From L3 to seL4
What Have We Learnt in 20 Years of L4 Microkernels?

Kevin Elphinstone and Gernot Heiser
NICTA and UNSW, Sydney
{kevin.elphinstone,gernot}@nicta.com.au

Abstract

The L4 microkernel has undergone 20 years of use and evolution. It has an active user and developer community, and there are commercial versions which are deployed on a large scale and in safety-critical systems. In this paper we examine the lessons learnt in those 20 years about microkernel design and implementation. We revisit the L4 design papers, and examine the evolution of design and implementation from the original L4 to the latest generation of L4 kernels, especially seL4, which has pushed the L4 model furthest and was the first OS kernel to undergo a complete formal verification of its implementation as well as a sound analysis of worst-case execution times. We demonstrate that while much has changed, the fundamental principles of minimality and high IPC performance remain the main drivers of design and implementation decisions.

1 Introduction

Twenty years ago, Liedtke [1993a] demonstrated with his L4 kernel that microkernel IPC could be fast, a factor 10–20 faster than other contemporary microkernels.

Microkernels minimize the functionality that is provided by the kernel: the kernel provides a set of general mechanisms, while user-mode servers implement the actual operating system (OS) services [Levin et al., 1975]. User code obtains a system service by communicating with servers via an inter-process communication (IPC) mechanism, typically message passing. Hence, IPC is on the critical path of any service invocation, and low IPC costs are essential.

By the early '90s, IPC performance had become the achilles heel of microkernels: typical cost for a one-way message was around 100 μs, which was too high for building performant systems, with a resulting trend to move core services back into the kernel [Condict et al., 1994]. There were also arguments that high IPC costs were an (inherent?) consequence of the structure of microkernel-based systems [Chen and Bershad, 1993].

In this context, the order-of-magnitude improvement of IPC costs Liedtke demonstrated was quite remarkable. It was followed by work discussing the philosophy and mechanisms of L4 [Liedtke, 1995, 1996], the demonstration of a para-virtualized Linux on L4 with only a few percent overhead [Härtig et al., 1997], the deployment of L4 kernels on billions of mobile devices and in safety-critical systems, and, finally, the world's first functional correctness proof of an OS kernel [Klein et al., 2009]. It also had a strong influence on other research systems, such as Barrelfish [Baumann et al., 2009].

In this paper we examine the development of L4 over the last 20 years. Specifically we look at what makes modern L4 kernels tick, how this relates to Liedtke's original design and implementation, and which of his microkernel "essentials" have passed the test of time. We specifically examine how the lessons of the past have influenced the design of the latest generation of L4 microkernels, exemplified by *seL4* [Klein et al., 2009], but point out where other current L4 versions have made different design decisions.

2 Background

2.1 The L4 Microkernel Family

L4 developed out of an earlier system, called L3, developed by Liedtke [1993b] in the early 1980s on i386 platforms. L3 was a complete OS with built-in persistence, and it already featured user-mode drivers, still a

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522720

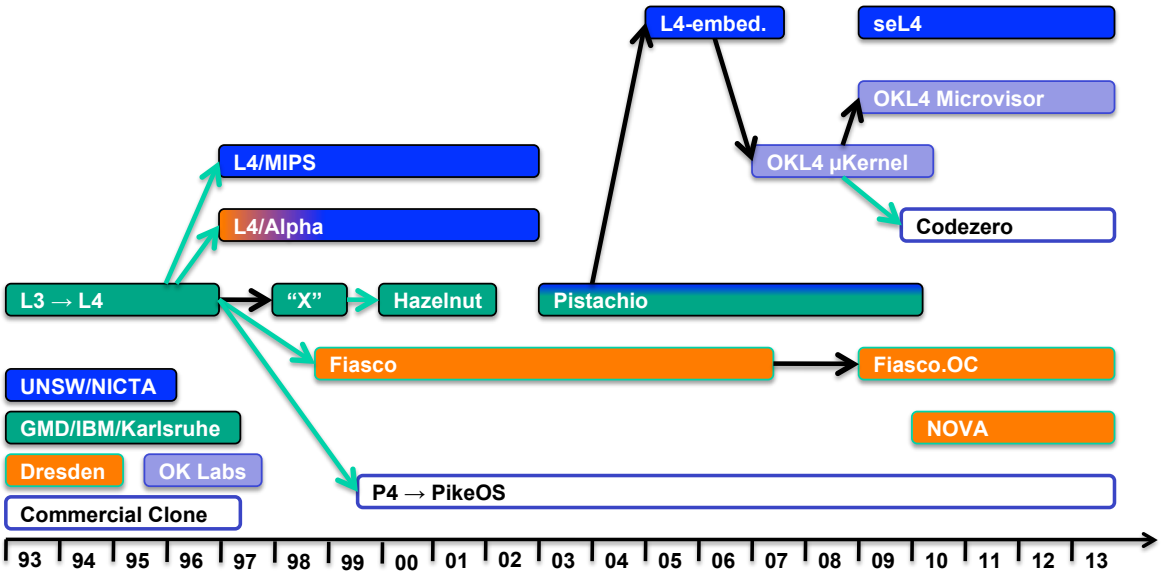

Figure 1: The L4 family tree (simplified). Black arrows indicate code, green arrows ABI inheritance. Box colours indicate origin as per key at the bottom left.

characteristic of L4 microkernels. It was commercially deployed in a few thousand installations (mainly schools and legal practices). Like all microkernels at the time, L3 suffered from IPC costs of the order of 100 μs.

Liedtke initially used L3 to try out new ideas, and what he referred to as "L3" in early publications [Liedtke, 1993a] was actually an interim version of a radical re-design. He first used the name "L4" with the "V2" ABI circulated in the community from 1995.

In the following we refer to this version as the "original L4". Liedtke implemented it completely in assembler on i486-based PCs and soon ported it to the Pentium.

This initial work triggered a twenty-year evolution, with multiple ABI revisions and from-scratch implementations, as depicted in Figure 1. It started with TU Dresden and UNSW re-implementing the ABI (with necessary adaptations) on 64-bit Alpha and MIPS processors, the latter implemented all longer-running operations in C. Both kernels achieved sub-microsecond IPC performance [Liedtke et al., 1997a] and were released as open source. The UNSW Alpha kernel was the first multiprocessor version of L4.

Liedtke, who had moved from GMD to IBM Watson, kept experimenting with the ABI in what became known as *Version X*. GMD and IBM imposed an IP regime which proved too restrictive for other researchers, prompting Dresden to implement a new x86 version from scratch, called *Fiasco* in reference to their experience in trying to deal with IP issues. The open-source Fiasco was the first L4 version written almost completely in a higher-level language (C++) and is the oldest L4 codebase still actively maintained. It was the first L4 kernel with significant commercial use (estimated shipments up to 100,000).

After Liedtke's move to Karlsruhe, he and his students did their own from-scratch implementation (in C), *Hazelnut*, which was the first L4 kernel that was ported (rather than re-implemented) to another architecture (from Pentium to ARM).

Karlsruhe's experience with Version X and Hazelnut resulted in a major ABI revision, V4, aimed at improving kernel and application portability, multi-processor support and addressing various other shortcomings. After Liedtke's tragic death in 2001, his students implemented the design in a new open-source kernel, *L4Ka::Pistachio* ("Pistachio" for short). It was written in C++, originally on x86 and PowerPC, and at UNSW/NICTA we soon after ported it to MIPS, Alpha, 64-bit PowerPC and ARM.[1] In most of these ports, less than 10% of the code changed.

At NICTA we soon re-targeted Pistachio for use in resource-constrained embedded systems, resulting in a fork called *NICTA::Pistachio-embedded* ("L4-embedded"). It saw massive-scale commercially deployment when Qualcomm adopted it as a protected-mode real-time OS for the firmware of their wireless modem processors. The NICTA spinout Open Kernel Labs took on the support and further development of the kernel, re-

[1] There were also Itanium [Gray et al., 2005] and SPARC versions, but they were never completed.

Name	Year	Processor	MHz	Cycles	μs
Original	1993	i486	50	250	5.00
Original	1997	Pentium	160	121	0.75
L4/MIPS	1997	R4700	100	86	0.86
L4/Alpha	1997	21064	433	45	0.10
Hazelnut	2002	Pentium 4	1,400	2,000	1.38
Pistachio	2005	Itanium 2	1,500	36	0.02
OKL4	2007	XScale 255	400	151	0.64
NOVA	2010	Core i7 (Bloomfield) 32-bit	2,660	288	0.11
seL4	2013	Core i7 4770 (Haswell) 32-bit	3,400	301	0.09
seL4	2013	ARM11	532	188	0.35
seL4	2013	Cortex A9	1,000	316	0.32

Table 1: One-way IPC cost of various L4 kernels.

named the *OKL4 microkernel*.[2] Another deployed version is PikeOS, a commercial V2 clone by Sysgo, certified for use in safety-critical avionics and deployed in aircraft and trains.[3]

The influence of EROS [Shapiro et al., 1999] and increasing security focus resulted in a move to capabilities [Dennis and Van Horn, 1966] for access control, first with the 2.1 release of OKL4 (2008) and soon followed by Fiasco (renamed *Fiasco.OC*). Aiming for formal verification, which seemed infeasible for a code base not designed for the purpose, we instead opted for a from-scratch implementation for our capability-based *seL4* kernel.

The last few years also saw two new designs specifically aimed to support virtualization as the primary concern, *NOVA* [Steinberg and Kauer, 2010] from Dresden and the *OKL4 Microvisor* [Heiser and Leslie, 2010] from OK Labs.

A common thread throughout those two decades is the *minimality principle* introduced in Section 3.1, and a strong focus on the performance of the critical IPC operation; kernel implementors generally aim to stay close to the limits set by the micro-architecture, as shown in Table 1. Consequently, the L4 community tends to measure IPC latencies in cycles rather than microseconds, as this better relates to the hardware limits. In fact, Section 3.1 provides an interesting view of the context-switch-friendliness of the hardware (compare the cycle counts for Pentium 4 and Itanium, both from highly-optimised IPC implementations!)

[2]Total deployment is now in the billions, see Open Kernel Labs press release http://www.ok-labs.com/releases/release/ok-labs-software-surpasses-milestone-of-1.5-billion-mobile-device-shipments, January 2012.

[3]See Sysgo press releases http://www.sysgo.com/.

2.2 Modern representatives

We base our examination of the evolution of L4 design and evaluation on seL4, which we know well and which in many ways evolved furthest from the original design. We note where other recent versions ended up with different designs, and try to understand the reasons behind, and what this tells us about the degree of agreement about microkernel design in the L4 community.

Unlike any of the other systems, seL4 is designed from the beginning to support formal reasoning about security and safety, while maintaining the L4 tradition of minimality, performance and the ability to support almost arbitrary system architectures.

This led us to a radically new resource-management model, where all spatial allocation is explicit and directed by user-level code, including kernel memory [Elkaduwe et al., 2008]. It is also the first protected OS kernel in the literature with a complete and sound worst-case execution time (WCET) analysis [Blackham et al., 2011].

A second relevant system is Fiasco.OC, which is unique in that it is a code base that has lived through most of L4 history, starting as a clone of the original ABI (and not even designed for performance). It has, at some point in time, supported many different L4 ABI versions, often concurrently, and is now a high-performance kernel with the characteristics of the latest generation, including capability-based access control. Fiasco served as a testbed for many design explorations, especially with respect to real-time support [Härtig and Roitzsch, 2006].

Then there are two recent from-scratch designs: NOVA [Steinberg and Kauer, 2010], designed for hardware-supported virtualization on x86 platforms, and the OKL4 Microvisor [Heiser and Leslie, 2010] ("OKL4" for short), which was designed as a commercial platform for efficient para-virtualization on ARM processors.

Name	Architecture	Size (kLOC)		
		C/C++	asm	Total
Original	486	0	6.4	6.4
L4/Alpha	Alpha	0	14.2	14.2
L4/MIPS	MIPS64	6.0	4.5	10.5
Hazelnut	x86	10.0	0.8	10.8
Pistachio	x86	22.4	1.4	23.0
L4-embedded	ARMv5	7.6	1.4	9.0
OKL4 3.0	ARMv6	15.0	0.0	15.0
Fiasco.OC	x86	36.2	1.1	37.6
seL4	ARMv6	9.7	0.5	10.2

Table 2: Source lines of code (SLOC) of various L4 kernels.

3 Microkernel Design

Liedtke [1995] outlines principles and mechanisms which drove the design of the original L4. We examine how these evolved over time, and, specifically, how they compare with the current generation.

3.1 Minimality

The main drivers of Liedtke's designs were minimality and IPC performance, with a strong belief that the former helps the latter. Specifically, he formulated the microkernel *minimality principle*:

> A concept is tolerated inside the μ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality [Liedtke, 1995].

This principle, which is a more pointed formulation of "only minimal mechanisms and no policy in the kernel," has continued to be the foundation of the design of L4 microkernels. The discussion in the following sections will demonstrate the community's on-going efforts to remove features or replace them with more general (and powerful) ones.

The adherence to this principle can be seen from the comparison of source code sizes, shown in Table 2: while it is normal for systems to grow in size over time, seL4, the latest member of the family (and, arguably, the one that diverged strongest from the traditional model) is still essentially the same size as the early versions.[4] Verification provided a particular strong motivation for minimality, as even 9,000 SLOC pushed the limits of what was achievable.

Retained: Minimality as key design principle.

Nevertheless, none of the designers of L4 kernels to date claim that they have developed a "pure" microkernel in the sense of strict adherence to the minimality principle. For example, all of them have a scheduler in the kernel, which implements a particular scheduling policy (usually hard-priority round-robin). To date, noone has come up with a truly general in-kernel scheduler or a workable mechanism which would delegate all scheduling policy to user-level without imposing high overhead.

3.2 IPC

We mentioned earlier the importance of IPC performance, and that the design and implementation of L4 kernels consistently aimed at maximising it. However, the details have evolved considerably.

3.2.1 Synchronous IPC

The original L4 supported synchronous (rendezvous-style) IPC as the only communication, synchronisation, and signalling mechanism. Synchronous IPC avoids buffering in the kernel and the management and copying cost associated with it. It is also a prerequisite for a number of implementation tricks we will cover later, specifically the lazy scheduling (Section 4.2), direct process switch (Section 4.3), and temporary mapping (Section 3.2.2) optimisations.

While certainly minimal, and simple conceptually and in implementation, experience taught us significant drawbacks of this model: it forces a multi-threaded design onto otherwise simple systems, with the resulting synchronisation complexities. For example, the lack of functionality similar to UNIX `select()` required separate threads per interrupt source, and a single-threaded server could not wait for client requests and interrupts at the same time.

We addressed this in L4-embedded by adding *asynchronous notifications*, a very simple form of asynchronous IPC. We later refined this model in seL4 as *asynchronous endpoints* (AEPs, endpoints will be explained in Section 3.2.3): sending is non-blocking and asynchronous to the receiver, who has a choice of blocking or polling for a message. Logically, an asynchronous endpoint is similar to multiple binary semaphores allocated within a single word: Each AEP has a single-word *notification field*. A send operation specifies a mask of

[4] In fact, seL4's SLOC count is somewhat bloated as a consequence of the C code being mostly a "blind" manual translation from Haskell [Klein et al., 2009], together with generated bit-field accessor functions, resulting in hundreds of small functions. The kernel compiles into about 9 k ARM instructions.

bits (usually a single bit), which is OR-ed into the notification field. Waiting is effectively `select()` across the notification field.

In our present design, an asynchronous endpoint can be *bound* to a specific thread. If a notification arrives while the thread is waiting on a synchronous endpoint, the notification is delivered like a synchronous message (with an indication that it is really a notification).

In summary, seL4, like most other L4 kernels, retains the model of synchronous IPC but augments it with asynchronous notification. OKL4 has completely abandoned synchronous IPC, and replaced it by *virtual IRQs* (essentially asynchronous notifications). NOVA has augmented synchronous IPC with counting semaphores [Steinberg and Kauer, 2010], while Fiasco.OC has also augmented synchronous IPC with virtual IRQs.

Replaced: Synchronous IPC augmented with (seL4, NOVA, Fiasco.OC) or replaced by (OKL4) asynchronous notification.

Having two forms of IPC is a violation of minimality, so the OKL4 approach is more pure in this particular point (although its channel abstraction constitutes a different violation, see Section 3.2.2). Furthermore, the utility of synchronous IPC becomes more dubious in a multicore context: an RPC-like server invocation sequentialises client and server, which should be avoided if they are running on separate cores. We therefore expect communication protocols based on asynchronous notification to become more prevalent, and a move to asynchronous-only IPC sometime in the future remains a possibility.

3.2.2 IPC message structure

Original L4 IPC had rich semantics. Besides in-register ("short") messages it supported messages of almost arbitrary size (a word-aligned "buffer" as well as multiple unaligned "strings") in addition to in-register arguments. Coupled with the all-synchronous design, this approach avoids redundant copying.

Register arguments support zero-copy: the kernel always initiates the IPC from the sender's context and switches to the receiver's context without touching the message registers.

A drawback is the architecture-dependent and (especially on x86-32) small size of zero-copy messages. In fact, the number of available registers changed frequently with ABI changes, as changes to syscall arguments used or freed up registers.

Pistachio introduced the concept of a moderate-size (configurable in the range of 16–64) set of *virtual message registers*. The implementation mapped some of them to physical registers, the rest was contained in a per-thread pinned part of the address space. The pinning ensures register-like semantics without the possibility of a page fault. Inlined access functions hide the distinction between physical and memory-backed registers from the user. seL4 and Fiasco.OC continue to use this approach.

The motivation is two-fold: Virtual message registers improve portability across architectures, and more importantly, they reduce the performance penalty for moderately-sized messages exceeding the number of physical registers: copying a small number of words is cheaper than establishing the temporary mapping involved in "long" IPC, as described below.

The benefits of in-register message transfers has diminished over time, as the architectural costs of context switching dominate IPC performance. For example, in-register message transfer on the ARM11 improves IPC performance by 10% (for a 4-word message) compared to passing via the kernel stack; on Cortex A9 this reduces to 4%. On x86-32, reserving any registers for message passing is detrimental to the compiler's ability to optimise the code.

Replaced: Physical by virtual message registers.

"Long" messages could specify multiple buffers in a single IPC invocation to amortise the hardware mode- and context-switch costs. Long messages could be delivered with a single copy: executing in the sender's context, the kernel sets up a temporarily mapped window into the receiver's address space, covering (parts of) the message destination, and copies directly to the receiver.

This could trigger a page fault during copying in either the source or destination address space, which required the kernel to handle nested exceptions. Furthermore, the handling of such an exception required invoking a user-level page-fault handler. The handler had to be invoked while the kernel was still handling the IPC system call, yet the invocation had to pretend that the fault happened during normal user-mode execution. On return, the original system-call context had to be re-established. The result was significant kernel complexity, with many tricky corner cases that risked bugs in the implementation.

While long IPC provides functionality which cannot be emulated without some overhead, in practice it was rarely used: Shared buffers can avoid any explicit copying between address spaces, and are generally preferred for bulk data transfer. Additionally, asynchronous interfaces can be used for batching of transfers without resorting to explicit batching support in the kernel.

The main use of long IPC was for legacy POSIX read-write interfaces to servers, which require transferring the contents of arbitrary buffers to servers who do not necessarily have access to the client's memory. However, the

rise of Linux as POSIX middleware, where Linux effectively shares memory with its applications, replaced this common use case with pass-by-reference. The remaining use cases either had interface flexibility or could be implemented with shared memory. Long IPC also violates the minimality principle (which talks about functionality, not performance).

As a consequence of this kernel complexity and the existence of user-level alternatives, we removed long IPC from L4-embedded, and NOVA and Fiasco.OC do not provide it either.

For seL4 there are even stronger reasons for staying away from supporting long messages: the formal verification approach explicitly avoided any concurrency in the kernel [Klein et al., 2009], and nested exceptions introduce a degree of concurrency. They also break the semantics of the C language by introducing additional control flow. While it is theoretically possible to formally reason about nested exceptions, it would make the already challenging verification task even harder. Of course, the in-kernel page faults could be avoided with extra checking, but that would introduce yet more complexity (besides penalising best-case performance), and would still require a more complex formal model to prove checking is complete and correct.

Abandoned: Long IPC.

OKL4 diverges at this point, by providing a new, asynchronous, single-copy, bulk-transfer mechanism called *channel* [Heiser and Leslie, 2010]. However, this is really a compromise retained for compatibility with the earlier OKL4 microkernel, which was aimed at memory-starved embedded systems. It was used to retrofit protection boundaries into a highly multithreaded (> 50 threads) real-time application, where a separate communication page per pair of communicating threads was too costly.

3.2.3 IPC destinations

Original L4 had threads as the targets of IPC operations. The motivation was to avoid the cache and TLB pollution associated with a level of indirection, although Liedtke [1993a] notes that ports could be implemented with an overhead of 12% (mostly 2 extra TLB misses). The model required that thread IDs were unique identifiers.

This model has a drawback of poor information hiding. A multi-threaded server has to expose its internal structure to clients, in order to spread client load, or use a gateway thread, which could become a bottleneck and would impose additional communication and synchronisation overhead. There were a number of proposals to mitigate this but they all had drawbacks. Additionally, large-page support in modern CPUs has reduced the TLB pollution of indirection by increasing the likelihood of co-location on the same page. Last but not least, the global IDs introduced covert channels [Shapiro, 2003].

Influenced by EROS [Shapiro et al., 1999], *IPC endpoints* were adopted as IPC destinations by seL4 and Fiasco.OC [Lackorzynski and Warg, 2009]). seL4 (synchronous) endpoints are essentially ports: the root of the queue of pending senders or receivers is a now a separate kernel object, instead of being part of the recipient's thread control block (TCB). Unlike Mach ports, IPC endpoints do not provide any buffering.

Replaced: Thread IDs by port-like IPC endpoints as message destinations.

3.2.4 IPC timeouts

A blocking IPC mechanism creates opportunities for denial-of-service (DOS) attacks. For example, a malicious (or buggy) client could send a request to a server without ever attempting to collect the reply; owing to the rendezvous-style IPC, the sender would block indefinitely unless it implements a watchdog to abort and restart. L4's long IPC enables a slightly more sophisticated attack: A malicious client could send a long message to a server, ensure that it would page fault, and prevent its pager from servicing the fault.

To protect against such attacks, IPC operation in the original L4 had timeouts. Specifically, an IPC syscall specified 4 timeouts: one to limit blocking until start of the send phase, one to limit blocking in the receive phase, and two more to limit blocking on page faults during the send and receive phases (of long IPC).

Timeout values were encoded in a floating-point format that supported the values of zero, infinity, and finite values ranging from one millisecond to weeks. They added complexity for managing wakeup lists.

Practically, however, timeouts were of little use as a DOS defence. There is no theory, or even good heuristics, for choosing timeout values in a non-trivial system, and in reality, only the values zero and infinity were used: A client sends and receives with infinite timeouts, while a server waits for a request with an infinite but replies with a zero timeout. (The client uses an RPC-style `call` operation, consisting of a send followed by an atomic switch to a receive phase, guaranteeing that the client is ready to receive the server's reply.) Traditional watchdog timers represent a better approach to detecting unresponsive IPC interactions (e.g. resulting from deadlocks).

Having abandoned long IPC, in L4-embedded we replaced timeouts by a single flag supporting a choice

of polling (zero timeout) or blocking (infinite timeout). Only two flags are needed, one for the send and one for the receive phase. seL4 follows this model. A fully-asynchronous model, such as that of OKL4, is incompatible with timeouts and has no DOS issues that would require them.

Timeouts could also be used for timed sleeps by waiting on a message from a non-existing thread, a feature useful in real-time system. Dresden experimented with extensions, including absolute timeouts, which expire at a particular wall clock time rather than relative to the commencement of the system call. Our approach is to give userland access to a (physical or virtual) timer.

Abandoned: IPC timeouts in seL4, OKL4.

3.2.5 Communication Control

In the original L4, the kernel delivered the sender's unforgeable ID to the receiver. This allows a server to implement a form of discretionary access control, by ignoring undesirable messages. However, a server can be bombarded with spurious large messages by malicious clients. The time consumed by receiving such messages (even if copying is done by the kernel) prevents the server from performing useful work, and checking which ones to discard also costs time. Hence such messages can constitute a DOS attack, which can only be avoided by kernel support that prevents undesirable messages being sent in the first place [Liedtke et al., 1997b]. Mandatory access control policies also require a mechanism for mediating and authorising communication.

Original L4 provided this through a mechanism called *clans and chiefs*: Processes were organised in a hierarchy of "clans", each of which had a designated "chief". Inside the clan, all messages are transferred freely and the kernel guarantees message integrity. But messages crossing a clan boundary, whether outgoing or incoming, are redirected to the clan's chief, who can thus control the flow of messages. The mechanism also supports *confinement* [Lipner, 1975] of untrusted subsystems.

Liedtke [1995] argued that the clans-and-chiefs model only added two cycles per IPC operation, as clan IDs were encoded in thread IDs for quick comparison. However, the low overhead only applies where direct communication is possible. Once messages get re-directed, each such re-direction adds two messages to a (logically) single round-trip IPC, a significant overhead. Furthermore, the strict thread hierarchy was unwieldy in practice (and was probably the feature most cursed by people trying to build L4-based systems). For mandatory access control, the model quickly deteriorated into a chief per process. It is a prime example of kernel-enforced policy (address-space hierarchy) limiting the design space.

As a consequence of these drawbacks, many L4 implementations did not implement clans and chiefs (or disabled the feature at build time), but that meant that there was no way to control IPC. There were experiments with models based on a more general form of IPC redirection [Jaeger et al., 1999], but these failed to gain traction. The problem was finally resolved with flexible capability-mediated access control to endpoints.

Abandoned: Clans and chiefs.

3.3 User-level device drivers

A key consequence of the minimality principle, and maybe the most radical novelty of L4 (or, rather, its predecessor, L3 [Liedtke et al., 1991]) was to make all device drivers user-level processes.[5] This is still a hallmark of all L4 kernels, and verification is a strong motivator for sticking with the approach: adding any unverified code, such as drivers, into the kernel would obliterate any guarantees, and verifying the large amount of driver code in real-world systems is out of reach for now.

A small number of drivers are still best kept in the kernel. In a modern L4 kernel this typically means a timer driver, used for preempting user processes at the end of their time slice, and a driver for the interrupt controller, which is required to safely distribute interrupts to user-level processes.

The user-level driver model is tightly coupled with modelling interrupts as IPC messages, which the kernel sends to the driver. Details of the model (IPC from a virtual thread vs upcall), as well as the association and acknowledgment protocol, have changed over the years (and at times changed back and back again) but the general approach still applies.

The most notable change was moving from synchronous to asynchronous IPC for interrupt delivery. This was driven by implementation simplification, as synchronous delivery required the emulation of virtual in-kernel threads as the sources of interrupt IPC.

User-level drivers have benefited from virtualisation-driven hardware improvements. I/O memory management units (IOMMUs) have enabled safe pass-through device access for drivers. User-level drivers have also benefited from hardware developments that reduce interrupt overheads, specifically interrupt coalescing support on modern network interfaces. In the x86 world, they have profited from dramatically decreased context-switching costs enabled by TLB tagging (among others).

Retained: User-level drivers as a core feature.

[5] Strictly speaking, this had been done before, in the Michigan Terminal system [Alexander, 1972] and the Monads OS [Keedy, 1979], but those designs had little direct impact on later ones and there is no information about performance.

Of course, user-level drivers have now become mainstream. They are supported (if not encouraged) on Linux, Windows and MacOS. Overheads in those systems are generally higher than in L4 with its highly optimised IPC, but we have shown in the past that low overheads are achievable even on Linux, at least on context-switch friendly hardware [Leslie et al., 2005]. In practice, though, only a tiny fraction of devices are performance-critical.

3.4 Resource management

Original L4's resource management, like its approach to communication control, was heavily based on process hierarchies. This applied to managing processes as well as virtual memory. Hierarchies are an effective way of managing and recovering resources, and provide a model of constraining sub-systems (where system mechanisms restrict children's privileges to be a subset of their parent's), but the cost is rigidity. However, the hierarchies are a form of policy, and as such a bad match for a microkernel (as discussed in Section 3.2.5).

Capabilities can provide a way out of the constraints of the hierarchy, which is one of several reasons all modern L4 kernels adopted capability-based access-control. Here we examine the most important resource-management issues arising form the original L4 model, and how we deal with them now.

3.4.1 Process hierarchy

A process (in L4 this is essentially a page table and a number of associated threads) consumes kernel resources, and unchecked allocation of TCBs and page tables could easily lead to denial of service. Original L4 dealt with that through a process hierarchy: "Task IDs" were essentially capabilities over address spaces, allowing creation and deletion.

There was a finite number of them (of the order of thousands), which the kernel handed out first-come, first-served. They could be delegated, but only up or down the hierarchy. (They were also closely tied to the thread hierarchy used for IPC control, see Section 3.2.5.) In a typical setup, the initial user process would grab all task IDs before creating any further processes.

Perhaps unsurprisingly, this model proved inflexible and restrictive; it was eventually replaced by fully-fledged capabilities.

Abandoned: Hierarchical process management.

3.4.2 Recursive page mappings

Original L4 tied authority over physical memory frames to existing page mappings. Having a valid mapping (of a frame) in its address space gave a process the right to map this page into another address space. Instead of mapping, a process could *grant* one of its pages, which removed the page (and any authority over it) from the grantor. A mapping (but not a grant) could be revoked by an *unmap* operation. Address spaces were created empty, and were populated using the mapping primitive.

The recursive mapping model was anchored in a primordial address space σ_0, which received a (one-on-one) mapping of all free frames left over after the kernel booted. σ_0 was the page-fault handler of all processes created at boot time, and would map each of its pages *once* to the first process that requested it (by faulting on an address in the page).

Note that, while the L4 memory model creates a hierarchy of mappings originating from each frame, it does not force a hierarchical view of address spaces: Mappings were established through IPC (similar to transferring a capability through an IPC message), and a process could map one of its pages to any other process it was allowed to send IPC to (provided the recipient agreed to receive mappings). Compared to Mach, L4 has no memory object semantics, only low-level address space management mechanisms that are closer to Mach's in-kernel `pmap` interface than its user-visible memory object abstraction [Rashid et al., 1988]. Memory objects, copy-on-write, and shadow-chains are all user-level created abstractions or implementation approaches.

The recursive mapping model was conceptually simple and elegant, and Liedtke was clearly proud of it – it figured prominently in many papers, including the first [Liedtke, 1993a], and in all his presentations. Yet, experience showed that there were significant drawbacks.

In order to support revocation at page granularity, the recursive address-space model requires substantial bookkeeping in the form of a *mapping database*. Moreover, the generality of the L4 memory model allows two colluding processes to force the kernel to consume large amounts of memory by recursively mapping the same frame to different pages in each other's address space, a potential DOS attack especially on 64-bit hardware, which can only prevented by controlling IPC (via the dreaded clans-and-chiefs).

In L4-embedded we removed the recursive mapping model, after observing that for our real-world use cases, 25–50% of kernel memory use was consumed by the mapping database even without malicious processes. We replaced it by a model that more closely mirrors hardware, where mappings always originate from ranges of physical memory frames.

This approach comes at the expense of losing fine-grained delegation and revocation of memory (other than by brute-force scans of page tables), we therefore only considered it an interim pain relief. OKL4 somewhat

extends this minimal model, without achieving the generality and fine-grained control of the original model.

Mapping control is, of course, easily achieved in a capability-based system, using a variant of the standard grant-take model [Lipton and Snyder, 1977]. This is what seL4 does: the right to map is conveyed by a capability to a physical frame, not by having access to a virtual page backed by that frame, and thus seL4's model is not recursive. Even with a frame capability, mapping is strictly limited by the explicit kernel memory model used to bookkeep the mappings, as described below.

Xen provides an interesting point of comparison. *Grant tables* allow the creation (based on possession of a valid mapping) of what is effectively a frame capability, which can be passed to another domain to establish shared mappings [Fraser et al., 2004]. A recent proposal extends grant tables to allow for revocation of frames [Ram et al., 2010]. The semantics of the memory mapping primitives is loosely similar to that of seL4, minus the propagation of page faults. In Xen's case, the overhead for supporting fine-grained delegation and revocation is only paid in instances of sharing.

NOVA and Fiasco.OC both retain the recursive address space model, with authority to map determined by possession of a valid mapping. The consequent inability to restrict mapping and thus book keeping allocation is addressed by per-task kernel memory pools in Fiasco.OC.

The existing L4 address space models (specifically fine-grained delegation and revocation) represent different trade-offs between generality and minimality of mechanism, and potentially more space-efficient domain-specific approaches.

Multiple approaches: Some L4 kernels retain the model of recursive address-space construction, while seL4 and OKL4 originate mappings from frames.

3.4.3 Kernel memory

While capabilities provide a clean and elegant model for delegation, by themselves they do not solve the problem of resource management. A single malicious thread with grant right on a mapping can still use this to create a large number of mappings, forcing the kernel to consume large amounts of memory for meta-data, and potentially DOS-ing the system.

L4 kernels traditionally had a fixed-size heap from which the kernel allocated memory for its data structures. Original L4 had a *kernel pager*, called σ_1, through which the kernel could request additional memory from userland. This does not solve the problem of malicious (or buggy) user code forcing unreasonable memory consumption, it only shifts the problem. Consequently, σ_1 was not supported by most L4 kernels.

The fundamental problem, shared by most other OSes, is the insufficient isolation of user processes through the shared kernel heap. A satisfactory approach must be able to provide complete isolation. The underlying issue is that, even in a capability system, where authority is represented by capabilities, it is not possible to reason about the security state if there are resources outside the capability system.

Kernels that manage memory as a cache of user-level content only partially address this problem. While caching-based approaches remove the opportunity for DOS attacks based on memory exhaustion, they do not enable the strict isolation of kernel memory that is a prerequisite for performance isolation or real-time systems, and potentially introduce covert channels.

Liedtke et al. [1997b] examined this issue and proposed per-process kernel heaps together with a mechanism to donate extra memory to the kernel on exhaustion. NOVA, Fiasco and OKL4 all adopted variations of this approach. Per-process kernel heaps simplify user level (by removing control of allocation) at the expense of the ability to revoke allocations without destroying the process, and the ability to reason directly about allocated memory (as opposed to just bounding it). The trade-off is still being explored in the community.

We took a substantially different approach with seL4; its model for managing kernel memory is seL4's main contribution to OS design. Motivated by the desire to reason about resource usage and isolation, we subject *all* kernel memory to authority conveyed by capabilities (except for the fixed amount used by the kernel to boot up, including its strictly bounded stack). Specifically, we completely remove the kernel heap, and provide userland with a mechanism to identify authorised kernel memory whenever the kernel allocates data structures. A side-effect is that this reduces the size and complexity of the kernel, a major bonus to verification.

The key is *making all kernel objects explicit* and subject to capability-based access control. This approach is inspired by hardware-based capability systems, specifically CAP [Needham and Walker, 1977] where hardware-interpreted capabilities directly refer to memory. HiStar [Zeldovich et al., 2011] also makes all kernel objects explicit, though it takes a caching approach to memory management.

Of course, user-visible kernel objects do not mean that someone with authority over a kernel object can directly read or write it. The capability provides the right to invoke (a subset of) object-specific methods, which includes destruction of the object. (Objects, once created, never change their size.) Crucially, the kernel object types include unused memory, called *Untyped* in seL4,

Object	Description
TCB	Thread control block
Cnode	Capability storage
Synchronous Endpoint	Port-like rendezvous object for synchronous IPC
Asynchronous Endpoint	Port-like object for asynchronous notification.
Page Directory	Top-level page table for ARM and IA-32 virtual memory
Page Table	Leaf page table for ARM and IA-32 virtual memory
Frame	4 KiB, 64 KiB, 1 MiB and 16 MiB objects that can be mapped by page tables to form virtual memory
Untyped Memory	Power-of-2 region of physical memory from which other kernel objects can be allocated

Table 3: seL4 kernel objects.

which can be used to create other objects.

Specifically, the only operation possible on *Untyped* is to *retype* part of it into some object type. The relationship of the new object to the original *Untyped* is recorded in a *capability derivation tree*, which also records other kinds of capability derivation, such as the creation of capability copies (with potentially reduced privileges). Once some *Untyped* has been retyped, the only operation possible on the (corresponding part of) the original *Untyped* is to revoke the derived object (see below).

Retyping is the only way to create objects in seL4. Hence, by limiting access to *Untyped* memory, a system can control resource allocation. Retyping can also produce smaller *Untyped* objects, which can then be independently managed – this is key to delegating resource management. The derivation from *Untyped* also ensures the kernel integrity property that no two typed objects overlap.

Table 3 gives the complete set of seL4 object types and their use. Userland can only directly access (load/store/fetch) memory corresponding to a *Frame* that is mapped in its address space (by inserting the *Frame* capability into a *Page Table*).

The resulting model has the following properties:

1. All authority is explicitly conferred (via capabilities).

2. Data access and authority can be confined.

3. The kernel itself (for its own data structures) adheres to the authority distributed to applications, including the consumption of physical memory.

4. Each kernel object can be reclaimed independently of any other kernel objects.

5. All operations execute, or are preemptible, in "short" time (constant or linear in the size of an object no bigger than a page).

Properties 1–3 ensure that it is possible to reason about system resources as well as security. Especially Property 3 was crucial to formally proving the kernel's ability to ensure integrity, authority confinement and confidentiality [Murray et al., 2013; Sewell et al., 2011]. Property 5 ensures that all kernel latencies are bounded and thus supports its use for hard real-time systems [Blackham et al., 2011].

Property 4 ensures kernel integrity. Any holder of an appropriate capability can reclaim an object at any time (making the original *Untyped* again available for object creation). For example, page-table memory can be reclaimed without having to destroy the corresponding address space. This requires that the kernel is able to detect (and invalidate) any references to an object that is being reclaimed.

The requirement is satisfied with the help of the capability derivation tree. Objects are revoked by invoking the `revoke()` method on a *Untyped* object further up the tree; this will remove all capabilities to all objects derived from that *Untyped*. When the last capability to an object is removed, the object itself is deleted. This removes any in-kernel dependencies it may have with other objects, thus making it available for re-use. Removal of the last capability is easy to detect, as it cleans up the last leaf node in the capability tree referring to a particular memory location.

Revocation requires user-level book-keeping to associate *Untyped* capabilities with objects, often at the granularity of higher-level abstractions (such as processes) defined at user level. The precise semantics of *Untyped* and its relationship to user-level book-keeping is still being explored.

Added: User-level control over kernel memory in seL4, kernel memory quota in Fiasco.OC.

3.4.4 Time

Apart from memory, the other key resource that must be shared in a system is the CPU. Unlike memory, which can be sub-divided and effectively shared between multiple processes concurrently, the CPU can only be used by a single thread at a time, and must therefore be time-multiplexed.

All versions of L4 have achieved this multiplexing through a fixed-policy scheduler (pluggable in Fiasco.OC). The scheduling model of the original L4,

hard-priority round-robin, is still alive, despite being a gross heresy against the core microkernel religion of policy-freedom. All past attempts to export scheduling policy to user level have failed, generally due to intolerable overheads, or were incomplete or domain-specific.

Especially the Dresden group, which has a focus on real-time issues, experimented extensively with time issues, including absolute timeouts (see Section 3.2.4). They also explored several approaches to scheduling, as well as system structures suitable for real-time and analysed L4-based real time systems [Härtig and Roitzsch, 2006].

While able to address some specific problems, Dresden did not develop a policy-free and universal mechanism, and Fiasco.OC reverted to essentially the traditional L4 model. A more recent proposal for *scheduling contexts* allows mapping of hierarchical priority-based schedules onto a single priority scheduler [Lackorzynski et al., 2012]. While promising, this is not yet able to deal with the breadth of scheduling approaches used in the real-time community, especially earliest-deadline-first (EDF) scheduling.

One might argue that the notion of a single, general-purpose kernel suitable for all purposes may not be as relevant as it once was – these days we are used to environment-specific plugins. However, the formal verification of seL4 creates a powerful disincentive to changing the kernel, it strongly reinforces the desire to have a single platform for all usage scenarios. Hence, a policy-free approach to dealing with time is as desirable as it has ever been.

Unresolved: Principled, policy-free control of CPU time.

3.4.5 Multicore

Multiprocessor issues have been explored early-on in the L4 community. Most of the work, L4/Alpha and L4/MIPS notwithstanding, was done on x86 platforms, which were the earliest affordable multiprocessors. Early x86 multiprocessors and multicores had high inter-core communication cost and no shared caches. Consequently, the standard approach was to use per-processor scheduling queues (and minimal sharing of kernel data across cores), and thread migration only happening on explicit request by userland. Uhlig [2005] explored locking, synchronisation and consistency issues on platforms with many cores, and developed approaches for scalable concurrency control of kernel data structures based on RCU [McKenney et al., 2002]. NOVA and Fiasco.OC make extensive use of RCU.

With the shift of emphasis from high-end server to embedded and real-time platforms, multiprocessor issues took a back stage, and were only revived recently with the advent of multicore versions of embedded processors. These are characterised by low inter-core communication cost and usually shared L2 caches, implying tradeoffs which differ from those on x86. The resulting low migration costs do typically not justify the overhead of a system call for migrating threads, and a global scheduler makes more sense.

Here, verification introduces new constraints. As discussed in Section 3.2.2, concurrency presents huge challenges for verification, and we kept it out of the kernel as much as possible. For multicores this means adopting either a big kernel lock or a multikernel approach [Baumann et al., 2009]. For a microkernel, where system calls are short, the former is not as silly as it may seem at first, as lock contention will be low, at least for a small number of cores sharing an L2.

The approach we are exploring at present is a *clustered multikernel*, a hybrid of a big-lock kernel (across cores which share an L2 cache), and a restricted variant of a multikernel (no memory migration is permitted between kernels) [von Tessin, 2012]. The clustered multikernel avoids concurrency in the majority of kernel code, which enables some of the formal guarantees to continue hold under some assumptions. The most significant assumptions are that (1) the lock itself, and any code outside of it, is correct and race free, and that (2) the kernel is robust to any concurrent changes to memory shared between the kernel and user-level (for seL4 is this is only a block of virtual IPC message registers).

The attraction of this approach is that it retains the existing uniprocessor proof with only small modifications. We have formally lifted a parallel composition of the uniprocessor automatons and shown that refinement still holds. The disadvantage is that the formal guarantees no longer cover the entire kernel, and the large-step semantics used by the lifting framework preclude further extension of the formal framework to cover reasoning about the correctness of the lock, user-kernel concurrency, and any relaxation of resource migration restrictions.

A variation of a clustered multikernel may eventually be the best approach to obtaining full formal verification of a multiprocessor kernel, though we make no strong representations here. Much more work is required on the formal side to reason about fine-grained interleaving at the scale of a microkernel.

Unresolved: Handling of multicore processors in the age of verification.

4 Microkernel Implementation

Liedtke [1993a] list a set of design decisions and implementation tricks which helped making IPC fast in the

original i486 version, although a number of them smell of premature optimisation.

Some have already been mentioned, such as the temporary mapping window used in the (now obsolete) long IPC. Others are uncontroversial, such as the send-receive combinations in a single system call (the client-style `call` for an RPC-like invocation and the server-style `reply-and-wait`). We will discuss the remaining in more detail, including some traditional L4 implementation approaches which were less-publicised but long taken for granted in the community.

4.1 Strict process orientation and virtual TCB array

The original L4 had a separate kernel stack for each thread, allocated above its TCB on the same page. The TCB's base address was therefore a fixed offset from the stack base, and could be obtained by masking the least significant bits off the kernel stack pointer. Only a single TLB entry was required to cover both a thread's TCB and stack.

Furthermore, all TCBs were allocated in a sparse, virtually-addressed array, indexed by thread ID. During IPC, this enables a very fast lookup of the destination TCB, without first checking the validity of the ID: If the caller supplies an invalid ID, the lookup may access an unmapped TCB, triggering a page fault, which the kernel handles by aborting the IPC. If no fault happened, the validity of the thread ID can be established by comparing the caller-supplied value with the one found in the TCB. (Original L4's thread IDs had version numbers, which changed when the thread was destroyed and re-created. This was done to make thread IDs unique in time. Recording the current ID in the TCB allowed detecting stale thread IDs.)

Both features come at a cost: The many kernel stacks dominate the per-thread memory overhead, and they also increase the kernel's cache footprint. The virtual TCB array increases the kernel's virtual memory use and thus the TLB footprint, but avoids the additional cache footprint for the lookup table that would otherwise be required. Processors with a single page size and untagged TLBs left little opportunity to optimise beyond grouping data structures to minimise the number of pages touched. However, RISC processors had large-page sizes (or physical memory addressing) and tagged TLBs which changed the trade-offs.

The kernel's memory use became a significant issue when L4 was gaining traction in the embedded space, so the design needed revisiting.

Initial experiments with a single-stack kernel on a Pentium showed a reduction in kernel memory consumption, and improvements in IPC performance on micro-benchmarks [Haeberlen, 2003]. Warton [2005] performed a thorough performance evaluation of the Pistachio process kernel vs an event-based (single-stack) kernel with continuations on an ARMv5 processor. He demonstrated comparable performance (generally within 1%) on micro-benchmarks, but a 20% performance advantage of the event kernel on a multi-tasking workload (AIM7). He also found that the event kernel's per-thread memory use was a quarter of that of the process kernel, despite the event kernel requiring more than twice the TCB size of the process kernel (to store the continuations).

Concurrently, Nourai [2005] analysed the trade-offs of virtual vs physical addressing of TCBs. He implemented physical addressing, also in Pistachio, although on a MIPS64 processor. He found little if any differences in IPC performance in micro-benchmarks, but significantly better performance of the physically-addressed kernel on workloads that stressed the TLB. MIPS is somewhat anomalous in that it supports physical addressing even with the MMU enabled, while on most other architectures "physical" addressing is simulated by idempotent large-page mappings, potentially in conjunction with "global" mappings. Still Nourai's results convincingly indicate that there is no significant performance benefit from the virtually-addressed TCBs.

An event-based kernel that avoids in-kernel page-fault exceptions preserves the semantics of the C language. As discussed earlier in Section 3.2.2, remaining within the semantics of C reduces the complexity of verification.

Together, these results made us choose an event-based design with physically-addressed kernel data for L4-embedded, and seL4 followed suit. While this decision was driven initially by the realities of resource-starved embedded systems and later the needs of verification, the approach's benefits are not restricted to those contexts, and we believe it is generally the best approach on modern hardware.

Replaced: Process kernel by event kernel in seL4, OKL4 and NOVA.

Abandoned: Virtual TCB addressing.

4.2 Lazy scheduling

In the rendezvous model of IPC, a thread's state frequently alternates between runnable and blocked. This implies frequent queue manipulations, moving a thread into and out of the ready queue, often many times within a time slice.

Liedtke's *lazy scheduling* trick minimises these queue manipulations: When a thread blocks on an IPC operation, the kernel updates its state in the TCB but leaves

the thread in the ready queue, with the expectation it will unblock soon. When the scheduler is invoked upon a time-slice preemption, it traverses the ready queue until it finds a thread that is really runnable, and removes the ones that are not. The approach was complemented by lazy updates of wakeup queues.

Lazy scheduling moves work from a high-frequency operation to the less frequently invoked scheduler. We observed the drawback when analysing seL4's worst-case execution time (WCET) for enabling its use in hard real-time systems [Blackham et al., 2012]: The execution time of the scheduler is only bounded by the number of threads in the system!

To address the issue, we adopted an alternative optimisation, referred to as *Benno scheduling*, which does not suffer from pathological timing behaviour: Here, the ready queue contains all runnable threads *except* the currently executing one. Hence, the ready queue usually does not get modified when threads block or unblock during IPC. At preemption time, the kernel inserts the (still runnable but no longer executing) preempted thread into the ready queue. The removal of timeouts means that there are no more wakeup queues to manipulate. Endpoint wait queues must be strictly maintained, but in the common case (of a server responding to client requests received via a single endpoint) they are hot in cache, so the cost of those queue manipulations is low. This approach has similar average-case performance as lazy scheduling, while also having a bounded WCET.

Replaced: Lazy scheduling by Benno scheduling.

4.3 Direct process switch

L4 traditionally tries to avoid running the scheduler during IPC. If a thread gets blocked during an IPC call, the kernel switches to a readily-identifiable runnable thread, which then executes on the original thread's time slice, generally ignoring priorities. This approach is called *direct process switch*.

It makes more sense than one might think at first, especially when assuming that servers have at least the same priority as clients. On the one hand, if a (client) thread performs a `call` operation (to a server), the caller will obviously block until the callee replies. Having been able to execute the syscall, the thread must be the highest-priority runnable thread, and the best way to observe its priority is to ensure that the callee completes as quickly as possible (and the callee is likely of higher priority anyway).

On the other hand, if a server replies (using `reply-and-wait`) to a waiting client, and the server has a request waiting from another client, it makes sense to continue the server to take advantage of the primed cache by executing the receive phase of its IPC.

Modern L4 versions, concerned about correct real-time behaviour, retain direct-process switch where it conforms with priorities, and else invoke the scheduler. In fact, direct-process switch is a form of time-slice donation, and Steinberg et al. [2005] showed that can be used to implement priority-inheritance and priority-ceiling protocols. Fiasco.OC and NOVA support this by allowing the user to specify donation on a per-call basis.

Replaced: Direct process switch subject to priorities in seL4 and optional in Fiasco.OC and NOVA.

4.4 Preemption

Traditionally L4 implementations had interrupts disabled while executing within the kernel, although some (like L4/MIPS) contained preemption points in long-running operations, where interrupts were briefly enabled. Such an approach significantly simplifies kernel implementation, as most of the kernel requires no concurrency control, and generally leads to better average-case performance.

However, the original L4 ABI had a number of long-running system calls, and early Fiasco work made the kernel fully preemptive in order to improve real-time performance [Hohmuth and Härtig, 2001]. Later ABI versions removed most of the long-running operations, and Fiasco.OC reverted to the original, mostly non-preemptible approach.

In the case of seL4, there is an additional reason for a non-preemptible kernel: avoiding concurrency to make formal verification tractable [Klein et al., 2009]. Given seL4's focus on safety-critical systems, many of which are of a hard real-time nature, we need hard bounds on the latency of interrupt delivery. It is therefore essential to avoid long-running kernel operations, and use preemption points where this is not possible (e.g. the practically unbounded object deletion). We put significant effort into placement of preemption points, as well as on data structures and algorithms that minimises the need for them [Blackham et al., 2012].

Note that a continuation-based event kernel provides natural support for preemption points (by making them continuation points).

Retained: Mostly non-preemptible design with strategic preemption points.

4.5 Non-portability

Liedtke [1995] makes the point that a microkernel implementation should not strive for portability, as a

hardware abstraction introduces overheads and hides hardware-specific optimisation opportunities. He cites subtle architectural changes between the "compatible" i486 and Pentium processors resulting in shifting trade-offs and implying significant changes in the optimal implementation.

This argument was debunked by Liedtke himself, with the high-performance yet portable Hazelnut kernel and especially Pistachio. Careful design and implementation made it possible to develop an implementation that was 80–90% architecture-agnostic.

In seL4, the architecture-agnostic code (between x86 and ARM) only accounts for about 50%. About half the code deals with virtual memory management which is necessarily architecture-specific. The lower fraction of portable code is a result of seL4's overall smaller size, with most (architecture-agnostic) resource-management code moved to userland. There is little architecture-specific optimisation except for the IPC fastpath. Steinberg [2013] similarly estimates a 50% rewrite for porting NOVA to ARM.

Replaced: Non-portable implementation by significant portion of architecture-agnostic code.

4.6 Non-standard calling convention

The original L4 kernel was completely implemented in assembler, and therefore the calling convention for functions was irrelevant inside the kernel. At the ABI, all registers which were not needed as syscall parameters were designated as message registers. The library interface provided inlined assembler stubs to convert the compiler's calling convention to the kernel ABI (in the hope the compiler would optimise away any conversion overhead).

The next generation of L4 kernels, starting with L4/MIPS, were all written at least partially in C. At the point of entering C code, these kernels had to re-establish the C compiler's calling convention, and revert to the kernel's convention on return. This made calling C functions relatively expensive, and therefore discouraged the use of C except for inherently expensive operations.

Later kernels where written almost exclusively in C (Hazelnut) or C++ (Fiasco, Pistachio). The cost of the calling-convention mismatch (and the lack of Liedtke-style masochism required for micro-optimising every bit of code) meant that the C code did not exhibit performance that was competitive to the old assembler kernel. The implementors of those kernels therefore started to introduce hand-crafted assembler *fast paths*. These led to IPC performance comparable to the original L4 (see Table 1).

The traditional approach was unsuitable for seL4, as the verification framework could only deal with C code [Klein et al., 2009], and we wanted to verify the kernel's functionality as completely as feasible. This requires restricting assembler code to the bare minimum, and rules out calling-convention conversions, forcing us to adopt the tool chain's standard calling conventions.

Abandoned: Non-standard calling conventions.

4.7 Implementation language

seL4 is also highly dependent on fast-path code to obtain competitive IPC performance, but the fast paths must now be implemented in C. The assembler fast path had already been abandoned in the commercial OKL4 kernel because of the high maintenance cost of assembler code, which in the commercial environment outweighed any performance degradation. For seL4 we were willing to tolerate no more than a 10% degradation in IPC performance, compared to the fastest kernels on the same architecture.

Fortunately, it turned out that by carefully hand-crafting the fast path, we can achieve highly-competitive IPC latencies [Blackham and Heiser, 2012]. Specifically this means manually re-ordering statements, making use of (verified) invariants that the compiler is unable to determine by static analysis,

In fact, the finally achieved latency of 188 cycles for a one-way IPC on an ARM11 processor is about 10% *better* than the fastest IPC we had measured on any other kernel on the same hardware! This is partially a result of the simplified seL4 ABI and IPC semantics, and the fact that the event-based kernel no longer requires saving and restoring the C calling convention on a stack switch. We also benefit from improved compilers, especially their support for annotating condition branches for the common case, which helps code locality.

In any case, this result demonstrates that assembler implementations are no longer justified by performance arguments. The Dresden team in fact found that they could achieve highly-competitive performance without any fastpathing.

Abandoned: Assembler code for performance.

The first L4 kernel written completely in a high-level language was Fiasco, which chose C++ rather than C (which had be used for parts of the MIPS kernel a few years earlier). Given the state of C++ compilers at the time, this may seem a courageous decision, but is at least partially explained by the fact that Fiasco was not initially designed with performance in mind. This changed later, and the recent Fiasco experience demonstrates that,

when used correctly, there is no performance penalty from C++ code.

The Karlsruhe team also chose C++ for Pistachio, mostly to support portability. Despite a high degree of enthusiasm about C++ in Dresden and Karlsruhe, we never saw any convincing advantages offered by C++ for microkernel implementation. Furthermore, OK Labs found that the availability of good C++ compilers was a real problem in the embedded space, and they converted their version of the microkernel back to straight C.

For seL4, the requirements of verifications forced the choice of C. While Dresden's VFiasco project attempted to verify the C++ kernel [Hohmuth and Tews, 2005], it never completed formalising the semantics of the C++ subset used by Fiasco. In contrast, by using C to implement seL4, we could build on an existing formalisation of C [Norrish, 1998], a key enabler for the verification.

Abandoned: C++ for seL4 and OKL4.

5 Conclusions

It is rare that a research operating system has both a significant developer community, significant commercial deployment, as well as a long period of evolution. L4 is such a system, with 20 years of evolution of the API, of design and implementation principles, and about a dozen from-scratch implementations. We see this as a great opportunity to reflect on the principles and know-how that has stood the test of time, and what has failed to survive increased insights, changed deployment scenarios and the evolution of CPU architectures.

Design choices and implementation tricks came and went (including some which were close to the original designer's heart). However, the most general principles behind L4, minimality (including running device drivers at user level) and a strong focus on performance, still remain relevant and foremost in the minds of developers. Specifically we find that the key microkernel performance metric, IPC latency, has remained essentially unchanged (in terms of clock cycles), as far as comparisons across vastly different ISAs and micro architectures have any validity, in stark contrast to the trend identified by Ousterhout [1990] just a few years before L4 was created. Furthermore, and maybe most surprisingly, the code size has essentially remained constant, a rather unusual development in software systems.

Formal verification increased the importance of minimality, and also increased pressure for simplification of the implementation. Several design decisions, such as the simplified message structure, user-level control of kernel memory and the approach to multicores are strongly influenced by verification. It also impacted a number of implementation approaches, such as the use of an event-oriented kernel, adoption of standard calling convention, and the choice of C as the implementation language. However, we do not think that this has led to tradeoffs which we would consider inferior when ignoring verification.

With formal verification, L4 has convincingly delivered on one of the core promises microkernels made many years ago: robustness. We think it is a great testament to the brilliance of Liedtke's original L4 design that this was achieved while, or maybe due to, staying true to the original L4 philosophy. It may have taken an awfully long time, but time has finally proved right the once radical ideas of Brinch Hansen [1970].

There is one concept that has, so far, resisted any satisfactory abstraction: *time*. L4 kernels still implement a specific scheduling policy – in most cases priority-based round-robin – the last major holdout of policy in the kernel. This probably represents the largest limitation of generality of L4 kernels. There is work underway at Dresden and NICTA that indicates that a single, parameterised kernel scheduler may actually be able to support all standard scheduling policies, and we expect it will not take another 20 years to get there.

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

L4 would not exist without its inventor, Jochen Liedtke, and we pay tribute to his brilliance. We are also greatly indebted to the many people who contributed to L4 over two decades, generations of staff and students at IBM Watson, TU Dresden, University of Karlsruhe, the University of New South Wales and NICTA; there are too many to name them all.

We are specifically grateful to members of the L4 community who provided feedback on drafts of this paper: Andrew Baumann, Ben Leslie, Chuck Gray, Hermann Härtig. We thank Adam Lackorzynski for digging out the original L4 sources and extracting SLOC-counts, and Adrian Danis for some last-minute seL4 optimisations and measurements. We thank the anonymous SOSP reviewers for their insightful comments, and our shepherd John Ousterhout for his feedback.

References

Michael T. Alexander. Organization and features of the Michigan terminal system. In *AFIPS Conference Pro-*

ceedings, 1972 Spring Joint Computer Conference, pages 585–591, 1972.

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, October 2009. ACM.

Bernard Blackham and Gernot Heiser. Correct, fast, maintainable – choose any three! In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*, pages 13:1–13:7, Seoul, Korea, July 2012. doi: 10.1145/2349896.2349909.

Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pages 339–348, Vienna, Austria, November 2011. doi: 10.1109/RTSS.2011.38.

Bernard Blackham, Yao Shi, and Gernot Heiser. Improving interrupt response time in a verifiable protected microkernel. In *Proceedings of the 7th EuroSys Conference*, pages 323–336, Bern, Switzerland, April 2012. doi: 10.1145/2168836.2168869.

Per Brinch Hansen. The nucleus of a multiprogramming operating system. *Communications of the ACM*, 13: 238–250, 1970.

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

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

Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–155, 1966.

Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. Kernel design for isolation and assurance of physical memory. In *1st Workshop on Isolation and Integration in Embedded Systems*, pages 35–40, Glasgow, UK, April 2008. ACM SIGOPS. doi: 10.1145/1435458.

Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS)*, 2004.

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

Andreas Haeberlen. Managing kernel memory resources from user level. Diploma thesis, Dept of Computer Science, University of Karlsruhe, April 2003. URL http://os.ibds.kit.edu/english/97_639.php.

Hermann Härtig and Michael Roitzsch. Ten years of research on L4-based real-time systems. In *Proceedings of the 8th Real-Time Linux Workshop*, Lanzhou, China, 2006.

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

Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the 1st Asia-Pacific Workshop on Systems (APSys)*, pages 19–24, New Delhi, India, August 2010.

Michael Hohmuth and Hermann Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, USA, 2001.

Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verified operating system. In *Proceedings of the 2nd Workshop on Programming Languages and Operating Systems (PLOS)*, Glasgow, UK, July 2005.

Trent Jaeger, Kevin Elphinstone, Jochen Liedtke, Vsevolod Panteleenko, and Yoonho Park. Flexible access control using IPC redirection. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, USA, March 1999.

J. Leslie Keedy. On the programming of device drivers for in-process systems. Monads Report 5, Dept. of Computer Science, Monash University, Clayton VIC, Australia, 1979.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM. doi: 10.1145/1629575.1629596.

Adam Lackorzynski and Alexander Warg. Taming subsystems: capabilities as universal resource access control in L4. In *2nd Workshop on Isolation and Integration in Embedded Systems*, pages 25–30, Nuremburg, Germany, March 2009.

Adam Lackorzynski, Alexander Warg, Marcus Völp, and Hermann Härtig. Flattening hierarchical scheduling. In *International Conference on Embedded Software*, pages 93–102, Tampere, Finland, October 2012.

Ben Leslie, Peter Chubb, Nicholas FitzRoy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting (Rita) Shen, Kevin Elphinstone, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, September 2005.

Roy Levin, Ellis S. Cohen, William M. Corwin, Fred J. Pollack, and William A. Wulf. Policy/mechanism separation in HYDRA. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles*, pages 132–140, 1975.

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

Jochen Liedtke. A persistent system in real use: Experience of the first 13 years. In *Proceedings of the 3rd IEEE International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 2–11, Asheville, NC, USA, December 1993b. IEEE.

Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

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

Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two years of experience with a μ-kernel based OS. *ACM Operating Systems Review*, 25(2):51–62, April 1991.

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

Jochen Liedtke, Nayeem Islam, and Trent Jaeger. Preventing denial-of-service attacks on a μ-kernel for WebOSes. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 73–79, Cape Cod, MA, USA, May 1997b. IEEE.

Steven. B. Lipner. A comment on the confinement problem. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles*, pages 192–196. ACM, 1975.

Richard J. Lipton and Lawrence Snyder. A linear time algorithm for deciding subject security. *Journal of the ACM*, 24(3):455–464, 1977. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/322017.322025.

Paul E. McKenney, Dipankar Sarma, Andrea Arcangelli, Andi Kleen, Orran Krieger, and Rusty Russell. Read copy update. In *Proceedings of the Ottawa Linux Symposium*, 2002. URL http://www.rdrop.com/users/paulmck/rclock/rcu.2002.07.08.pdf.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013. ISBN 10.1109/SP.2013.35.

Roger M. Needham and R.D.H. Walker. The Cambridge CAP computer and its protection system. In *Proceedings of the 6th ACM Symposium on Operating Systems Principles*, pages 1–10. ACM, November 1977.

Michael Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge Computer Laboratory, 1998.

Abi Nourai. A physically-addressed L4 kernel. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, March 2005. Available from publications page at http://ssrg.nicta.com.au/.

John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the 1990 Summer USENIX Technical Conference*, pages 247–56, June 1990.

Kaushik Kumar Ram, Jose Renato Santos, and Yoshio Turner. Redesigning Xen's memory sharing mechanism for safe and efficient I/O virtualization. In *Proceedings of the 2nd Workshop on I/O Virtualization*, Pittsburgh, PA, USA, 2010. USENIX.

Richard Rashid, Avadis Tevanian, Jr., Michael Young, David Golub, Robert Baron, David Black, William J. Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, C-37:896–908, 1988.

Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *2nd International Conference on Interactive Theorem Proving*, volume 6898 of *Lecture Notes in Computer Science*, pages 325–340, Nijmegen, The Netherlands, August 2011. Springer. doi: http://dx.doi.org/10.1007/978-3-642-22863-6_24.

Jonathan S. Shapiro. Vulnerabilities in synchronous IPC designs. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2003. URL citeseer.ist.psu.edu/shapiro03vulnerabilities.html.

Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Charleston, SC, USA, December 1999. URL http://www.eros-os.org/papers/sosp99-eros-preprint.ps.

Udo Steinberg. Personal communication, 2013.

Udo Steinberg and Bernhard Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th EuroSys Conference*, Paris, France, April 2010.

Udo Steinberg, Jean Wolter, and Hermann Härtig. Fast component interaction for real-time systems. In *Euromicro Conference on Real-Time Systems*, pages 89–97, Palma de Mallorca, Spain, July 2005.

Volkmar Uhlig. *Scalability of Microkernel-Based Systems*. PhD thesis, University of Karlsruhe, Karlsruhe, Germany, June 2005.

Michael von Tessin. The clustered multikernel: An approach to formal verification of multiprocessor OS kernels. In *Proceedings of the 2nd Workshop on Systems for Future Multi-core Architectures*, Bern, Switzerland, April 2012.

Matthew Warton. Single kernel stack L4. BE thesis, School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, November 2005.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. *Communications of the ACM*, 54(11): 93–101, November 2011.