkernel implementations and processor architectures. The second is to leave to kernel engineers the maximum leeway in the choice of architecture-specific optimisations and tradeoffs among performance, predictability, memory footprint, and power consumption.

The specification is contained in a reference manual [19] that details the hardware-independent L4 API and 32/64 bit ABI, the layout of public kernel data structures such as the user thread control block (UTCB) and the kernel information page (KIP), CPU-specific extensions to control caches and frequency, and the IPC protocols to handle, among other things, memory mappings and interrupts at the user-level.

In principle, every L4 microkernel implementation should adhere to its specification. In practice, however, some deviations can occur. To avoid them, the L4-embedded specification is currently being used as the basis of a regression test suite, and precisely defined in the context of a formal verification of its implementation [20].

3.2. The L4 API and its implementations

L4 evolved over time from the original L4/x86 into a small family of microkernels serving as vehicles for OS research and industrial applications [19, 21]. In the late 1990s, because of licensing problems with then-current kernel, the L4 community started the Fiasco [22, 23] project, a variant of L4 that, during its implementation, was made preemptable via a combination of lock-free and wait-free synchronisation techniques [24]. DROPS [25] (Dresden real-time operating system) is an OS personality that runs on top of Fiasco and provides further support for real-time besides the preemptability of the kernel, namely a scheduling framework for periodic real-time tasks with known execution times distributions [26].

Via an entirely new kernel implementation Fiasco tackled many of the issues that we will discuss in the rest of the paper: timeslice donation, priority inversion, priority inheritance, kernel preemptability, and so on [22, 27, 28]. Fiasco solutions, however, come at the cost of higher kernel complexity and an IPC overhead that has not been precisely quantified [28].

Unlike the Fiasco project, our goal is not to develop a new real-time microkernel starting with a clean slate and freedom from constraints, but to analyse and improve the real-time properties of NICTA::Pistachio-embedded (L4-embedded), an implementation of the N1 API specification [1] already deployed in high-end embedded and mobile systems as a virtualisation platform [29].

Both the L4-embedded specification and its implementation are largely based on L4Ka::Pistachio version 0.4 (L4Ka) [2], with special provisions for embedded systems such as

a reduced memory footprint of kernel data structures, and some changes to the API that we will explain later. Another key requirement is IPC performance, because it directly affects virtualisation performance.

Our questions are the following ones: can L4-embedded support "as it is" real-time applications? Is an entirely new implementation necessary, or can we get away with only small changes in the existing one? What are these changes, and what are the tradeoffs involved?

In the rest of the paper, we try to give an answer to these questions by discussing the features of L4Ka and L4-embedded that affect the applications' temporal behaviour on uniprocessor systems (real-time on SMP/SMT systems entails entirely different considerations, and its treatment is outside the scope of this paper). They include scheduling, synchronous IPC, timeouts, interrupts, and asynchronous notifications.

Please note that this paper mainly focuses on L4Ka::Pistachio version 0.4 and L4-embedded N1 microkernels. For the sake of brevity, we will refer to them as simply L4, but the reader should be warned that much of the following discussion applies only to these two versions of the kernel. In particular, Fiasco makes completely different design choices in many cases. For reasons of space, however, we cannot go in depth. The reader should refer to the above-mentioned literature for further information.

3.3. Scheduler

The L4 API specification defines a 256-level, fixed-priority, time-sharing round-robin (RR) scheduler. The RR scheduling policy runs threads in priority order until they block in the kernel, are preempted by a higher priority thread, or exhaust their timeslice. The standard length of a timeslice is 10 ms but can be set between ϵ (the shortest possible timeslice) and ∞ with the `Schedule()` system call. If the timeslice is different from ∞, it is rounded to the minimum granularity allowed by the implementation that, like ϵ, ultimately depends on the precision of the algorithm used to update it and to verify its exhaustion (on timeslices see Sections 4.1, 4.4, and 4.5). Once a thread exhausts its timeslice, it is enqueued at the end of the list of the running threads of the same priority, to give other threads a chance to run. RR achieves a simple form of fairness and, more importantly, guarantees progress.

FIFO is a scheduling policy closely related to RR that does not attempt to achieve fairness and thus is somewhat more appropriate for real-time. As defined in the POSIX 1003.1b real-time extensions [30], FIFO-scheduled threads run until they relinquish control by yielding to another thread or by blocking in the kernel. L4 can emulate FIFO with RR by setting the threads' priorities to the same level and their timeslices to ∞. However, a maximum of predictability is achieved by assigning only one thread to each priority level.

3.4. Synchronous IPC

L4 IPC is a rendezvous in the kernel between two threads that partner to exchange a message. To keep the kernel simple and fast, L4 IPC is synchronous: there are no buffers or message ports, nor double copies, in and out of the kernel. Each partner performs an `Ipc(dest, from_spec, &from)` syscall that is composed of an optional send phase to the *dest* thread, followed by an optional receive phase from a thread specified by the *from_spec* parameter. Each phase can be either blocking or nonblocking. The parameters *dest* and *from_spec* can take values among all standard thread ids. There are some special thread ids, among which there are *nilthread* and *anythread*. The *nilthread* encodes "send-only" or "receive-only" IPCs. The *anythread* encodes "receive from any thread" IPCs.

Under the assumptions that IPC syscalls issued by the two threads cannot execute simultaneously, and that the first invoker requests a blocking IPC, the thread blocks and the scheduler runs to pick a thread from the ready queue. The first invoker remains blocked in the kernel until a suitable partner performs the corresponding IPC that transfers a message and completes the communication. If the first invoker requests a nonblocking IPC and its partner is not ready (i.e., not blocked in the kernel waiting for it), the IPC aborts immediately and returns an error.

A convenience API prescribed by the L4 specification provides wrappers for a number of common IPC patterns encoding them in terms of the basic syscall. For example, `Call(dest)`, used by clients to perform a simple IPC to servers, involves a blocking send to thread *dest*, followed by a blocking receive from the same thread. Once the request is performed, servers can reply and then block waiting for the next message by using `ReplyWait(dest, &from_tid)`, an IPC composed of a nonblocking send to *dest* followed by a blocking receive from *anythread* (the send is nonblocking as typically the caller is waiting, thus the server can avoid blocking trying to send replies to malicious or crashed clients). To block waiting for an incoming message one can use `Wait()`, a send to *nilthread* and a blocking receive from *anythread*. As we will see in Section 4.4, for performance optimisations the threads that interact in IPC according to some of these patterns are scheduled in special (and sparsely documented) ways.

L4Ka supports two types of IPC: standard IPC and long IPC. Standard IPC transfers a small set of 32/64-bit message registers (MRs) residing in the UTCB of the thread, which is always mapped in the physical memory. Long IPC transfers larger objects, like strings, which can reside in arbitrary, potentially unmapped, places of memory. Long IPC has been removed from L4-embedded because it can page-fault and, on nonpreemptable kernels, block interrupts and the execution of other threads for a large amount of time (see Section 4.7). Data transfers larger than the set of MRs can be performed via multiple IPCs or shared memory.

3.5. IPC timeouts

IPC with timeouts cause the invoker to block in the kernel until either the specified amount of time has elapsed or the partner completes the communication. Timeouts were originally intended for real-time support, and also as a way for clients to recover safely from the failure of servers by aborting a pending request after a few seconds (but a good way to

determine suitable timeout values was never found). Timeouts are also used by the `Sleep`() convenience function, implemented by L4Ka as an IPC to the current thread that times out after the specified amount of microseconds. Since timeouts are a vulnerable point of IPC [31], they unnecessarily complicate the kernel, and more accurate alternatives can be implemented by a time server at user level, they have been removed from L4-embedded (Fiasco still has them, though).

3.6. User-level interrupt handlers

L4 delivers a hardware interrupt as a synchronous IPC message to a normal user-level thread which registered with the kernel as the *handler thread* for that interrupt. The interrupt messages appear to be sent by special in-kernel *interrupt threads* set up by L4 at registration time, one per interrupt. Each interrupt message is delivered to exactly one handler, however a thread can be registered to handle different interrupts. The timer tick interrupt is the only one managed internally by L4.

The kernel handles an interrupt by masking it in the interrupt controller (IC), preempting the current thread, and performing a sequence of steps equivalent to an IPC `Call`() from the in-kernel interrupt thread to the user-level handler thread. The handler runs in user-mode with its interrupt disabled, but the other interrupts enabled, and thus it can be preempted by higher-priority threads, which possibly, but not necessarily, are associated with other interrupts. Finally, the handler signals that it finished servicing the request with a `Reply`() to the interrupt thread, that will then unmask the associated interrupt in the IC (see Section 4.7).

3.7. Asynchronous notification

Asynchronous notification is a new L4 feature introduced in L4-embedded, not present in L4Ka. It is used by a sender thread to notify a receiver thread of an event. It is implemented via the IPC syscall because it needs to interact with the standard synchronous IPC (e.g., applications can wait with the same syscall for either an IPC or a notification). However, notification is neither blocking for the sender, nor requires the receiver to block waiting for the notification to happen. Each thread has 32 (64 on 64-bit systems) notification bits. The sender and the receiver must agree beforehand on the semantics of the event, and which bit signals it. When delivering asynchronous notification, L4 does not report the identity of the notifying thread: unlike in synchronous IPC, the receiver is only informed of the event.

4. L4 AND REAL-TIME SYSTEMS

The fundamental abstractions and mechanisms provided by the L4 microkernel are implemented with data structures and algorithms chosen to achieve speed, compactness, and simplicity, but often disregarding other nonfunctional aspects, such as timeliness and predictability, which are critical for real-time systems.

In the following, we highlight the impact of some aspects of the L4 design and its implementations (mainly L4Ka and L4-embedded, but also their ancestors), on the temporal behaviour of L4-based systems, and the degree of control that user-level software can exert over it in different cases.

Table 1: Timer tick periods.

Version	Architecture	Timer tick (μs)
L4-embedded N1	StrongARM	10 000
L4-embedded N1	XScale	5000
L4::Ka Pistachio 0.4	Alpha	976
L4::Ka Pistachio 0.4	AMD64	1953
L4::Ka Pistachio 0.4	IA-32	1953
L4::Ka Pistachio 0.4	PowerPC32	1953
L4::Ka Pistachio 0.4	Sparc64	2000
L4::Ka Pistachio 0.4	PowerPC64	2000
L4::Ka Pistachio 0.4	MIPS64	2000
L4::Ka Pistachio 0.4	IA-64	2000
L4::Ka Pistachio 0.4	StrongARM/XScale	10 000

4.1. Timer tick interrupt

The timer tick is a periodic timer interrupt that the kernel uses to perform a number of time-dependent operations. On every tick, L4-embedded and L4Ka subtract the tick length from the remaining timeslice of the current thread and preempt it if the result is less than zero (see Algorithm 1). In addition, L4Ka also inspects the wait queues for threads whose timeout has expired, aborts the IPC they were blocked on and marks them as runnable. On some platforms L4Ka also updates the kernel internal time returned by the `SystemClock`() syscall. Finally, if any thread with a priority higher than the current one was woken up by an expired timeout, L4Ka will switch to it immediately.

Platform-specific code sets the timer tick at kernel initialisation time. Its value is observable (but not changeable) from user space in the `SchedulePrecision` field of the `ClockInfo` entry in the KIP. The current values for L4Ka and L4-embedded are in Table 1 (note that the periods can be trivially made uniform across platforms by editing the constants in the platform-specific configuration files).

In principle the timer tick is a kernel implementation detail that should be irrelevant for applications. In practice, besides consuming energy each time it is handled, its granularity influences in a number of observable ways the temporal behaviour of applications.

For example, the real-time programmer should note that, while the L4 API expresses the IPC timeouts, timeslices, and `Sleep`() durations in microseconds, their actual accuracy depends on the tick period. A timeslice of 2000 μs lasts 2 ms on SPARC, PowerPC64, MIPS, and IA-64, nearly 3 ms on Alpha, nearly 4 ms on IA-32, AMD64, and PowerPC32, and finally 10 ms on StrongARM (but 5 ms in L4-embedded running on XScale). Similarly, the resolution of `SystemClock`() is equal to the tick period (1–10 ms) on most architectures, except for IA-32, where it is based on the time-stamp counter (TSC) register that increments with CPU clock pulses. Section 4.5 discusses other consequences.

```
void scheduler_t :: handle_timer_interrupt(){
...
 /* Check for not infinite timeslice and expired */
  if ((current->timeslice_length != 0) &&
   ((get_prio_queue(current)->current_timeslice
       -= get_timer_tick_length()) <= 0))
  {
   // We have end-of-timeslice.
   end_of_timeslice (current);
  }
...
```

Algorithm 1: L4 kernel/src/api/v4/schedule.cc.

Timing precision is an issue common to most operating systems and programming languages, as timer tick resolution used to be "good enough" for most time-based operating systems functions, but clearly is not for real-time and multimedia applications. In the case of L4Ka, a precise implementation would simply reprogram the timer for the earliest timeout or end-of-timeslice, or read it when providing the current time. However, if the timer I/O registers are located outside the CPU core, accessing them is a costly operation that would have to be performed in the IPC path each time a thread blocks with a timeout shorter than the current one (recent IA-32 processors have an on-core timer which is fast to access, but it is disabled when they are put in deeper sleep modes).

L4-embedded avoids most of these issues by removing support for IPC timeouts and the `SystemClock()` syscall from the kernel, and leaving the implementation of precise timing services to user level. This also makes the kernel faster by reducing the amount of work done in the IPC path and on each tick. Timer ticks consume energy, thus will likely be removed in future versions of L4-embedded, or made programmable based on the timeslice. Linux is recently evolving in the same direction [32]. Finally, malicious code can exploit easily-predictable timer ticks [33].

4.2. IPC and priority-driven scheduling

Being synchronous, IPC causes priority inversion in real-time applications programmed incorrectly, as described in the following scenario. A high-priority thread A performs IPC to a lower-priority thread B, but B is busy, so A blocks waiting for it to partner in IPC. Before B can perform the IPC that unblocks A, a third thread C with priority between A and B becomes ready, preempts B and runs. As the progress of A is impeded by C, which runs in its place despite having a lower priority, this is a case of *priority inversion*. Since priority inversion is a classic real-time bug, RTOSes contain special provisions to alleviate its effects [34]. Among them are priority inheritance (PI) and priority ceiling (PC), both discussed in detail by Liu [35]; note that the praise of PI is not unanimous: Yodaiken [36] discusses some cons.

The L4 research community investigated various alternatives to support PI. A naïve implementation would extend IPC and scheduling mechanisms to track temporary dependencies established during blocking IPCs from higher- to lower-priority threads, shuffle priorities accordingly, resume execution, and restore them once IPC completes. Since an L4-based system executes thousands of IPCs per second, the introduction of systematic support for PI would also impose a fixed cost on nonreal-time threads, possibly leading to a significant impact on overall system performance. Fiasco supports PI by extending L4's IPC and scheduling mechanisms to donate priorities through scheduling contexts that migrate between tasks that interact in IPC [28], but no quantitative evaluation of the overhead that this approach introduces is given.

Elphinstone [37] proposed an alternative solution based on statically structuring the threads and their priorities in such a way that a high-priority thread never performs a potentially blocking IPC with a lower-priority busy thread. While this solution fits better with the L4 static priority scheduler, it requires a special arrangement of threads and their priorities which may or may not be possible in all cases. To work properly in some corner cases this solution also requires keeping the messages on the incoming queue of a thread sorted by the static priority of their senders. Greenaway [38] investigated, besides scheduling optimisations, the costs of sorted IPC, finding that it is possible "...to implement priority-based IPC queueing with little effect on the performance of existing workloads."

A better solution to the problem of priority inversion is to encapsulate the code of each critical section in a server thread, and run it at the priority of the highest thread which may call it. Caveats for this solution are ordering of incoming calls to the server thread and some of the issues discussed in Section 4.4, but overall they require only a fraction of the cost of implementing PI.

4.3. Scheduler

The main issue with the L4 scheduler is that it is hardwired both in the specification and in the implementation. While it is fine for most applications, sometimes it might be convenient to perform scheduling decisions at the user level [39], feed the scheduler with application hints, or replace it with

different ones, for example, deadline-driven or time-driven. Currently the API does not support any of them.

Yet, the basic idea of microkernels is to provide applications with mechanisms and abstractions which are sufficiently expressive to build the required functionality at the user level. Is it therefore possible, modulo the priority inheritance issues discussed in Section 4.2, to perform priority-based real-time scheduling only relying on the standard L4 scheduler? Yes, but only if two optimisations common across most L4 microkernel implementations are taken into consideration: the short-circuiting of the scheduler by the IPC path, and the simplistic implementation of timeslice donation. Both are discussed in the next two sections.

4.4. IPC and scheduling policies

L4 invokes the standard scheduler to determine which thread to run next when, for example, the current thread performs a yield with the `ThreadSwitch(nilthread)` syscall, exhausts its timeslice, or blocks in the IPC path waiting for a busy partner. But a scheduling decision is also required when the partner is ready, and as a result at the end of the IPC more than one thread can run. Which thread should be chosen? A straightforward implementation would just change the state of the threads to runnable, move them to the ready list, and invoke the scheduler. The problem with this is that it incurs a significant cost along the IPC critical path.

L4 minimises the amount of work done in the IPC path with two complementary optimisations. First, the IPC path makes scheduling decisions without running the scheduler. Typically it switches directly to one of the ready threads according to policies that possibly, but not necessarily, take their priorities into account. Second, it marks as non-runnable a thread that blocks in IPC, but defers its removal from the ready list to save time. The assumption is that it will soon resume, woken up by an IPC from its partner. When the scheduler eventually runs and searches the ready list for the highest-priority runnable thread, it also moves any blocked thread it encounters into the waiting queue. The first optimisation is called *direct process switch*, the second *lazy scheduling*; Liedke [17] provides more details.

Lazy scheduling just makes some queue operations faster. Except for some pathological cases analysed by Greenaway [38], lazy scheduling has only second-order effects on real-time behaviour, and as such we will not discuss it further. Direct process switch, instead, has a significant influence on scheduling of priority-based real-time threads, but since it is seen primarily as an optimisation to avoid running the scheduler, the actual policies are sparsely documented, and missing from the L4 specification. We have therefore analysed the different policies employed in L4-embedded and L4Ka, reconstructed the motivation for their existence (in some cases the policy, the motivation, or both, changed as L4 evolved), and summarised our findings in Table 2 and the following paragraphs. In the descriptions, we adopt this convention: "A" is the *current* thread, that sends to the *dest* thread "B" and receives from the *from* thread "C." The policy applied depends on the type of IPC performed:

`Send()` at the end of a send-only IPC two threads can be run: the sender A or the receiver B; the current policy respects priorities and is cache-friendly, so it switches to B only if it has higher priority, otherwise continues with A. Since asynchronous notifications in L4-embedded are delivered via a send-only IPC, they follow the same policy: a waiting thread B runs only if it has higher priority than the notifier A, otherwise A continues.

`Receive()` thread A that performs a receive-only IPC from C results in a direct switch to C.

`Call()` client A which performs a call IPC to server B results in a direct switch of control to B.

`ReplyWait()` server A that responds to client B, and at the same time receives the next request from client C, results in a direct switch of control to B only if it has a strictly higher priority than C, otherwise control switches to C.

Each policy meets a different objective. In `Send()` it strives to follow the scheduler policy: the highest priority thread runs — in fact it only approximates it, as sometimes A may *not* be the highest-priority runnable thread (a consequence of timeslice donation: see Section 4.5). L4/MIPS [40] was a MIPS-specific version of L4 now superseded by L4Ka and L4-embedded. In its early versions the policy for `Send()` was set to continue with the receiver B to optimise a specific OS design pattern used at the time; in later versions the policy changed to always continue with the sender A to avoid priority inversion.

In other cases, the policies at the two sides of the IPC cooperate to favour brief IPC-based thread interactions over the standard thread scheduling by running the ready IPC partner on the timeslice of the current thread (also for this see Section 4.5).

Complex behaviour

Complex behaviour can emerge from these policies and their interaction. As the IPC path copies the message from sender to receiver in the final part of the send phase, when B receives from an already blocked A, the IPC will first switch to A's context in the kernel. However, once it has copied the message, the control may or may not immediately go back to B. In fact, because of the IPC policies, what will actually happen depends on the type of IPC A is performing (send-only, or send+receive), which of its partners are ready, and their priorities.

A debate that periodically resurfaces in the L4 community revolves around the policy used for the `ReplyWait()` IPC (actually the policy applies to any IPC with a send phase followed by a receive phase, of which `ReplyWait()` is a case with special arguments). If both B and C can run at the end of the IPC, and they have the same priority, the current policy arbitrarily privileges C. One effect of this policy is that a loaded server, although it keeps servicing requests, it limits the progress of the clients who were served and could resume execution. A number of alternative solutions which

Table 2: Scheduling policies in general-purpose L4 microkernels (∗ = see text).

Situation	When/where applied	Scheduling policy	switch_to(...)
ThreadSwitch (to)	application syscall	timeslice donation	to
ThreadSwitch (*nilthread*)	application syscall	scheduler	(highest pri. ready)
End of timeslice (typically 10 ms)	timer tick handler runs scheduler	scheduler	(highest pri. ready)
send(dest) blocks (no partner)	ipc send phase runs scheduler	scheduler	(highest pri. ready)
recv(from) blocks (no partner)	ipc recv phase runs scheduler	scheduler	(highest pri. ready)
send(dest) [Send()]	ipc send phase	direct process switch	maxpri(current, dest)
send(dest) [Send()] L4/MIPS	ipc send phase	timeslice donation	dest
send(dest) [Send()] L4/MIPS*	ipc send phase	(arbitrary)	current
recv(from) [Receive()]	ipc recv phase	timeslice donation	from
send(dest)+recv(dest) [Call()]	ipc send phase	timeslice donation	dest
send(dest)+recv(*anythread*) [ReplyWait()]	ipc recv phase	direct process switch	maxpri (dest,*anythread*)*
send(dest)+recv(from)	ipc recv phase	direct process switch	maxpri (dest, from)
Kernel interrupt path	handle_interrupt()	direct process switch	maxpri(current, handler)
Kernel interrupt path	handle_interrupt()*	timeslice donation	handler
Kernel interrupt path	irq_thread() completes Send()	timeslice donation	handler
Kernel interrupt path	irq_thread(), irq after Receive()	(as handle_interrupt())	(as handle_interrupt())
Kernel interrupt path L4-embedded	irq_thread(), no irq after Receive()	scheduler	(highest pri. ready)
Kernel interrupt path L4Ka	irq_thread(), no irq after Receive()	direct process switch	idle_thread

meet different requirements are under evaluation to be implemented in the next versions of L4-embedded.

Temporary priority inversion

In the `Receive`() and `Call`() cases, if A has higher priority than C, the threads with intermediate priority between A and C will not run until C blocks, or ends its timeslice. Similarly, in the `ReplyWait`() case, if A has higher priority than the thread that runs (either B or C, say X), other threads with intermediate priority between them will not run until X blocks, or ends its timeslice. In all cases, if the intermediate threads have a chance to run before IPC returns control to A, they generate temporary priority inversion for A (this is the same real-time application bug discussed in Section 4.2).

Direct switch in QNX

Notably, also the real-time OS QNX Neutrino performs a direct switch in synchronous IPCs when data transfer is involved [41]:

> Synchronous message passing
>
> This inherent blocking synchronises 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.

IPC fastpath

Another optimisation of the L4 IPC is the *fastpath*, a hand-optimised, architecture-specific version of the IPC path which can very quickly perform the simplest and most common IPCs: transfer untyped data in registers to a specific thread that is ready to receive (there are additional requirements to fulfill: for more details, Nourai [42] discusses in depth a fastpath for the MIPS64 architecture). More complex IPCs are routed to the standard IPC path (also called the *slowpath*) which handles all the cases and is written in C. The fastpath/slowpath combination does not affect real-time scheduling, except for making most of the IPCs faster (more on this in Section 4.6). However, for reasons of scheduling consistency, it is important that if the fastpath performs a scheduling decision, then it replicates the same policies employed in the slowpath discussed above and shown in Table 2.

4.5. Timeslice donation

An L4 thread can donate the rest of its timeslice to another thread, performing the so-called *timeslice donation* [43]. The thread receiving the donation (recipient) runs briefly: if it does not block earlier, it runs ideally until the donor timeslice ends. Then the scheduler runs and applies the standard scheduling policy that may preempt the recipient and, for example, run another thread of intermediate priority between it and the donor which was ready to run since before the donation.

L4 timeslice donations can be explicit or implicit. Explicit timeslice donations are performed by applications with the `ThreadSwitch(to_tid)` syscall. They were initially intended by Liedtke to support user-level schedulers, but never used for that purpose. Another use is in mutexes

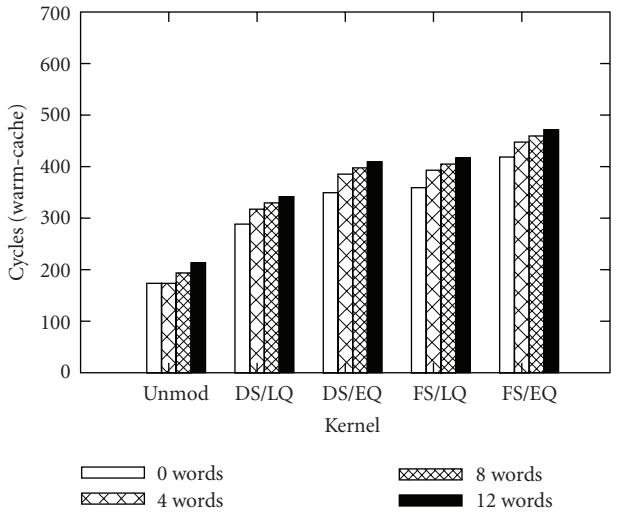

Figure 1: Raw IPC costs versus optimisations.

that — when contended — explicitly donate timeslices to their holder to speed-up the release of the mutex. Implicit timeslice donations happen in the kernel when the IPC path (or the interrupt path, see Section 4.7) transfers control to a thread that is ready to rendezvous. Note, however, that although implicit timeslice donation and direct process switch conflate in IPC, they have very different purposes. Direct process switch optimises scheduling in the IPC critical path. Timeslice donation favours threads interacting via IPC over standard scheduling. Table 2 summarises the instances of timeslice donation found in L4Ka and L4-embedded.

This is the theory. In practice, both in L4Ka and L4-embedded, a timeslice donation will *not* result in the recipient running for the rest of the donor timeslice. Rather, it will run *at least* until the next timer tick, and *at most* for *its own* timeslice, before it is preempted and normal scheduling is restored. The actual timeslice of the donor (including a timeslice of ∞) is not considered at all in determining how long the recipient runs.

This manifest deviation from what is stated both in the L4Ka and L4-embedded specifications [19] (and implied by the established term "timeslice donation") is due to a simplistic implementation of timeslice accounting. In fact, as discussed in Section 4.1 and shown in Algorithm 1, the scheduler function called by the timer tick handler simply decrements the timeslice of the current thread. It neither keeps track of the donation it may have received, nor does it propagate them in case donations are nested. In other words, what currently happens upon timeslice donation in L4Ka and L4-embedded is better characterised as a *limited timer tick donation*. The current terminology could be explained by earlier L4 versions which had timeslices and timerticks of coinciding lengths. Fiasco correctly donates timeslices at the price of a complex implementation [28] that we cannot discuss here for space reasons. Finally, Liedtke [44] argued that kernel fine-grained time measurement can be cheap.

The main consequence of timeslice donation is the temporary change of scheduling semantics (i.e., priorities are temporarily disregarded). The other consequences depend on the relative length of donor timeslices and timer tick. If both threads have a normal timeslice and the timer tick is set to the same value, the net effect is just about the same. If the timer tick is shorter than the donor timeslice, what gets donated is statistically much less, and definitely platform-dependent (see Table 1). The different lengths of the donations on different platforms can resonate with particular durations of computations, and result in occasional large differences in performance which are difficult to explain. For example, the performance of I/O devices (that may deliver time-sensitive data, e.g., multimedia) decreases dramatically if the handlers of their interrupts are preempted before finishing and are resumed after a few timeslices. Whether this will happen or not can depend on the duration of a donation from a higher priority interrupt dispatcher thread. Different lengths of the donations can also conceal or reveal race conditions and priority inversions caused by IPC (see Section 4.4).

4.6. IPC performance versus scheduling predictability

As discussed in Sections 4.4 and 4.5, general-purpose L4 microkernels contain optimisations that complicate priority-driven real-time scheduling. A natural question arises: how much performance is gained by these optimisations? Would it make sense to remove these optimisations in favour of priority-preserving scheduling? Elphinstone et al. [45], as a follow-up to [46] (subsumed by this paper), investigated the performance of L4-embedded (version 1.3.0) when both the direct switch (DS) and lazy scheduling (LQ, lazy queue manipulation) optimisations are removed, thus yielding a kernel which schedules threads strictly following their priorities. For space reasons, here we briefly report the findings, inviting the reader to refer to the paper for the rest of the details. Benchmarks have been run on an Intel XScale (ARM) PXA 255 CPU at 400 MHz.

Figure 1 shows the results of ping-pong, a tight loop between a client thread and server thread which exchange a fixed-length message. Unmod is the standard L4-embedded kernel with all the optimisations enabled, including the fastpath; the DS/LQ kernel has the same optimisations, except that, as the experimental scheduling framework, it lacks a fastpath implementation, in this and the subsequent kernels all IPCs are routed through the slowpath; the DS/EQ kernel performs direct switch and *eager* queuing (i.e., it disables lazy queuing). The FS/LQ and FS/EQ kernels perform *full scheduling* (i.e., *respect* priorities in IPC), and lazy and eager queuing, respectively. Application-level performance has been evaluated using the Re-aim benchmark suite run in a Wombat, Iguana/L4 system (see the paper for the full Re-Aim results and their analysis).

Apparently, the IPC "de-optimisation" gains scheduling predictability but reduces the raw IPC performance. However, its impact at the application level is limited. In fact, it has been found that "...the performance gains [due to the two optimisations] are modest. As expected, the overhead of IPC depends on its frequency. Removing the optimisations reduced [Re-aim] 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." Summarising [45], it is possible to have a real-time friendly, general-purpose L4 microkernel without the issues caused by priority-unaware scheduling in the IPC path discussed in Section 4.4, at the cost of a moderate loss of IPC performance. Based on these findings, scheduling in successors of L4-embedded will be revised.

4.7. Interrupts

In general, the causes of interrupt-related glitches are the most problematic to find and are the most costly to solve. Some of them result from subtle interactions between how and when the hardware architecture generates interrupt requests and how and when the kernel or a device driver decides to mask, unmask or handle them. For these reasons, in the following paragraphs we first briefly summarise the aspects of interrupts critical for real-time systems. Then we show how they influence real-time systems architectures. Finally, we discuss the ways in which L4Ka and L4-embedded manage interrupts and their implications for real-time systems design.

Interrupts and real-time

In a real-time system, interrupts have two critical roles. First, when triggered by timers, they mark the passage of real-time and specific instants when time-critical operations should be started or stopped. Second, when triggered by peripherals or sensors in the environment, they inform the CPU of asynchronous events that require immediate consideration for the correct functioning of the system. Delays in interrupt handling can lead to jitter in time-based operations, missed deadlines, and the lateness or loss of time-sensitive data.

Unfortunately, in many general-purpose systems (e.g., Linux) both drivers and the kernel itself can directly or indirectly disable interrupts (or just pre-emption, which has a similar effect on time-sensitive applications) at unpredictable times, and for arbitrarily long times. Interrupts are disabled not only to maintain the consistency of shared data structures, but also to avoid deadlocks when taking spin locks and to avoid unbounded priority inversions in critical sections. Code that manipulates hardware registers according to strictly timed protocols should disable interrupts to avoid interferences.

Interrupts and system architecture

In Linux, interrupts and device drivers can easily interfere with real-time applications. A radical solution to this problem is to interpose between the kernel and the hardware a layer of privileged software that manages interrupts, timers and scheduling. This layer, called "real-time executive" in Figure 2, can range from a interrupt handler to a full-fledged RTOS (see [47–50], among others) and typically provides a real-time API to run real-time applications at a priority higher than the Linux kernel, which runs, de-privileged, as a low-priority thread. For obvious reasons, this is known as the *dual-kernel* approach. A disadvantage of some of these earlier real-time Linux approaches like RT-Linux and RTAI is that real-time applications seem (documentation is never very clear) to share their protection domain and privileges among themselves, with the real-time executive, or with Linux and its device drivers, leading to a complex and vulnerable system exposed to bugs in any of these subsystems.

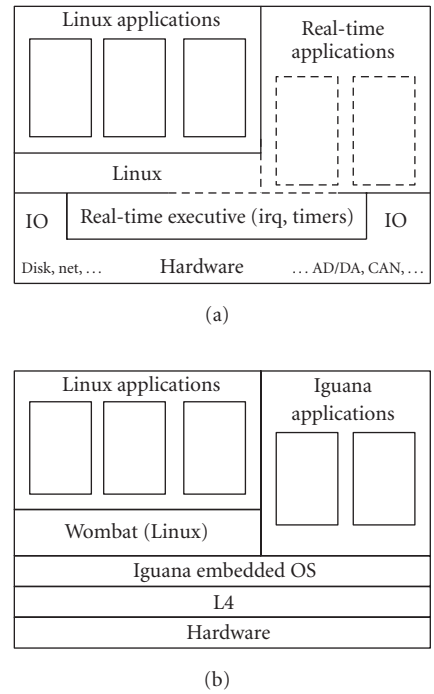

Figure 2: (a) A typical real-time Linux system. (b) Wombat, Iguana OS, and L4.

L4-embedded, combined with the Iguana [51] embedded OS and Wombat [29] (virtualised Linux for Iguana) leads to a similar architecture (Figure 2(b)) but with full memory protection and control of privileges for all components enforced by the embedded OS and the microkernel itself. Although memory protection does not come for free, it has been already proven that a microkernel-based RTOS can support real-time Linux applications in separate address spaces at costs, in terms of interrupt delays and jitter, comparable to those of blocked interrupts and caches, costs that seem to be accepted by designers [27]. However the situation in the Linux field is improving. XtratuM overcomes the protection issue by providing instead "a memory map per OS, enabling memory isolation among different OSes" [48], thus an approach more similar to Figure 2(b). In a different effort, known as the *single-kernel* approach, the standard Linux kernel is modified to improve its real-time capabilities [52].

L4 interrupts

As introduced in Section 3.6, L4 converts all interrupts (but the timer tick) into IPC messages, which are sent to a user-level thread which will handle them. The internal interrupt

path comprises three routines: the generic `irq_thread`(), the generic `handle_interrupt`(), and a low-level, platform-specific handler that manages the IC.

When L4 receives an interrupt, the platform-specific handler disables it in the IC and calls `handle_interrupt`(), which creates an interrupt IPC message and, if the user-level handler is not waiting for it, enqueues the message in the handler's message queue, marks the in-kernel interrupt thread as runnable (we will see its role shortly), and returns to the current (interrupted) thread. If instead the handler is waiting, and the current thread is an interrupt kernel thread or the idle thread, `handle_interrupt`() switches directly to the handler, performing a *timeslice donation*. Finally, if the handler is waiting and the current thread was neither an interrupt thread nor the idle thread, it does *direct process switch* and switches to the handler only if it has higher priority than the current thread, otherwise it moves the interrupt thread in the ready queue, and switches to the current thread, like the IPC path does for a `Send`() (see Sections 4.4 and 4.5).

Each in-kernel interrupt thread (one for each IRQ line) executes `irq_thread`(), a simple endless loop that performs two actions in sequence. It delivers a pending message to a user-level interrupt handler which became ready to receive, and then blocks waiting to receive its reply when it processed the interrupt. When it arrives, `irq_thread`() re-enables the interrupt in the IC, marks itself halted and, if a new interrupt is pending, calls `handle_interrupt`() to deliver it (which will suspend the interrupt thread and switch to the handler, if it is waiting). Finally, it yields to another ready thread (L4-embedded) or to the idle thread (L4Ka). In other words, the interrupt path *mimics* a `Call`() IPC. The bottom part of Table 2 summarises the scheduling actions taken by the interrupt paths of the L4Ka and L4-embedded microkernels.

Advantages

In L4-based systems, only the microkernel has the necessary privileges to enable and disable interrupts globally in the CPU and selectively in the interrupt controller. All user-level code, including drivers and handlers, has control only over the interrupts it registered for, and can disable them only either by simply not replying to an interrupt IPC message, or by deregistering altogether, but cannot mask any other interrupt or all of them globally (except by entering the kernel, which, in some implementations, e.g., L4Ka, currently disables interrupts).

An important consequence of these facts is that L4-based real-time systems do not need to trust drivers and handlers time-wise, since they cannot programmatically disable all interrupts or preemption. More importantly, at the user-level, mutual exclusion between a device driver and its interrupt handler can be done using concurrency-control mechanisms that do not disable preemption or interrupts like spin locks must do in the kernel. Therefore, user-level driver-handler synchronisations only have a *local* effect, and thus neither unpredictably perturb the timeliness of other components of the system, nor contribute to its overall latency.

Another L4 advantage is the unification of the scheduling of applications and interrupt handlers. Interrupts can negatively impact the timeliness of a system in different ways, but at least for the most common ones, L4 allows simple solutions which we discuss in the following.

A typical issue is the long-running handler, either because it is malicious, or simply badly written as is often the case. Even if it cannot disable interrupts, it can still starve the system by running as the highest-priority ready thread. A simple remedy to bound its effects is to have it scheduled at the same priority as other threads, if necessary tune its timeslice, and rely on L4's round-robin scheduling policy which ensures global progress (setting its priority lower than other threads would unfairly starve the device).

A second issue is that critical real-time threads must not be delayed by less important interrupts. In L4, low-priority handlers cannot defer higher priority threads by more than the time spent by the kernel to receive each user-registered interrupt once and queue the IPC message. Also the periodic timer tick interrupt contributes to delaying the thread, but for a bounded and reasonably small amount of time.

Consider finally a third common issue: thrashing caused by interrupt overload, where the CPU spends all its time handling interrupts and nothing else. L4 prevents this by design, since after an interrupt has been handled, it is the handler which decides if and when to handle the next pending interrupt, not the microkernel. In this case, even if the handler runs for too long because it has no provision for overload, it can still be throttled via scheduling as discussed above.

Notably, in all these cases it is not necessary to trust drivers and handlers to guarantee that interrupts will not disrupt in one way or another the timeliness of the system. In summary, running at user-level makes interrupt handlers and device drivers in L4-based systems *positively constrained*, in the sense that their behaviour — as opposed to the in-kernel ones — cannot affect the OS and applications beyond what is allowed by the protection and scheduling policies set for the system.

Disadvantages

Interrupt handling in L4 has two drawbacks: latency and, in some implementations, a nonpreemptable kernel. The interrupt latency is defined as the time between when the interrupt is asserted by the peripheral and the first instruction of its handler is executed. The latency is slightly higher for L4 user-level handlers than for traditional in-kernel ones since even in the best-case scenario more code runs and an additional context switch is performed. Besides, even if thoroughly optimised, L4 (like Linux and many RTOSes) does not provide specific real-time guarantees (i.e., a firm upper bound in cycles), neither for its API in general, nor for its IPC in particular. That said, interrupt latency in L4 is bounded and "smaller" than normal IPC (as the interrupt IPC path is much simpler than the full IPC path).

Both kernel preemptability and latency in L4-embedded are currently considered by researchers and developers. The current implementation of L4Ka disables interrupts while in kernel mode. Since in the vast majority of cases the time spent in the kernel is very short, especially if compared to monolithic kernels, and preempting the kernel has about

the same cost as a fast system call, L4-embedded developers maintain that the additional complexity of making it *completely* preemptable is not justified.

However, the L4Ka kernel takes a very long time to "unmap" a set of deeply nested address spaces, and this increases both the interrupt and the preemption worst-case latencies. For this reason, in L4-embedded the virtual memory system has been reworked to move part of the memory management to user level, and introduce preemption points where interrupts are enabled in long-running kernel operations. Interestingly, also the Fiasco kernel implementation has a performance/preemptability tradeoff. It is located in the registers-only IPC path, which in fact runs with interrupts disabled and has explicit preemption points [53].

A notable advantage of a nonpreemptable kernel, or one with very few preemption points, is that it can be formally verified to be correct [20].

With respect to latency, Mehnert et al. [27] investigate the worst-case interrupt latency in Fiasco. An ongoing research project [54] aims, among other things, at precisely characterising the L4-embedded interrupt latency via a detailed timing analysis of the kernel. Finally, to give an idea of the interrupt latencies achieved with L4-embedded, work on the interrupt fast path (an optimised version of the interrupt IPC delivery path) for a MIPS 2000/3000-like superscalar CPU reduced the interrupt latency by a factor of 5, from 425 cycles for the original C version down to 79 cycles for the hand-optimised assembly version.

User-level real-time reflective scheduling

The L4 interrupt subsystem provides a solid foundation for user-level real-time scheduling. For example, we exploited the features and optimisations of L4-embedded described in this paper to realise a *reflective* scheduler, a time-driven scheduler based on reflective abstractions and mechanisms that allow a system to perform *temporal reflection*, that is to explicitly model and control its temporal behaviour as an object of the system itself [55].

Since the reflective scheduler is time-driven, and to synchronise uses mutexes and asynchronous notifications, its correctness is not affected by direct process switch optimisation in the IPC path. Our implementation does not require any changes to the microkernel itself, and thus does not impact on its performance. On a 400 MHz XScale CPU it performs time-critical operations with both accuracy and precision of the order of microseconds [56]. To achieve this performance we implemented the reflective scheduler using a user-level thread always ready to receive the timer IPC, and with a priority higher than normal applications. In this configuration, L4 can receive the interrupt and perform a direct process switch from kernel mode to the user level thread without running the kernel scheduler. As a consequence, the latency of the reflective scheduler is low and roughly constant, and can be compensated. A simple closed-loop feedback calibration reduces it from 52–57 μs to $-4-+3\,\mu$s. For space reasons we cannot provide more details. Please find in Ruocco [56] the full source code of the scheduler and the real-time video analysis application we developed to evaluate it.

5. CONCLUSIONS

In this paper we discussed a number of aspects of L4 microkernels, namely L4-embedded and its general-purpose ancestor L4::Ka Pistachio, that are relevant for the design of real-time systems. In particular we showed why the optimisations performed in the IPC path complicate real-time scheduling. Fortunately, these optimisations can be removed with a small performance loss at application level, achieving real-time friendly scheduling in L4-embedded. Finally, we highlighted the advantages of the L4 interrupt subsystem for device drivers and real-time schedulers operating at the user level. Overall, although there is still work ahead, we believe that with few well-thought-out changes microkernels like L4-embedded can be used successfully as the basis of a significant class of real-time systems, and their implementation is facilitated by a number of design features unique to microkernels and the L4 family.

ACKNOWLEDGMENTS

The author wants to thank Hal Ashburner, Peter Chubb, Dhammika Elkaduwe, Kevin Elphinstone, Gernot Heiser, Robert Kaiser, Ihor Kuz, Adam Lackorzynski, Geoffrey Lee, Godfrey van der Linden, David Mirabito, Gabriel Parmer, Stefan Petters, Daniel Potts, Marco Ruocco, Leonid Ryzhyk, Carl van Schaik, Jan Stoess, Andri Toggenburger, Matthew Warton, Julia Weekes, and the anonymous reviewers. Their feedback improved both the content and style of this paper.

REFERENCES

[1] National ICT Australia, *NICTA L4-embedded kernel reference manual version N1*, October 2005, http://ertos.nicta.com.au/software/kenge/pistachio/latest/refman.pdf.

[2] L4Ka Team, "L4Ka: pistachio kernel," http://l4ka.org/projects/pistachio/.

[3] P. S. Langston, "Report on the workshop on micro-kernels and other kernel architectures," April 1992, http://www.langston.com/Papers/uk.pdf.

[4] J. Liedtke, "Towards real microkernels," *Communications of the ACM*, vol. 30, no. 9, pp. 70–77, 1996.

[5] D. B. Golub, R. W. Dean, A. Forin, and R. F. Rashid, "UNIX as an application program," in *Proceedings of the Summer USENIX*, pp. 87–59, Anaheim, California, USA, June 1990.

[6] J. Liedtke, "On μ-kernel construction," in *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP '95)*, pp. 237–250, Copper Mountain, Colorado, USA, December 1995.

[7] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr., "Exokernel: an operating system architecture for application-level resource management," in *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP '95)*, pp. 251–266, Copper Mountain, Colorado, USA, December 1995.

[8] I. M. Leslie, D. McAuley, R. Black, et al., "The design and implementation of an operating system to support distributed multimedia applications," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1280–1296, 1996.

[9] D. R. Engler and M. F. Kaashoek, "Exterminate all operating system abstractions," in *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS '95)*, pp. 78–85, Orcas Island, Washington, USA, May 1995.

[10] J. Liedtke, "A persistent system in real use experience of the first 13 years," in *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems (IWOOOS '93)*, pp. 2–11, Asheville, North Carolina, USA, December 1993.

[11] D. Hildebrand, "An architectural overview of QNX," in *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures*, pp. 113–126, Berkeley, California, USA, 1992.

[12] B. Leslie, P. Chubb, N. Fitzroy-Dale, et al., "User-level device drivers: achieved performance," *Journal of Computer Science and Technology*, vol. 20, no. 5, pp. 654–664, 2005.

[13] T. P. Baker, A. Wang, and M. J. Stanovich, "Fitting Linux device drivers into an analyzable scheduling framework," in *Proceedings of the 3rd Workshop on Operating System Platforms for Embedded Real-Time Applications*, Pisa, Italy, July 2007.

[14] D. Engler, *The Exokernel operating system architecture*, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, Massachussets, USA, 1999.

[15] J. Kamada, M. Yuhara, and E. Ono, "User-level realtime scheduler exploiting kernel-level fixed priority scheduler," in *Proceedings of the International Symposium on Multimedia Systems*, Yokohama, Japan, March 1996.

[16] J. Liedtke, "μ-kernels must and can be small," in *Proceedings of the 15th International Workshop on Object Orientation in Operating Systems (IWOOOS '96)*, pp. 152–161, Seattle, Washington, USA, October 1996.

[17] J. Liedtke, "Improving IPC by kernel design," in *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP '03)*, pp. 175–188, Asheville, North Carolina, USA, December 2003.

[18] G. Gray, M. Chapman, P. Chubb, D. Mosberger-Tang, and G. Heiser, "Itanium — a system implementor's tale," in *Proceedings of the USENIX Annual Technical Conference (ATEC '05)*, pp. 265–278, Anaheim, California, USA, April 2005.

[19] L4 headquarters, "L4 kernel reference manuals," June 2007, http://l4hq.org/docs/manuals/.

[20] G. Klein, M. Norrish, K. Elphinstone, and G. Heiser, "Verifying a high-performance micro-kernel," in *Proceedings of the 7th Annual High-Confidence Software and Systems Conference*, Baltimore, Maryland, USA, May 2007.

[21] R. Kaiser and S. Wagner, "Evolution of the PikeOS microkernel," in *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems (MIKES '07)*, Sydney, Australia, March 2007.

[22] H. Härtig and M. Roitzsch, "Ten years of research on L4-based real-time systems," in *Proceedings of the 8th Real-Time Linux Workshop*, Lanzhou, China, 2006.

[23] M. Hohmuth, "The Fiasco kernel: requirements definition," Tech. Rep. TUD-FI98-12, Technische Universität Dresden, Dresden, Germany, December 1998.

[24] M. Hohmuth and H. Härtig, "Pragmatic nonblocking synchronization for real-time systems," in *Proceedings of the USENIX Annual Technical Conference*, pp. 217–230, Boston, Massachussets, USA, June 2001.

[25] H. Härtig, R. Baumgartl, M. Borriss, et al., "DROPS: OS support for distributed multimedia applications," in *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, pp. 203–209, Sintra, Portugal, September 1998.

[26] C. J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig, "Quality-assuring scheduling — using stochastic behavior to improve resource utilization," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS '01)*, pp. 119–128, London, UK, December 2001.

[27] F. Mehnert, M. Hohmuth, and H. Härtig, "Cost and benefit of separate address spaces in real-time operating systems," in *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS '02)*, pp. 124–133, Austin, Texas, USA, December, 2002.

[28] U. Steinberg, J. Wolter, and H. Härtig, "Fast component interaction for real-time systems," in *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS '05)*, pp. 89–97, Palma de Mallorca, Spain, July 2005.

[29] B. Leslie, C. van Schaik, and G. Heiser, "Wombat: a portable user-mode Linux for embedded systems," in *Proceedings of the 6th Linux Conference*, Canberra, Australia, April 2005.

[30] ISO/IEC, *The POSIX 1003.1 Standard*, 1996.

[31] J. S. Shapiro, "Vulnerabilities in synchronous IPC designs," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 251–262, Oakland, California, USA, May 2003.

[32] S. Siddha, V. Pallipadi, and A. Van De Ven, "Getting maximum mileage out of tickless," in *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, June 2007.

[33] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Secretly monopolizing the CPU without superuser privileges," in *Proceedings of the 16th USENIX Security Symposium*, Boston, Massachussets, USA, April 2007.

[34] Mike Jones, "What really happened on Mars Rover Pathfinder," *The Risks Digest*, vol. 19, no. 49, 1997, based on David Wilner's keynote address of 18th IEEE Real-Time Systems Symposium (RTSS '97), December, 1997, San Francisco, California, USA. http://research.microsoft.com/~mbj/Mars_Pathfinder/.

[35] J. W. S. Liu, *Real-Time Systems*, Prentice-Hall, Upper Saddle River, New Jersey, USA, 2000.

[36] V. Yodaiken, "Against priority inheritance," http://www.yodaiken.com/papers/inherit.pdf.

[37] K. Elphinstone, "Resources and priorities," in *Proceedings of the 2nd Workshop on Microkernels and Microkernel-Based Systems*, K. Elphinstone, Ed., Lake Louise, Alta, Canada, October 2001.

[38] D. Greenaway, "From 'real fast' to real-time: quantifying the effects of scheduling on IPC performance," B. Sc. thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2007.

[39] J. Stoess, "Towards effective user-controlled scheduling for microkernel-based systems," *Operating Systems Review*, vol. 41, no. 4, pp. 59–68, 2007.

[40] G. Heiser, "Inside L4/MIPS anatomy of a high-performance microkernel," Tech. Rep. 2052, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2001.

[41] QNX Neutrino IPC, http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/sys_arch/kernel.html#NTOIPC.

[42] A. Nourai, "A physically-addressed L4 kernel," B. Eng. thesis, School of Computer Science & Engineering, The University of New South Wales, Sydney, Australia, 2005, http://www.disy.cse.unsw.edu.au/.

[43] B. Ford and S. Susarla, "CPU inheritance scheduling," in *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*, pp. 91–105, Seattle, Washington, USA, October 1996.

[44] J. Liedtke, "A short note on cheap fine-grained time measurement," *Operating Systems Review*, vol. 30, no. 2, pp. 92–94, 1996.

[45] K. Elphinstone, D. Greenaway, and S. Ruocco, "Lazy queueing and direct process switch — merit or myths?" in *Proceedings of the 3rd Workshop on Operating System Platforms for Embedded Real-Time Applications*, Pisa, Italy, July 2007.

[46] S. Ruocco, "Real-time programming and L4 microkernels," in *Proceedings of the 2nd Workshop on Operating System Platforms for Embedded Real-Time Applications*, Dresden, Germany, July 2006.

[47] "Xenomai — implementing a RTOS emulation framework on GNU/Linux," 2004, http://www.xenomai.org/documentation/branches/v2.3.x/pdf/xenomai.pdf.

[48] M. Masmano, I. Ripoll, and C. Crespo, "An overview of the XtratuM nanokernel," in *Proceedings of the Workshop on Operating System Platforms for Embedded Real-Time Applications*, Palma de Mallorca, Spain, July 2005.

[49] Politecnico di Milano — Dipartimento di Ingegneria Aerospaziale, "RTAI the real-time application interface for Linux," http://www.rtai.org/.

[50] V. Yodaiken and M. Barabanov, "A real-time Linux," http://citeseer.ist.psu.edu/article/yodaiken97realtime.html.

[51] Iguana OS, http://www.ertos.nicta.com.au/iguana/.

[52] S. Rostedt and D. V. Hart, "Internals of the RT Patch," in *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, January 2007.

[53] R. Reusner, "Implementierung eines echtzeit-IPC-Pfades mit unterbrechungspunkten für L4/Fiasco," Diplomarbeit Thesis, Fakultät Informatik, Technische Universität Dresden, Dresden, Germany, 2006.

[54] S. M. Petters, P. Zadarnowski, and G. Heiser, "Measurements or static analysis or both?" in *Proceedings of the 7th Workshop Worst-Case Execution-Time Analysis*, Pisa, Italy, July 2007.

[55] S. Ruocco, *Temporal reflection*, Ph.D. thesis, Università degli Studi di Milano, Milano, Italy, 2004.

[56] S. Ruocco, "User-level fine-grained adaptive real-time scheduling via temporal reflection," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS '06)*, pp. 246–256, Rio de Janeiro, Brazil, December 2006.