Real-Time Programming and L4 Microkernels

Sergio Ruocco
National ICT Australia*
and
School of Computer Science and Engineering
University of New South Wales, Sydney 2052, Australia
sergio.ruocco@nicta.com.au

Abstract

L4-embedded is a microkernel successfully deployed in mobile devices with soft real-time requirements that now faces the challenges of tightly integrated systems, where user interface, multimedia, OS, wireless protocols and even software-defined radios must run on a single CPU. This paper discusses the various aspects of real-time programming on L4-embedded, focusing on the issues caused by the extreme speed optimisations it inherited from its ancestors. We conclude that real-time programming on L4-embedded is facilitated by a number of design features unique to microkernels and L4, but a review of the tradeoffs between performance and predictability would ease priority-driven real-time programming.

1. Introduction

The most challenging front of real-time today are mobile embedded systems. They run fully-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, that poses both security and real-time challenges. Therefore complete isolation from untrusted applications is indispensable for user data confidentiality, proper system functioning, 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 operating systems foundations.

L4-embedded [24] is a second-generation microkernel that meets these 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" [15]. The aim of this paper is to shed some light on these issues by a thorough analysis of the L4-embedded internals that determine its temporal behaviour, to assess them as strengths or weaknesses with respect to real-time, and finally to indicate where research and development are currently focussing, or should probably focus, towards their improvement.

It has been found that (i) some aspects of the L4 design are clear advantages for real-time systems, for example interrupt handlers and device drivers cannot impact system timeliness; (ii) the extreme performance optimisations that L4-embedded inherited from previous implementations, especially those performed in the critical IPC path, are, to a large degree, the main sources of complexity for real-time scheduling.

We conclude that (i) real-time programming on top of L4-embedded is possible, provided that programmers are aware of some of its implementation details, and take appropriate measures for the case at hand; (ii) a review of the current tradeoffs between performance and predictability would ease priority-driven real-time programming.

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 in detail L4-embedded internals and their implications for real-time systems design. Finally Section 5 concludes the paper.

*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.

2. Microkernels

Microkernels are minimalist operating systems 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 [22]. Yet operating systems based on first-generation microkernels like Mach [6] 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, however, showed that the fault was not in the microkernel idea, but in its initial implementation. 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 TLBs [20]. 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 [4], L4 [20] and Nemesis [17]:

Exokernels developed at MIT in 1994-95 based on the idea that kernel abstractions restrict flexibility and performance, hence must be *eliminated* [3]. The role of the exokernel is to protect and 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 [19] based on a design philosophy less extreme than exokernels, but equally aggressive with respect to performance. L4 provides high flexibility and performance to an operating system via the least set of privileged abstractions.

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

Besides academic research, since the early '80s 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 early '80s for the 80x86 family of CPUs [8]. Since then it evolved and has been ported to a number of different architectures. GreenHills Integrity is a highly optimised commercial embedded RTOS with preemptable kernel and low interrupt latency, and it 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 base 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 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 bases for workstation and server Unices in the '90s were apparently meant to address real-time issues of different nature and a coarser scale, as real-time applications (typically multimedia) compete with many other processes and must deal with large kernel latency, memory protection and swapping.

As a microkernel, L4 has intrinsic provisions for real-time. Memory pagers, being at user-level, support application-specific policies. A real-time application can explicitly pin the logical pages that contain time-sensitive code and data in physical memory, so as to avoid page faults (also TLB entries should be pinned, though).

The microkernel design principle more helpful for real-time is user-level device drivers [16]. 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. Interrupt disabling is just one of the many critical issues for real-time in monolithic kernels. As we will see in Section 4.6, the user-level device driver model of L4 avoids this problem. Two other L4 features intended for real-time support are IPC timeouts, used for time-based activation of threads (on timeouts see 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 these 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

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

hardwire scheduling policies that are disastrous for at least some classes of real-time systems and cannot be avoided from 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 [2], p.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" [17].

L4, like many RTOSes, contains a priority-based scheduler hardwired in the kernel. While this limitation can be circumvented with some ingenuity via user-level scheduling [13] at the cost of additional context-switches, "all that is wired in the kernel cannot be modified by higher levels" [21]. As we will see in Section 4, this is exactly the problem with some L4 optimisations, 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 [21], 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 [18] and continues to be carefully optimised today [7].

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 choice of architecture-specific optimisations and trade-offs among performance, predictability, memory footprint, and power consumption.

The specification is contained in a reference manual 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 user-level.

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

3.2. The L4 API

L4 evolved over time from the original L4/x86 into a small family of microkernels serving as vehicles for OS research and experimentation. In the late '90s, because of licensing problems with the then current kernel, the L4 community started the Fiasco [9] project, a variant of L4 that, during its implementation, was made preemptable via a combination of lock-free and wait-free synchronisation techniques [10]. Beside preemptability, Fiasco sports a sophisticated (but complex) support for a subset of real-time applications: those composed by strictly periodic tasks [25].

This paper focuses on NICTA::Pistachio-embedded (L4-embedded), an implementation of the N1 API specification [24]. Both the L4-embedded specification and its implementation are largely based on L4Ka::Pistachio version 0.4 (L4Ka) [14], with special provisions for embedded systems such as reduced memory footprint of kernel data structures and some changes to the API that we detail later. In the following, we discuss the features of L4Ka and L4-embedded that affect the application temporal behavior. Those include scheduling, synchronous IPC, timeouts, interrupts, and asynchronous notifications.

3.3. Scheduler

The L4 API specification defines a 256-level, fixed-priority, 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 [11], 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 non-blocking. The parameters *dest* and *from_spec* can take values among all standard thread ids. There are some special thread ids, among which *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 non-blocking 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 mandated 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 using `ReplyWait(dest,&from_tid)`, an IPC composed of a non-blocking send to *dest* followed by a blocking receive from *anythread* (the send is non-blocking as typically the caller is waiting, thus the server can avoid to block 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 residing in the thread's UTCB, which is always mapped in physical memory. Long IPC transfers larger objects, like strings, that can reside in arbitrary, potentially unmapped, places of memory. Long IPC has been removed from L4-embedded because it can pagefault and, on non-preemptable kernels, block interrupts and the execution of other threads for a large amount of time (see Section 4.6).

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 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 IPC timeouts unnecessarily complicate the kernel and more accurate alternatives can be implemented at user level, they have been removed from L4-embedded.

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.

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. While implemented via the IPC syscall, 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 non-functional 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.

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 N1 and L4Ka subtract the tick length from the remaining timeslice of the current thread and preempt it if the result is less than zero (Listing 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.

```
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 );
 }
...
```

Listing 1. L4 kernel/src/api/v4/schedule.cc

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.

In principle the timer tick is a kernel implementation detail that should be irrelevant for applications. In practice its granularity influences, in a number of observable ways, their temporal behaviour. As a consequence, all other things being equal, the actual fine-grained temporal behaviour of an L4-based system is essentially platform-dependent.

For example, while the 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 ARM (but 5 ms in L4-embedded running on StrongARM). 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 timestamp counter (TSC) register that increments with CPU clock pulses. Section 4.5 discusses other consequences.

Version	Architecture	Timer tick (μs)
L4-embedded N1	XScale	10000
L4-embedded N1	StrongARM	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	10000

Table 1. Timer tick periods.

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 L4, 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, in most cases timer IO registers are located outside the CPU core and accessing them is a costly operation that would have to be performed in the IPC path if a thread blocks with a timeout, and on each context switch.

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.

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 [12]. Among them are priority inheritance (PI) and priority ceiling (PC), both discussed in detail by Liu in [23]; Yodaiken [26] discusses the cons of PI. In order to support PI in L4, IPC and scheduling mechanisms must be extended 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 non-real-time threads, leading to a significant impact on overall system performance.

Elphinstone [1] 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 it also requires keeping the messages on the incoming queue of a thread sorted by the static priority of their senders. These changes, just like the ones necessary for PI, slow down the critical IPC path, complicate the kernel implementation, and have not been implemented. An efficient solution that does not require sorting is under evaluation, to be implemented in future L4-embedded kernels.

A better solution to the problem of priority inversion is to encapsulate critical sections in server threads. If the server thread is assigned the priority of the highest thread which may call it, it will implement the PC protocol at little or no extra cost. 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 hard-wired both in the specification and the implementation. While fine for most applications, sometimes it might be convenient to perform scheduling decisions at user level, feed the scheduler with application hints, or replace it with different one, e.g., deadline-driven or time-driven. Unfortunately the API does not support any of them.

Yet, the basic idea of microkernels is to provide applications with mechanisms and abstractions sufficiently expressive to build the required functionality at user level. Is it therefore possible, modulo the priority inheritance issues discussed in Section 4.2, to perform priority-based real-time scheduling just 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 threads' state to runnable, move them to the ready list, and invoke the scheduler. The problem with this is, of course, 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 into account their priorities. 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 [18] provides more details.

As resuming a thread in the ready queue is slightly faster than one in the waiting queue, lazy scheduling has only second-order effects in scheduling, 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

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()	(arbitrary)	idle_thread

Table 2. Scheduling policies in L4 microkernels (* = see text).

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 (that in some cases 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, else continues with A.

Receive() thread A that performs a receive-only IPC from C results in a direct transfer of control to C.

Call() client A that 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 (e.g., because of timeslice donation: see Section 4.5). In the 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, once active, although keeps servicing requests, limits the progress of the clients that were served and could resume execution. A number of alternative solutions that meet different requirements are under evaluation to be implemented in 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 betweeen them will not run until X blocks, or ends its timeslice. In all cases, if the intermediate threads have a chance to run before A's IPC terminates, they generate temporary priority inversion for A (this is the same real-time application bug discussed in Section 4.2).

Software evolution With respect to software evolution, it is interesting to look at how the policies change over time to meet different requirements. Today, to mantain a uniform behaviour between the standard IPC path and the ARM's IPC *fastpath* (a hand-optimised architecture-specific version of the IPC path invoked in special, faster cases), the `ReplyWait()` policy in the standard path has been changed to reflect the policy implemented in the fastpath. Once the fastpath is re-implemented correctly, the standard path may be returned to the standard policy.

4.5. Timeslice donation

An L4 thread can donate the rest of its timeslice to another thread, performing the so-called *timeslice donation* [5]. 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 run another thread of intermediate priority between it and the donor that 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. Implicit timeslice donations happen in the kernel when the IPC path (or the interrupt path, see Section 4.6) transfers control to a thread that is ready to rendezvous. Note, however, that even though implicit timeslice donation and direct process switch combine 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 in both 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 is not considered at all in determining how long the recipient runs.

This manifest deviation from what is stated in the L4 specification (and implied by the established term 'timeslice donation') is a known bug, due to a simplistic implementation of timeslice accounting. In fact, as discussed in Section 4.1 and shown in Listing 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 *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 significantly more complex implementation that we cannot discuss here for space reasons.

The main consequence of timeslice donation is the temporary change of scheduling semantics. 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 IO devices (that may deliver time-sensitive data, e.g., multimedia) decreases dramatically if the handlers of their interrupts are preempted before finishing and resumed after 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). Finally, a timeslice of ∞ cannot be donated.

4.6. Interrupts

In general, the causes of interrupt-related glitches are the most problematic to find and 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 or handle them. For these reasons, in the following paragraphs we will first briefly summarise the aspects of interrupts critical for real-time systems, then we will discuss how, in detail, L4 manages interrupts, and finally the implications for L4-based real-time systems design.

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 systems 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 mantain the consistency of shared data structures, but also to avoid deadlocks when taking spinlocks and to avoid unbounded priority inversions in critical sections. Handlers that manipulate hardware registers according to strictly timed protocols disable all interrupts in the system.

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 that will handle them. The internal interrupt path comprises of 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 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 the 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 the 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, else 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).

The in-kernel interrupt thread executes `irq_thread()`, a simple endless loop that performs two actions in sequence. It delivers a pending message to a user-level interrupt handler that 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 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 de-registering altogether, but cannot mask any other interrupt or all of them globally (except by entering the kernel, that 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 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 spinlocks 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. A typical issue is the long-running handler, either because it is malicious, or simply badly written as it 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 that 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, 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, and not the microkernel that decides if and when to handle the next pending interrupt. 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 *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 Two drawbacks of L4 interrupts for real-time systems are interrupt latency and non-preemptable kernel. The interrupt latency is the time between when the interrupt is asserted by the peripheral and the first instruction of its handler is executed. The latency is 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 many RTOSes) does not provide specific real-time guarantees, neither for its API in general, nor for IPC in particular. Therefore, interrupt latency in L4 is not precisely characterised yet.

Both kernel preemptability and latency are currently considered by researchers and developers. The current imple-

mentation 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* pre-emptable 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. With respect to latency, an ongoing research project aims, among other things, at precisely characterising the L4 interrupt latency via a detailed timing analysis of the kernel.

5. Conclusions

We conclude that real-time programming on top of L4-embedded is facilitated by a number of design features unique to microkernels and L4 itself, provided that programmers are aware of some of its implementation details, and take appropriate measures for the case at hand. However, a review of the current tradeoffs between performance and predictability that L4-embedded inherited from L4Ka's extreme optimisations would ease priority-driven real-time scheduling. Future work will explore the applicability of other real-time scheduling paradigms to L4. In particular we aim to exploit the very same L4 optimisations to realise accurate and deterministic real-time schedulers that do not require any changes to the microkernel itself, and thus do not impact on its performance.

Acknowledgments

The author wants to thank Peter Chubb, Dhammika Elkaduwe, Kevin Elphinstone, Gernot Heiser, Ihor Kuz, Geoffrey Lee, Godfrey van der Linden, David Mirabito, Stefan Petters, Daniel Potts, Marco Ruocco, Leonid Ryzhyk, Carl van Schaik, Matthew Warton and the anonymous reviewers. Their feedback improved both the content and style of this paper.

References

[1] K. Elphinstone. Resources and Priorities. In K. Elphinstone, editor, *2nd Workshop on Microkernels and Microkernel-based Systems*, Lake Louise, Alta, Canada, Oct 2001.

[2] D. Engler. *The Exokernel operating system architecture*. Ph.D. Thesis, Massachusetts Institute of Technology, 1999.

[3] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *5th HotOS*, Orcas Island, WA, USA, May 1995.

[4] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *15th SOSP*, pages 251–266, Copper Mountain, CO, USA, Dec 1995.

[5] B. Ford and S. Susarla. CPU Inheritance Scheduling. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, 1996.

[6] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an application program. In *1990 Summer USENIX Techn. Conf.*, Jun 1990.

[7] C. Gray, M. Chapman, P. Chubb, D. Mosberger-Tang, and G. Heiser. Itanium — a system implementor's tale. In *2005 USENIX Techn. Conf.*, pages 264–278, Anaheim, CA, USA, Apr 2005.

[8] D. Hildebrand. An architectural overview of QNX. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126. USENIX, 1992.

[9] M. Hohmuth. The Fiasco kernel: requirements definition. Technical Report TUD-FI98-12, Dec 1998.

[10] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *2001 USENIX Techn. Conf.*, Boston, MA, USA, 2001.

[11] ISO/IEC. *The POSIX 1003.1 Standard.* 1996. [FIXME] ISBN 1-55937-061-0; 1996 revision of POSIX.1; includes POSIX.1(1990), POSIX.1b(1993), POSIX.1c(1995), and POSIX.1i(1995).

[12] M. Jones. What really happened on Mars Rover Pathfinder. *The Risks Digest*, 19, 1997. Based on David Wilner's keynote address of 18th IEEE Real-Time Systems Symposium (RTSS '97), Dec 3-5, 1997, San Francisco, CA, USA. http://catless.ncl.ac.uk/Risks/19.49.html.

[13] J. Kamada, M. Yuhara, and E. Ono. User-level realtime scheduler exploiting kernel-level fixed priority scheduler. In *Multimedia Japan*, Mar 1996.

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

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

[16] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *J. Comput. Sci. & Technol.*, 20(5):654–664, Sep 2005.

[17] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.

[18] J. Liedtke. Improving IPC by kernel design. In *14th SOSP*, pages 175–88, Asheville, NC, USA, Dec 1993.

[19] J. Liedtke. A persistent system in real use: Experience of the first 13 years. In *3rd IWOOOS*, pages 2–11, Asheville, NC, USA, Dec 1993. IEEE.

[20] J. Liedtke. On μ-kernel construction. In *15th SOSP*, pages 237–250, Copper Mountain, CO, USA, Dec 1995.

[21] J. Liedtke. μ-Kernels must and can be small. In *5th IWOOOS*, pages 152–161, Seattle, WA, USA, Oct 1996. IEEE.

[22] J. Liedtke. Towards real microkernels. *CACM*, 39(9):70–77, Sep 1996.

[23] J. W.-S. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

[24] National ICT Australia. *NICTA L4-embedded Kernel Reference Manual Version N1*, Oct 2005. http://ertos.nicta.com.au/Software/systems/kenge/pistachio/refman.pdf.

[25] U. A. Steinberg. Quality assuring scheduling. Diploma thesis, Dresden University of Technology, Mar 2004.

[26] V. Yodaiken. Against priority inheritance. Available at http://www.fsmlabs.com/against-priority-inheritance.html.