评注见文尾

L4 Microkernels: The Lessons from 20 Years of Research and Deployment

GERNOT HEISER and KEVIN ELPHINSTONE, NICTA and UNSW, Sydney, Australia

The L4 microkernel has undergone 20 years of use and evolution. It has an active user and developer community, and there are commercial versions that are deployed on a large scale and in safety-critical systems. In this article we examine the lessons learnt in those 20 years about microkernel design and implementation. We revisit the L4 design articles and examine the evolution of design and implementation from the original L4 to the latest generation of L4 kernels. We specifically look at 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, generality, and high inter-process communication (IPC) performance remain the main drivers of design and implementation decisions.

Categories and Subject Descriptors: D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Design, Performance, Security, Verification

Additional Key Words and Phrases: Microkernels, L4, seL4, IPC, message passing, minimality, performance, formal verification, real time, worst-case execution time, virtualization

ACM Reference Format:
Gernot Heiser and Kevin Elphinstone. 2016. L4 microkernels: The lessons from 20 years of research and deployment. ACM Trans. Comput. Syst. 34, 1, Article 1 (April 2016), 29 pages.
DOI: http://dx.doi.org/10.1145/2893177

1. INTRODUCTION

Twenty years ago, Liedtke [1993a] demonstrated with his L4 kernel that microkernel IPC could be fast, a factor 10–20 faster than contemporary microkernels. This was the start of a journey that created a whole family of microkernels and led to large-scale commercial deployment as well as to systems of unprecedented assurance.

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 [Brinch Hansen 1970; Levin et al. 1975]. Application code obtains a system service by communicating with servers via an interprocess 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 1990s, IPC performance had become the achilles heel of microkernels: The typical cost for a one-way message was around 100 μs, which was too high for building performant systems. This resulted in a trend to move core services back into

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.
Authors' addresses: G. Heiser and K. Elphinstone, School of Computer Science & Engineering, UNSW Sydney NSW 2052, Australia; Correspondence email: {gernot, k.elphinstone}@unsw.edu.au.

DOI: http://dx.doi.org/10.1145/2893177

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 paravirtualised 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 comprehensive formal verification, including a functional correctness proof of the implementation and a complete proof chain of high-level security properties down to the executable binary [Klein et al. 2014]. L4 also had a strong influence on other research systems, such as Pebble [Gabber et al. 1999], EROS [Shapiro et al. 1999], and Barrelfish [Baumann et al. 2009].

In this article we examine the development of L4 since the mid-1990s. Specifically, we look at what makes modern L4 kernels tick, how this relates to Liedtke's original design and implementation rules, 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 evolved from an earlier system, called L3, developed at GMD by Liedtke [1993b] in the mid-1980s on i386 platforms. L3 was a complete OS with built-in persistence, and it already featured user-mode drivers [Liedtke et al. 1991], still a 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 on 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 20yr evolution, with multiple ABI revisions and from-scratch implementations, as depicted in Figure 1. It started with TU Dresden and UNSW reimplementing the ABI, with necessary adaptations, on 64-bit Alpha and MIPS processors, the latter implemented all longer-running operations in C. Both kernels achieved submicrosecond 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, *Hazelnut*, which was written in C and was the first L4 kernel that was ported (rather than reimplemented) to another architecture (from Pentium to ARM).

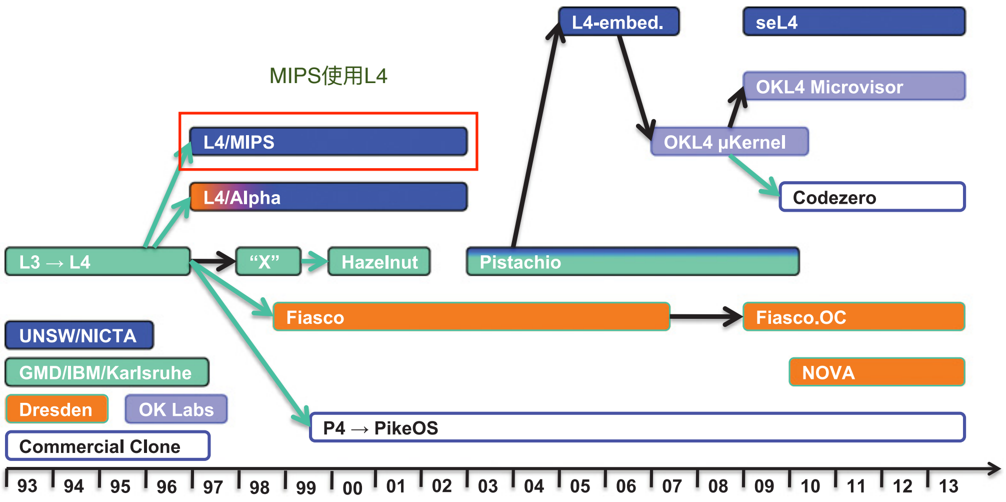

Fig. 1. The L4 family tree (simplified). Black arrows indicate code, green arrows ABI inheritance. Box colours indicate origin as per key at the bottom left.

Karlsruhe's experience with Version X and Hazelnut resulted in a major ABI revision, V4, aimed at improving kernel and application portability and multiprocessor 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 implemented in C++ on x86 and PowerPC; 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 then retargeted Pistachio for use in resource-constrained embedded systems, resulting in a fork called *NICTA::Pistachio-embedded* ("L4-embedded"). It saw massive-scale commercial deployment when Qualcomm adopted it as a protected-mode real-time OS for the firmware of their wireless modem processors. It is now running on the security processor of all recent Apple iOS devices [Apple Inc 2015]. NICTA spinout Open Kernel Labs (OK Labs) took on the support and further development of this kernel, renaming it the *OKL4 microkernel*.[2] Another deployed version is PikeOS, a commercial V2 clone by German company Sysgo, certified for use in safety-critical avionics and deployed in aircraft and trains.[3]

The influence of KeyKOS [Hardy 1985] and EROS [Shapiro et al. 1999] and an increased focus on security resulted in the adoption of capabilities [Dennis and Van Horn 1966] for access control, first with the 2.1 release of OKL4 (2008) and soon followed by Fiasco; Fiasco was renamed *Fiasco.OC* in reference to its use of object capabilities. 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.

Virtualized Linux became the de facto OS middleware, and most L4 applications required at least one Linux VM. This then led to new designs specifically aimed at supporting virtualisation as the primary concern, specifically *NOVA* [Steinberg and

[1]We also worked on Itanium [Gray et al. 2005] and SPARC versions, but they were never completed.
[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/.

Table I. One-Way Cross-Address-Space IPC Cost of Various L4 Kernels

Name	Year	Architecture	Processor	MHz	Cycles	μs
Original	1993	486	DX50	50	250	5.00
Original	1997	x86 (32-bit)	Pentium	160	121	0.75
L4/MIPS	1997	MIPS64	R4700	100	100[a]	1.00
L4/Alpha	1997	Alpha	21064	433	70–80[a]	0.17
Hazelnut	2002	x86 (32-bit)	Pentium II	400	273	0.68
Hazelnut	2002	x86 (32-bit)	Pentium 4	1,400	2,000	1.38
Pistachio	2005	IA-64	Itanium 2	1,500	36	0.02
OKL4	2007	ARM v5	XScale 255	400	151	0.64
NOVA	2010	x86 (32-bit)	Core i7 (Bloomfield)	2,660	288[b]	0.11
seL4	2013	x86 (32-bit)	Core i7 4770 (Haswell)	3,400	301	0.09
seL4	2013	ARMv6	ARM11	532	188	0.35
seL4	2013	ARMv7	Cortex A9	1,000	316	0.32

[a]These figures are somewhat higher than those of Liedtke et al. [1997a], who reported on incomplete kernel implementations.
[b]NOVA does not support the standard GCC version of thread local storage (using segment registers). Instead it supports TLS via a general-purpose register. This reduces context-switch costs at the expense of increased register pressure.

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 microarchitecture, as shown in Table I. Consequently, the L4 community tends to measure IPC latencies in cycles rather than microseconds, as this better relates to the hardware limits. In fact, the table 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 on contemporary architectures.

2.2. Modern Representatives

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

Security, in particular the ability to provide isolation and protect critical system assets from less trustworthy code, has always been a driver for L4's design. It is a prime motivation for running device drivers, network stacks, file systems, and so on, at the user level. In the past, however, the focus was mostly on providing such isolation with minimal performance impact while maintaining generality; actual assurance received little attention.

In contrast, seL4 was 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. The decision to support formal reasoning 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-mode OS kernel in the literature with a complete and sound worst-case execution time (WCET) analysis [Blackham et al. 2011, 2012].

Table II. Source Lines of Code (SLOC) of Various L4 Kernels

		Size (kSLOC)		
Name	Architecture	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 microkernel v3.0	ARMv6	15.0	0.0	15.0
Fiasco.OC	x86	36.2	1.1	37.6
seL4	ARMv6	9.7	0.5	10.2

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, over time, supported most 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 virtualisation on x86 platforms, and the OKL4 Microvisor [Heiser and Leslie 2010] ("OKL4" for short), which was designed as a commercial platform for efficient paravirtualisation on ARM processors.

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 and Generality

The main drivers of Liedtke's designs were minimality and IPC performance, with a strong conviction that the former helps the latter. Specifically, he formulated the microkernel *minimality principle* [Liedtke 1995]:

> 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.

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. It means that the kernel is just a thin, no-frills wrapper around hardware, a "CPU driver,"[4] which only controls the hardware on behalf of a higher instance that defines policy. The discussion in the following sections will demonstrate the community's ongoing 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 II: 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

[4]Former L4 developer Charles Gray is credited with coining this term [Baumann et al. 2009].

traditional model), is still essentially the same size as the early versions.[5] Verification provided a particular strong motivation for minimality, as even 9,000 SLOC pushed the limits of what was achievable.

An aim of L4 that was rarely stated explicitly, but was always a driver for the design of kernel mechanisms, is *generality*: L4 always strove to be a foundation on which (almost arbitrary) systems can be built, essentially anything that makes sense to run on a processor powerful enough to provide protection.

Some versions of L4 compromised somewhat on the generality aim, especially the OKL4 versions with their focus on resource-constrained embedded systems. But such compromises were always understood as stop-gap measures, forced by commercial realities while the search for a more general model continued.

Kept: Minimality as key design principle and generality as the overall aim.

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, no one 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. In fact, in its simplest version (short messages passed in registers) it is nothing but a context switch that leaves the message registers untouched. This is consistent with the idea of the thin CPU driver, and typical L4 implementations have IPC costs that are only 10% to 20% above the hardware limit (defined as the cost of two mode switches, a switch of page tables, plus saving and restoring addressing context and user-visible processor state).

Synchronous IPC 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 multithreaded 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.

Furthermore, synchronous message passing is clearly the wrong way of synchronising activities across processor cores. On a single processor, communication between threads requires that (eventually) a context switch happens, and combining the context switch with communication minimises overheads. Consequently, the classical L4 IPC model is that of a user-controlled context switch that bypasses the scheduler; some

[5]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.

payload is delivered through nonswitched registers, and further optional payload by kernel copy.

On hardware that supports true parallelism, an RPC-like server invocation sequentialises client and server, which should be avoided if they are running on separate cores [Baumann et al. 2009; Soares and Stumm 2010].

We addressed this in L4-embedded by adding *notifications*, a simple, nonblocking signalling mechanism. We later refined this model in seL4's *notification objects*: A notification contains a set of flags, the *notification word*, which is essentially an array of binary semaphores. A *signal* operation on a notification object sets a subset of the flags without blocking. The notification word can be checked by polling or by waiting (blocking) for a signal—effectively `select()` across the notification word.

Our present design provides another feature aimed at reducing the need for multithreaded code, unifying waiting for IPC and notifications. For example, a file server might have an IPC interface for client requests, as well as a notification used by the disk driver to signal I/O completion. The unification feature *binds* a notification object to a thread (the server of the above example). If a notification is signalled while the thread is waiting for a message, the notification is converted into a single-word message and delivered to the thread (with an indication that it is really a notification).

Notifications are not an introduction of asynchronous IPC through the backdoor but rather a (partial) decoupling of synchronisation from communication. While strictly not minimal (in that they add no functionality that could not be emulated with other mechanisms), they are essential for exploiting concurrency of the hardware.[6]

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

Kept: Synchronous IPC for efficient client-server interaction without scheduler invokation (except in OKL4).

New: Semaphore-like notifications for synchronisation of real concurrency.

3.2.2. IPC Message Structure. Original L4 IPC had rich semantics. Besides in-register ("short") messages, it supported messages of almost arbitrary size in addition to in-register arguments: a word-aligned "buffer" as well as multiple unaligned "strings." 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) 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 *virtual message registers* (originally 64 and later a configuration option). The implementation mapped some of them to physical registers, and the rest was contained in a per-thread pinned part of the address space. The pinning ensures register-like access 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.

[6]We had confused ourselves, and many others, on this issue, by initially talking about asynchronous IPC. We credit Anton Burtsev with prompting us to clarify our thinking through his question at SOSP'13.

The motivation is two-fold: virtual message registers greatly improve portability across architectures. Furthermore, 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 four-word message) compared to passing via the kernel stack; on Cortex A9 this reduces to 4%. On latest-generation x86 processors, reserving any registers for message passing is so detrimental to the compiler’s ability to optimise the code that it results in an overall loss of performance.

Replaced: Physical by virtual message registers.

In original L4, “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 reestablished. The result was significant kernel complexity, with many tricky corner cases that risked bugs in the implementation.

While long IPC provides functionality that 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. Furthermore, the introduction of virtual registers increased the supported size of “simple” messages to the point where the cost of the copy exceeds that of the basic system-call overhead. In cases where the trust relationship requires bulk copy by a trusted agent, the cost (extra system call) of making this a user-level process has become largely irrelevant.

The main use of long IPC was for POSIX-style read-write interfaces to servers, which require transferring the contents of arbitrary buffers to servers that do not necessarily have access to the client’s memory. However, the rise of virtualised 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. Furthermore, it 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 running on a platform with only a few MiB of RAM, where a separate communication page per pair of communicating threads was too costly. Arguably, this design is no longer justified.

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 multithreaded 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 colocation on the same page. Last but not least, the global IDs introduced covert channels [Shapiro 2003].

Influenced by EROS [Shapiro et al. 1999], seL4 and Fiasco.OC [Lackorzynski and Warg 2009]) adopted *IPC endpoints* as IPC destinations. seL4 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 [Accetta et al. 1986], IPC endpoints do not provide any buffering.

In order to help servers identify clients without requiring per-client endpoints, seL4 provides *badged capabilities*, similar to the *distinguished capabilities* of KeyKOS [Bromberger et al. 1992]. Capabilities with different badges but derived from the same original capability refer to the same (endpoint) object but on invokation deliver to the receiver the badge as an identification of the sender.

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 operations in the original L4 had timeouts. Specifically, an IPC syscall specified four 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 nontrivial system, and in practice, 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.[7] 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 nonexisting 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 redirected, each such redirection 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 a 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 on endpoints.

[7]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.

Abandoned: Clans and chiefs.

3.3. User-Level Device Drivers

A key consequence of the minimality principle, and maybe the most controversial feature of L4 (or, rather, its predecessor, L3 [Liedtke et al. 1991]), was to make all device drivers user-level processes.[8] 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 (changed back and back again), but the general approach still applies.

The most notable change was moving from IPC to notifications as the mechanism for interrupt delivery. The main driver here was implementation simplification, as delivery as messages required the emulation of virtual in-kernel threads as the sources of interrupt IPC, while signalling notifications is a natural match to what the hardware does.

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.

Kept: User-level drivers as a core feature.

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. 2005a]. 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 subsystems: System mechanisms restrict children's privileges to be a subset of their parent's. The cost is rigidity. Moreover, 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

[8]This had been done before, in Brinch Hansen's Nucleus [1970], the Michigan Terminal system [Alexander 1972], and the Monads OS [Keedy 1979], but there is no hard information about performance of those systems, and general expectation was that such designs performed poorly. MINIX also had user-level drivers [Tanenbaum 2016] but was not published until after Liedtke created L3 [Liedtke 1993b].

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. L4 does not have a first-class process concept, it is a higher-level abstraction that consists of an address space, represented by a page table, and a number of associated threads. These consume 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, a few thousand, 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, which imposes a particular policy, proved inflexible and restrictive; it was eventually replaced by full-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 page in its address space, a process had the right to map the 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 an identity 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. 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 articles, 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, see Section 3.2.5).

In L4-embedded we removed the recursive mapping model, after observing that for our real-world use cases, 25% to 50% of kernel memory 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 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 provides: 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 in Section 3.4.3.

Xen provides an interesting point of comparison [Fraser et al. 2004]. *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. A more 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 of fine-grained delegation and revocation represent different tradeoffs 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. Only seL4 provides fine-grained delegation of page access.

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, most L4 kernels did not support σ_1.

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 system where authority is represented by capabilities, it is not possible to reason about the security state if there are resources, such as kernel memory, 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 likely introduce side 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 foregoing 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 tradeoff is still being explored in the community.

We take 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. The only exception is the fixed amount of memory 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, 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 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 III gives the complete set of seL4 object types and their use for 32-bit ARM processors, x86 is very similar. 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 adheres to the authority distributed to applications for its own data structures, including the consumption of physical memory.
(4) Each kernel object can be reclaimed independently of any other kernel object.
(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 [Sewell et al. 2011; Murray

Table III. seL4 Kernel Objects for 32-bit ARM Processors

Object	Description
TCB	Thread control block
Cnode	Capability storage
Endpoint	Port-like rendezvous object for IPC
Notification	Array of flags resembling binary semaphores
Page Directory	Top-level page table for ARM virtual memory
Page Table	Leaf page table for ARM 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
IRQ Handler	The right to associate with or acknowledge a specific interrupt source.
IRQ Control	The right to manage the delegation of interrupts, i.e. create *IRQ Handler* capabilities.

et al. 2013]. Property 5 ensures that all kernel operations have low, bounded latencies and thus supports its use for hard real-time systems [Blackham et al. 2011]. There is one long-running operation, capability revocation, which requires preemption points. These ensure that the kernel is in a consistent state, that it has made progress, and to check for pending interrupts. If there are pending interrupts, then the kernel returns to usermode to restart the system call, which ensures that the interrupt gets handled first. The restarted system call continues the tear-down operation where the previous attempt was discontinued.

Property 4 ensures kernel integrity. Any holder of an appropriate capability can reclaim an object at any time, which makes 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 capability derivation tree helps satisfying this requirement. Objects are revoked by invoking the `revoke()` method on an *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 reuse. 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.

New: 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 subdivided 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 principle 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] but was only implemented in the legacy Fiasco kernel (the one predating the introduction of capabilities) as it does not work with the indirection provided by IPC endpoints.

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, as it strongly reinforces the desire to have a single platform for all usage scenarios, that is, generality. 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.

4. MICROKERNEL IMPLEMENTATION

Liedtke [1993a] lists a set of design decisions and implementation tricks that helped to make 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 that 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 at 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; the kernel handles this 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.[9]

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

[9]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 the kernel to detect stale thread IDs.

tagged TLBs, which changed the tradeoffs. Furthermore, like page faults in long IPC (see Section 3.2.2), the virtual TCB array required handling nested exceptions, that is, page faults triggered while in the kernel. It thus adds significant kernel complexity and massive challenges to formal verification.

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 microbenchmarks but a 20% performance advantage of the event kernel on a multitasking 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 for storing the continuations.

Concurrently, Nourai [2005] analysed the tradeoffs of virtual vs physical addressing of TCBs. He implemented physical addressing, also in Pistachio, although on a MIPS64 processor. He found few, 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, at least on MIPS processors, there is no performance benefit from the virtually addressed TCBs, and other modern processors are unlikely to show significantly different tradeoffs.

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

Together, these results motivated the move to 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 the high-frequency IPC 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: Instead of leaving blocked threads in the ready queue, we defer *entering* unblocked threads into the ready queue until preemption time. This changes the main scheduler invariant from "all runnable threads are in the ready queue" to "the set of runnable threads consists of the currently executing thread plus the content of the ready queue."

Benno scheduling retains the desirable property that 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. This constant-time operation is the only fix-up needed. In addition, 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 the 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 and small 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, then 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, then 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 to a waiting client using `reply-and-wait`, 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 it 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 预售预购

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 nonpreemptible approach.

In the case of seL4, there is an additional reason for a nonpreemptible 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 to use preemption points where this is not possible, specifically 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]. We note that a continuation-based event kernel, such as seL4, provides natural support for preemption points by making them continuation points.

Kept: Mostly nonpreemptible design with strategic preemption points.

4.5. Nonportability

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 tradeoffs 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% to 90% architecture agnostic. Importantly, this was achieved without an explicit hardware-abstraction layer (HAL). Given the minimalist nature of L4 mechanisms, and the fact that most are thin layers around hardware mechanisms, a HAL would not be substantially smaller than seL4 itself. In fact, L4 can be viewed as little more than an (imperfect) HAL.

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: Nonportable implementation by mostly architecture-agnostic code without an explicit hardware-abstraction layer.

4.6. Nonstandard 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 that 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 reestablish 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 I).

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 commercial OKL4 kernel had already been abandoned the assembler fast path 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 reordering 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.

Abandoned: Assembler code for performance.

The first L4 kernel written completely in a high-level language was Fiasco. The developers chose C++ rather than C, which had been 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. Meanwhile, the performance penalty from C++ code has decreased significantly.

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.

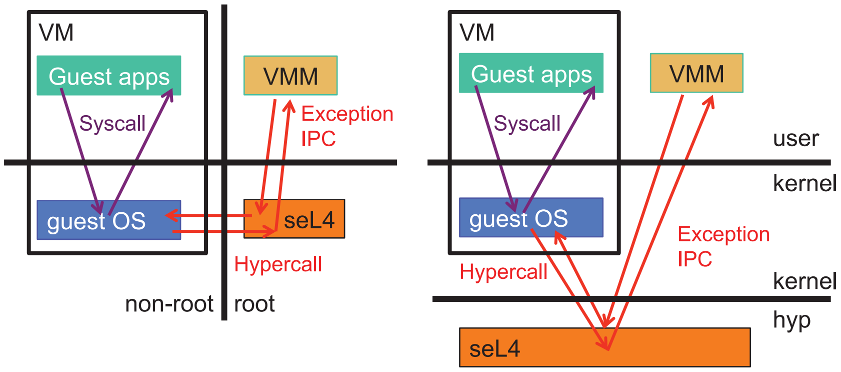

Fig. 2. Virtualisation architecture for x86 (left) and ARM (right): the microkernel forwards virtualisation exceptions to a user-level virtual-machine monitor.

For seL4, the requirements of verification 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. OTHER LESSONS

5.1. Virtualisation

The probably most significant use case that arose since Liedtke's original work is virtualisation. Early work on paravirtualised Linux [Härtig et al. 1997] led the way, although at this stage, virtualisation was simply a way to demonstrate achievable performance for full-functional systems. Linux soon became the OS middleware of choice for the Dresden group, used for supporting legacy software in real-time systems [Härtig et al. 1998].

At UNSW/NICTA, work on an architecture-independent paravirtualised Linux [Leslie et al. 2005b] led to adoption by Qualcomm and deployment on billions of mobile devices, including the use of virtualisation to share a single processor core between a Linux guest supporting apps and the real-time modem software [Heiser 2009].

Hardware support for virtualisation eliminated most of the engineering pain and made it easy to provide virtual machines with near-native performance. Contrary to most mainstream hypervisors, L4 kernels used as hypervisors minimise the trusted computing base, at least as far as isolation between virtual machines is concerned. Specifically, when serving as a hypervisor, the microkernel retains its role as a context-switching engine, doing little more than forwarding virtualisation exceptions to a user-mode virtual-machine monitor (VMM), just as it forwards interrupts to device drivers.

This architecture is indicated in Figure 2, which shows the various components and their privilege levels. The VMM is deprivileged, and there is one VMM instance per virtual machine [Steinberg and Kauer 2010]. As a consequence, while the VMM is part of a virtual machine's trusted computing base, it is not trusted by any other parts of the system; in particular, it cannot break isolation between virtual machines.

The seL4 VMM is approximately 20,000 SLOC. When combined with 9,000 SLOC for seL4 itself, these result are similar to results reported by Steinberg and Kauer [2010] for a deprivileged VMM and their virtualisation-specific kernel variant (NOVA). As they observed, the shared trusted computing base is orders of magnitude smaller in size than competing monolithic hypervisors.

Virtualisation has led to some kernel variants specifically designed as hypervisors and either compromising on generality (OKL4) or on minimality (NOVA). However, seL4 has very little functionality that is virtualisation specific, mostly context-switching extra virtual-machine state and delivering virtual-machine-related exceptions. The need to support virtualisation has required rethinking some design decisions, which has generally lead to cleaner and more general mechanisms.

Virtualisation was a driver for a cleaner, more general model.

5.2. Multicore

Multiprocessor issues were explored early 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 intercore communication cost and no shared caches. Consequently, the standard approach was to use per-processor scheduling queues and generally minimise sharing of kernel data across cores; threads migrate only upon explicit request from 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 servers 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 intercore communication cost and usually shared L2 caches, implying tradeoffs that differ from those on x86 as locality is less of an issue.

Verification also introduces new constraints. As discussed in Section 3.2.2, concurrency presents huge challenges for verification, and we kept it out of the seL4 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 a last-level cache [Peters et al. 2015]. Furthermore, sharing *any* kernel data makes little sense on a loosely coupled manycore without a shared on-chip cache. On such an architecture, the latency of migrating a few cache lines is in the thousands or tens of thousands of cycles [Baumann et al. 2009] and thus orders of magnitude larger than a typical L4 system call.

We are presently exploring a *clustered multikernel* (Figure 3): a hybrid of a big-lock kernel, across cores that share a cache, and a restricted variant of a multikernel where 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, the only such memory is a block of virtual IPC message registers.

Note that, while the clustering approach statically partitions memory between kernel images, which maps naturally to NUMA platforms, this does not prevent userland from viewing all memory as shared. In fact, each kernel instance presents itself as a virtual CPU to an unmodified SMP Linux guest that is able to efficiently support high-throughput workloads [Heiser et al. 2013].

The attraction of this approach is that it retains the existing uniprocessor proof with only small modifications. von Tessin [2013] lifted a parallel composition of the

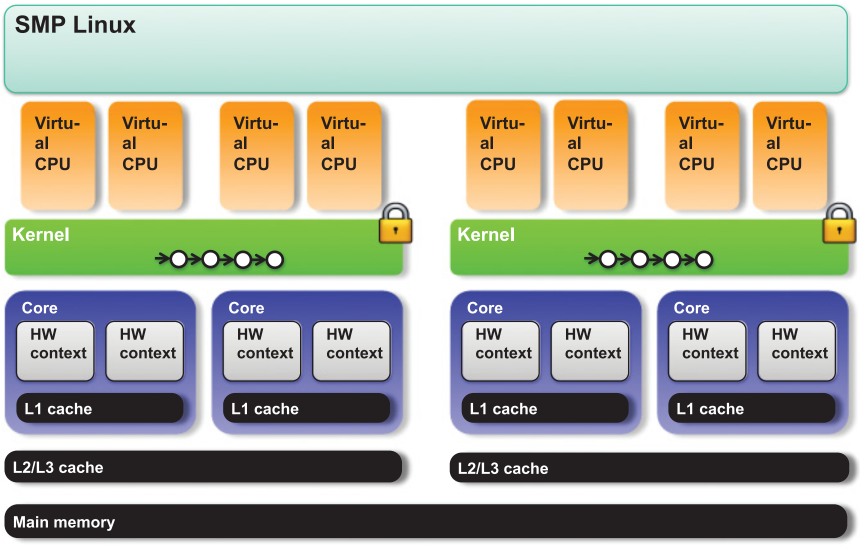

Fig. 3. Clustered multikernel: each closely coupled cluster shares a kernel image, protected by a big lock, while data is shared between kernel images running on different clusters. A Linux guest sees the whole cluster as a NUMA system.

uniprocessor automata and showed that refinement still holds. However, the formal guarantees no longer cover the entire system, only the individual kernel clusters. The large-step semantics used by the lifting framework assumes that kernel code is atomic between preemption opportunities. This does not allow extension of the formal framework to cover reasoning about the correctness of the lock, user-kernel concurrency, and any relaxation of resource migration restrictions. Such reasoning requires a small-step semantics, that is, modelling at the level of instructions.

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 claims 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.

5.3. The Influence of Architecture

Table I shows the enormous influence of (micro-)architecture on context-switching and thus IPC costs. In terms of cycles, these kept increasing on x86, culminating in the 2,000-cycle IPC on the Pentium 4 (NetBurst) architecture around the turn of the century. This created real challenges for microkernels: Our experience is that context switches that cost a few hundred cycles rarely ever matter, while thousands of cycles become a system performance issue, for example, with user-level drivers for high-bandwidth network interfaces.

Fortunately, with the increased focus on power consumption, the architectural trend reversed: the latest-generation x86 processors, beginning with the Haswell microarchitecture, enable context-switch times at par with RISC processors.[10] Interestingly, the trend for ARM processors is towards more expensive context switches. However,

[10]In fact, context-switch costs on Intel processors in 64-bit mode will be even lower, thanks to the tagged TLB and the removal of segment registers that are expensive to access. However, we do not yet have a sufficiently optimised x64 implementation to be more specific.

with tagged TLBs now the standard, agressive speculation apparently dead, and an increased number of architected registers no longer forcing long trap latencies due to the need to serialise much hidden state, things will hopefully never again get as bad as NetBurst.

One really significant architectural advance was the introduction of I/O MMUs as part of hardware-assist for virtualisation. This enabled low-overhead user-level drivers even for devices using DMA. Prior to that, user-level drivers for such devices had to be trusted, negating most of the benefit of removing them from the kernel, or DMA had to be controlled by expensive (in terms of engineering effort, as well as run-time cost) paravirtualisation of drivers. Here microkernels benefit from the fact that they face the same problem as hypervisors, namely the need to deprivilege device drivers.

Architecture matters.

5.4. The Influence of Verification

We have throughout commented on the influence verification had on seL4's design and implementation. Specifically, verification prohibited or discouraged

—a weak and unprincipled resource management model (Section 3.4.3)
—configurable variants, such as pluggable schedulers (Section 3.4.4)
—nested exceptions (Section 4.1)
—nonstandard calling conventions (Section 4.6)
—assembly code (Section 4.7)
—concurrency and locking (Section 5.2).

While dealing with these restrictions at times required more (mental) effort, in hindsight these challenges consistently led to more principled, cleaner designs. *The resulting solutions were inevitably closer to minimal and more general*. In other words, verification strongly reinforces the core principles of minimality and generality, which really constitute the heart of the L4 philosophy.

On top of that, verification imposed some coding discipline on the implementors. Almost all of this is in line with good software-engineering practice and thus is not really restrictive. The main exception is the prohibition on passing references to automatic variables to functions [Klein et al. 2009]. This is not an inherent requirement of verification, but a tradeoff between verification and implementation effort. It is the sole drawback of verification we observed to date and could be removed by more investment into the verification infrastructure.

The biggest concerns we had about verification, that it might force us into compromising performance or stop us from evolving the kernel, both turned out to be unfounded: seL4 is the best-performing L4 kernel where direct comparisons are available (see Section 4.7) and we found that the cost of keeping the proofs of an evolving kernel up to date scaled with the cost of evolving the implementation [Klein et al. 2014].

Verification forces clean and principled design and implementation, encourages minimality and generality, and has no significant drawbacks.

6. 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 underlying principles and examine what design and implementation

Table IV. Summary of What Did and Did Not Last the Distance
In the **Fate** column: A=Abandoned, K=Kept, R=Replaced, N=New; the **Agree** column indicates whether there is agreement in the community.

Feature	Fate	Comment	Agree?
Design Principles			
Minimality	K	remains core driver of design	yes
Generality	K	remains overall aim	yes
Abstractions/Mechanisms			
Synchronous IPC	K	mechanism for avoiding scheduling and redundant copying during client-server interaction	OKL4 differs
Notifications	N	semaphore-like communication mechanism for real concurrency	yes
Register messages	R	virtual registers replace physical for ABI stability and portability	yes
Long IPC	A	not minimal, reduced need due to longer (virtual) register messages, implementation complexity	yes
IPC targets	R	endpoint objects instead of global thread IDs for security, information hiding, supported by capability-based access control	yes
Clans & chiefs	A	inflexible and obsoleted by capability-based access control	yes
User-level drivers	K	core feature of L4 microkernels	yes
Hierarchical process management	A	inflexible and obsoleted by capability-based access control	yes
Recursive address spaces	R	mappings originate from frames with page-level delegation	Dresden retains
Kernel memory management	R	complete user-level control in seL4 and quota elsewhere	no
IPC timeouts	A	unnecessary complexity and practically unusable	Dresden differs
Time management	U	no satisfactory model yet	yes
Implementation Strategies			
Process kernel	R	moved to event-based kernel in all new implementations	yes
Virtual TCB addressing	R	no performance benefit on contemporary hardware, complexity of nested exceptions	yes
Lazy scheduling	R	Benno scheduling or strict scheduling	yes
Direct process switch	R	need to observe priorities	yes
Nonpreemptible	K	preemption points support low interrupt latency	yes
Nonportable	A	mostly architecture-agnostic implementation without HAL	yes
Nonstandard calling conventions	A	not worth it	yes
Assembler code	A	no longer needed for performance, incompatible with verification	yes
C++	A	dubious benefits, incompatible with verification	no

approaches have stood the test of time, and what has failed to survive increased insights, changed deployment scenarios and the evolution of CPU architectures.

As summarised in Table IV, design choices and implementation tricks came and went, including some that were close to the original designer's heart. However, the most general principles behind L4, minimality, including running device drivers at user level, generality, 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. This is 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 as well as generality, 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 conventions, and the choice of C as the implementation language. However, we do not think that this has led to tradeoffs we would consider inferior when ignoring verification; it certainly has not led to compromising the original goals of generality, performance, and minimality.

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 are a few issues that remain to be resolved. For example, the right approach to multicore, in the context of formal verification, is an open question, although there is recent progress.

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 contemporary L4 kernels. There is work in progress that indicates that a single, parameterised kernel scheduler may actually be able to support all standard scheduling policies [Lyons and Heiser 2014], and we expect it will not take another 20 years to get there.

ACKNOWLEDGMENTS

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, UNSW, 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 article: Andrew Baumann, Ben Leslie, Chuck Gray, and Hermann Härtig. We thank Adam Lackorzynski for digging out the original L4 sources and extracting SLOC-counts and Adrian Danis for many seL4 optimisations and measurements.

REFERENCES

Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. 1986. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Technical Conference*. US, 93–112.

Michael T. Alexander. 1972. Organization and features of the Michigan terminal system. In *AFIPS Conference Proceedings, 1972 Spring Joint Computer Conference*. 585–591.

Apple Inc. 2015. iOS security–iOS 9.0 or later. https://www.apple.com/business/docs/iOS Security Guide.pdf, September 2015.

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

Bernard Blackham and Gernot Heiser. 2012. Correct, fast, maintainable choose any three! In *Proceedings of the Asia-Pacific Workshop on Systems (APSys)*. 7.

Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. 2011. Timing analysis of a protected operating system kernel. In *Proceedings of the IEEE Real-Time Systems Symposium*. 339–348.

Bernard Blackham, Yao Shi, and Gernot Heiser. 2012. Improving interrupt response time in a verifiable protected microkernel. In *Proceedings of the EuroSys Conference*. 323–336.

Per Brinch Hansen. 1970. The nucleus of a multiprogramming operating system. *Commun. ACM* 13 (1970), 238–250.

Alan C. Bromberger, A. Peri Frantz, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. 1992. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Microkernels and Other Kernel Architectures*. 95–112.

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

Michael Condict, Don Bolinger, Dave Mitchell, and Eamonn McManus. 1994. *Microkernel Modularity with Integrated Kernel Performance*. Technical report. OSF Research Institute.

Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9 (1966), 143–155.

Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. 2008. Kernel design for isolation and assurance of physical memory. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*.

Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. 2004. 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)*.

Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. 1999. The Pebble component-based operating system. In *Proceedings of the 1999 USENIX Annual Technical Conference*. 267–282.

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

Andreas Haeberlen. 2003. *Managing Kernel Memory Resources from User Level*. Diploma Thesis. Dept of Computer Science, University of Karlsruhe. http://os.ibds.kit.edu/english/97_639.php.

Norman Hardy. 1985. KeyKOS architecture. *ACM Operating Systems Review* 19, 4 (Oct. 1985), 8–25.

Hermann Härtig, Robert Baumgartl, Martin Borriss, Claude-Joachim Hamann, Michael Hohmuth, Frank Mehnert, Lars Reuther, Sebastian Schönberg, and Jean Wolter. 1998. DROPS—OS support for distributed multimedia applications. In *Proceedings of the 8th SIGOPS European Workshop*.

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

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

Gernot Heiser. 2009. *The Motorola Evoke QA4: A Case Study in Mobile Virtualization*. White paper. Open Kernel Labs. Retrieved from https://www.researchgate.net/profile/Gernot_Heiser/publication/242743911_The_Motorola_Evoke_QA4_A_Case_Study_in_Mobile_Virtualization/links/00b7d53acc2c9d970d000000.pdf.

Gernot Heiser, Etienne Le Sueur, Adrian Danis, Aleksander Budzynowski, Tudor-Ioan Salomie, and Gustavo Alonso. 2013. RapiLog: Reducing system complexity through verification. In *Proceedings of the EuroSys Conference*. 323–336.

Gernot Heiser and Ben Leslie. 2010. The OKL4 microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the Asia-Pacific Workshop on Systems (APSys)*. 19–24.

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

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

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

J. Leslie Keedy. 1979. *On the Programming of Device Drivers for In-Process Systems*. Monads Report 5. Dept. of Computer Science, Monash University, Clayton VIC, AU.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.* 32, 1 (Feb. 2014), 2:1–2:70.

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. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM Symposium on Operating Systems Principles*. 207–220.

Adam Lackorzynski and Alexander Warg. 2009. Taming subsystems: Capabilities as universal resource access control in L4. In *Proceedings of the 2nd Workshop on Isolation and Integration in Embedded Systems*. 25–30.

Adam Lackorzynski, Alexander Warg, Marcus Völp, and Hermann Härtig. 2012. Flattening hierarchical scheduling. In *Proceedings of the International Conference on Embedded Software*. 93–102.

Ben Leslie, Peter Chubb, Nicholas FitzRoy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting (Rita) Shen, Kevin Elphinstone, and Gernot Heiser. 2005a. User-level device drivers: Achieved performance. *J. Comput. Sci. Technol.* 20, 5 (Sept. 2005), 654–664.

Ben Leslie, Carl van Schaik, and Gernot Heiser. 2005b. Wombat: A portable user-mode Linux for embedded systems. In *6th Linux.conf.au*. Canberra.

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

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

Jochen Liedtke. 1993b. 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)*. IEEE, 2–11.

Jochen Liedtke. 1995. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. 237–250.

Jochen Liedtke. 1996. Towards real microkernels. *Commun. ACM* 39, 9 (Sept. 1996), 70–77.

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

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

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

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

Richard J. Lipton and Lawrence Snyder. 1977. A linear time algorithm for deciding subject security. *J. ACM* 24, 3 (1977), 455–464.

Anna Lyons and Gernot Heiser. 2014. Mixed-criticality support in a high-assurance, general-purpose microkernel. In *Proceedings of the Workshop on Mixed Criticality Systems*. 9–14.

Paul E. McKenney, Dipankar Sarma, Andrea Arcangelli, Andi Kleen, Orran Krieger, and Rusty Russell. 2002. Read copy update. In *Proceedings of the Ottawa Linux Symposium*.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. 2013. seL4: From general purpose to a proof of information flow enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy*. 415–429.

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

Michael Norrish. 1998. *C Formalised in HOL*. Ph.D. Dissertation. University of Cambridge Computer Laboratory.

Abi Nourai. 2005. *A Physically-Addressed L4 Kernel*. BE Thesis. School of Computer Science and Engineering, Sydney, Australia. Available from publications page at http://ssrg.nicta.com.au/.

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

Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. 2015. For a microkernel, a big lock is fine. In *Proceedings of the Asia-Pacific Workshop on Systems (APSys)*.

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

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

Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. 2011. seL4 enforces integrity. In *Proceedings of the International Conference on Interactive Theorem Proving*. 325–340.

Jonathan S. Shapiro. 2003. Vulnerabilities in synchronous IPC designs. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*. US, 170–185.

Livio Soares and Michael Stumm. 2010. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

Udo Steinberg. 2013. Personal communication.

Udo Steinberg and Bernhard Kauer. 2010. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th EuroSys Conference*. 209–222.

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

Andrew S. Tanenbaum. 2016. Lessons learned from 30 years of MINIX. *Commun. ACM* 59, 3 (2016), 70–78.

Volkmar Uhlig. 2005. *Scalability of Microkernel-Based Systems*. Ph.D. Dissertation. University of Karlsruhe, Karlsruhe, Germany.

Michael von Tessin. 2012. The clustered multikernel: An approach to formal verification of multiprocessor OS kernels. In *2nd Workshop on Systems for Future Multi-core Architectures*. Bern, Switzerland, 1–6.

Michael von Tessin. 2013. *The Clustered Multikernel: An Approach to Formal Verification of Multiprocessor Operating-System Kernels*. Ph.D. Thesis. School of Computer Science and Engineering, UNSW, Sydney, Australia.

Matthew Warton. 2005. *Single Kernel Stack L4*. BE Thesis. School of Computer Science and Engineering, Sydney, Australia.

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

Received September 2015; accepted November 2015

评注：
因为芯片核心的架构设计与内核是互相借鉴相互补全最后达成最优分工的关系，本文通过对芯片各方面能力的评估，结合开发内核的大量权威经验，指出了如下几点："多核架构下的微核心通讯，需要提供非阻塞的同步支持。芯片架构对虚拟化的支持非常重要，这是架构健壮性与成熟性的表现之一，RISC指令集芯片在没有结构竞争与资源竞争时对于核心间状态切换过程有重大的影响，并且，CPU提供的存储访问架构极大地影响了其实际运行的效率，这一点上采取何种办法处理目前还有争议"。
最后，本文总结了经历时间洗礼最终达有成效或结论的芯片与系统技术，非常有指导意义。