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

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

Abstract

The L4 microkernel has undergone 20 years of use and evolution. It has an active user and developer community, and there are commercial versions which are deployed on a large scale. In this paper we examine the lessons learnt in those 20 years in relation to microkernel design. We revisit the L4 design papers, re-examine the approaches and conclusions, and provide insights into the long-term validity, or lack thereof, of the original design and implementation principles. We further examine the design of the seL4 kernel, which has pushed the L4 model furthest and was the first OS kernel to undergo a complete formal verification of its implementation. We use the seL4 design to illustrate how the microkernel has evolved, and to demonstrate the current state of evolution continues to possess the traits L4 is known for — performance and minimality.

1. Introduction

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

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

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

In this context, the order-of-magnitude improvement of IPC costs Liedtke demonstrated was quite remarkable. It was followed by work discussing the philosophy and mechanisms of L4 [Liedtke 1995; 1996a], the demonstration of a paravirtualized Linux on L4 with only a few percent overhead [Härtig et al. 1997], the deployment of an L4 version on billions of mobile devices,[1] and, finally, the world's first functional correctness proof of an OS kernel [Klein et al. 2009].

In this paper we examine the development of L4 over the last 20 years, Specifically we look at makes modern L4 kernels tick, and how this relates to Liedtke's original design and implementation, and which of his microkernel "essentials" have passed the test of time. We specifically examine how the lessons of the past have influenced at the design of seL4.

2. The L4 Microkernel Family

L4 developed out of an earlier system, called L3, developed by Liedtke [1993b] in the early 1980s on i386 platforms. L3 was a complete OS with built-in persistence, and it already featured user-mode drivers, still a characteristic of L4 microkernels. It was commercially deployed in a few thousand installations (mainly schools and legal practices). Initially, like all microkernels at the time, L3 suffered from IPC costs of the order of 100 μs.

Liedtke initially used L3 to try out new ideas, and what he referred to as "L3" in early publications [Liedtke 1993a] was actually an interim version of a radical re-design. The name "L4" was first used 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.

[1] 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.

Name	Year	Processor	MHz	Cycles
Original	1993	486	50	250
Original	1997	Pentium	160	121
L4/MIPS	1997	R4700	100	86
L4lpha	1997	21064	433	45
Pistachio	2005	Itanium 1	1,500	36
OKL4	2007	XScale 255	400	151
seL4	2013	ARM11	532	185

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

In the following years, Schönberg at TU Dresden started to develop an L4 implementation on the 64-bit Alpha processor (later completed at UNSW), while concurrently Elphinstone at UNSW did the same for MIPS64 processors. Both kernels were written from scratch, the Alpha kernel in Alpha PAL code, while the MIPS kernel had a core implemented in assembler but implemented longer-running operations, such as address-space manipulations, in C. Both achieved sub-microsecond IPC performance [Liedtke et al. 1997a] (see Table 1). They were the first open-source L4 kernels (both under GPL), and UNSW's release of the Alpha kernel was the first multiprocessor version of L4.

Liedtke kept experimenting with the assembler kernel, trying schemes that allowed performing intra-address-space IPC entirely at user level in a few dozen cycles [Liedtke and Wenske 2001], aimed at fast and transparent forwarding of requests from server gateway threads to worker threads. He referred to the experimental ABI as "Version X".

Around 1997, Hohmuth at Dresden started the implementation of a new L4 kernel for x86 processors, called Fiasco. It was the first L4 kernel written (almost) completely in a high-level language (C++), mostly for software-engineering reasons; it was also GPLed. Neither portability nor performance was an original design goal (although later versions were were tuned for high performance and ported to ARM).

Fiasco has over the years adapted to a series of changes of the L4 ABI, including a transition to capability-based access control (in what's now known as Fiasco.OC) and is actively maintained. It was early-on targeted for real-time use and was, unlike other L4 kernels, fully preemptible [Hohmuth and Härtig 2001], although later versions reverted to the mostly un-preemptible design that is characteristic of L4 kernels.

Soon after, Liedtke's students in Karlsruhe developed, again from scratch, the first L4 kernel designed to be portable, called *Hazelnut*, released in 1999 under GPL. It was implemented completely in C, originally on x86 and soon ported to ARM. Hazelnut eventually achieved performance competitive with Liedtke's all-assembler kernel by re-implementing performance-critical code paths in assembler, so-called "fast paths", a technique which was adopted by all other high-performance L4 kernels.

Until this time, all L4 kernels (other than Version X) implemented essentially the same (V2) ABI, with slight variants resulting from underlying ISA differences. Based on the Version X and Hazelnut experience, Liedtke and his students worked on a new ABI, called V4. It aimed at providing better hardware abstraction, but also attempted to overcome performance limitations which, as experience had show, resulted from some of the original mechanisms. More on this in Section 3.1.

In 2002, after Liedtke's death, Karlsruhe students Uhlig, Dannowski, Skoglund and LeVasseur built an implementation of V4 from scratch, and called it *L4Ka::Pistachio* ("Pistachio" for short). Released in 2003 under a BSD license, Pistachio was implemented in C++ and, from the beginning, designed to be portable as well as high-performance. The initial implementation was done concurrently on x86 and PowerPC processors, with a port to Itanium commenced soon afterwords (although the Itanium version was never completed). NICTA soon after ported Pistachio to Alpha, MIPS and ARM, and optimised the Itanium IPC path [Gray et al. 2005].

NICTA re-targeted the V4 ABI to resource-constrained embedded systems, calling the resulting ABI "N1" and the implementation NICTA::Pistachio-embedded (*L4-embedded* for short). Qualcomm adopted L4-embedded as a memory-protected real-time OS for their mobile chipset firmware and operating system, and L4-based mobile phones started shipping in 2006.

This lead to the creation of the company Open Kernel Labs (OK Labs), which marketed and further developed the kernel under the name OKL4. The 2.1 release of OKL4 (2008) was the first L4 kernel to use capability-based access control. OKL4 was distributed under an open-source license until version 3.0 (released in 2009), after which it went closed-source. OK Labs, having a strong interest in virtualisation, then built a hypervisor from scratch based on L4 principles, called the OKL4 Microvisor [Heiser and Leslie 2010]. At least 2 billion copies of versions of OKL4 were deployed on various mobile devices to date (and continue to ship).

A number of other commercial clones appeared over the years. Sysgo developed an independent implementation of the V2 ABI, called P4, which was later optimised, certified and deployed in avionics under the name PikeOS.[2] Codezero from B Labs is a recent GPLed clone of OKL4 V3.0 for ARMv7 processors in mobile devices.[3]

seL4 represents the most significant departure from the traditional L4 model. From the beginning, its design not only aimed to support formal verification, but also ease of reasoning about security and safety. This lead to a model where all (spatial) resource allocation is explicit and at user-level, including kernel memory [Elkaduwe et al. 2008]. It is also the first protected OS kernel in the literature with a complete and

[2] See Sysgo's web site http://www.sysgo.com/.

[3] See B Labs' web site http://dev.b-labs.com/.

Name	Architecture	Size (kLOC)		
		C/C++	asm	Total
Original	486	0.0	7	7
L4/Alpha	Alpha 21264	0.0	10.5	10.5
L4/MIPS	MIPS64/R4k	6.0	4.5	10.5
Hazelnut	x86	10.0	0.8	10.8
Pistachio	x86	22.4	1.4	23.0
L4-embedded	ARMv5	7.6	1.4	9.0
OKL4 3.0	ARMv6	15.0	0.0	15.0
Fiasco	x86	36.2	1.1	37.6
seL4	ARMv6	9.7	0.5	10.2

Table 2. Source lines of code (SLOC) of various L4 kernels. The size of the original kernel is an estimate derived from the 12 KiB binary size [Liedtke 1996b].

sound worst-case execution time (WCET) analysis [Blackham et al. 2011]. We will discuss seL4's design principles in Section 4.

3. Microkernel Design and Implementation

3.1 Principles and concepts

Liedtke [1995] outlines principles and mechanisms which drove the design of the original L4. We'll examine what is left of them in modern versions, especially seL4.

3.1.1 Minimality

Liedtke formulated a *minimality principle* as the core driver of his design:

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

This principle, which is a more pointed formulation of "only minimal mechanisms and no policy in the kernel," has continued to be a core driver of the design of L4 microkernels. This is possibly best demonstrated by a comparison of source code sizes, as shown in Table 2: seL4, the latest member of the family (and, arguably, the one that diverged strongest from the traditional model) is still essentially the same size as the early versions.[4]

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 workable mechanism which would delegate all scheduling policy to user-level without imposing high overhead.

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

Retained: Minimality is still the key design principle.

3.1.2 Recursive address spaces

A safe abstraction of memory is a fundamental requirement for a microkernel. The recursive address-space model of original L4 had an unusual approach: A new address space is initially empty. This (target) address space can be populated by another (source) address space (typically the target's "pager") mapping some of its own pages into the target space. The primitives are `map`, which logically creates a mapping from the target page to the source page, `grant`, which *moves* the source's page to the target (i.e. it creates a mapping from the target to the source's source, and the source address space completely loses the page) and `unmap`, which undoes a `map`. The recursion bottoms out in an address space "σ_0", which is magically created at boot time as a map of physical memory (as far as it is not reserved by the kernel).

The recursive address-space model is elegant, and naturally supports virtualisation at each level: a "virtualiser" can be inserted between the source and the destination, which forwards some mappings unchanged, but emulates others as needed. However, it has significant cost in terms of kernel complexity and memory overhead. If a page in an address space near the root of the hierarchy is revoked, all depending mappings must be unmapped as well.

Supporting this feature requires substantial bookkeeping in the form of a "mapping database". NICTA removed the recursive mapping model from the N1 ABI, after observing that typically 25–50% of kernel memory use was consumed by the mapping database. In L4-embedded, mappings always originate from frames, bringing the abstraction closer to hardware, an approach also adopted by the commercial OKL4 kernels. The cost of this change was foregoing the ability to delegate memory management. Full generality was restored later with seL4's memory-management model (see Section 4.2).

Abandoned: Recursive address spaces.

3.1.3 User-level device drivers and interrupts as IPC

The maybe most radical novelty of L4 (or, rather, its predecessor, L3 [Liedtke et al. 1991]) was to make all device drivers user-level processes.[5] This is still a hallmark of all L4 kernels. Like most of them, seL4 has a single driver in the kernel: a timer driver used for preempting user processes at the end of their time slice. All other device drivers run in user mode.

The user-level driver model is tightly coupled with modelling interrupts as IPC messages the kernel sends to the driver. Details of the model, as well as the association and acknowledgment protocol, have changed over the years (and

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

at times changed back and back again) but the principle still holds.

Retained: User-level drivers and interrupts as messages remain core philosophy.

3.1.4 Threads as IPC destinations

The original L4 had threads as the targets of IPC operations, although Liedtke [1993a] noted that ports could be implemented with a 12% overhead. The model required that thread IDs were unique identifiers.

The drawback of this model is poor information hiding. A multi-threaded server had to expose its internal structure to clients, or use a gateway thread, which could become a bottleneck and would impose additional communication and synchronization overhead. There were a number of proposals to mitigate this but they all had drawbacks. Furthermore, the global IDs introduced covert channels [Shapiro 2003].

Influenced by EROS [Shapiro et al. 1999], seL4 and other recent L4 kernels (such as Fiasco.OC [Lackorzynski and Warg 2009]) introduced *IPC endpoints* as IPC destinations. These are essentially lightweight ports: the root of the queue of pending messages/senders is a now a separate kernel object, instead of hanging off the recipient's thread control block (TCB).

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

3.1.5 Synchronous IPC and long messages

The original L4 only supported synchronous (rendezvous-style) IPC. It also supported messages of almost arbitrary size (a word-aligned "buffer" as well as multiple unaligned "strings") in addition to in-register arguments. This approach turned out to be problematic in a number of ways.

The all-synchronous design was driven by the desire to avoid redundant copying. Register arguments supported zero-copy: the kernel always initiated the IPC from the sender's context and switched to the receiver's context without touching the message registers.

"Long" messages could be delivered with a single copy: executing in the sender's context, the kernel sets up a temporary mapping 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, while still handling the IPC system call, pretending that the fault happened during normal user-mode execution, and on return re-establish the original system-call context. The result was significant kernel complexity, with many tricky corner cases that risked bugs in the implementation.

While long IPC provides functionality which cannot be emulated without some overhead, in practice it was rarely used. It also seemed to violate the minimality principle (which talks about functionality, not performance). Consequently, long IPC was removed from L4-embedded and the OKL4 microkernel derived from it.

For seL4 there were 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. 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).

Abandoned: "Long" IPC.

Besides this simplification, IPC was somewhat enriched in another way: by adding *asynchronous notifications*, first introduced in L4-embedded. These are a very simple form of asynchronous IPC: sending is asynchronous, in that the system call returns immediately after delivering the payload in the receiver's address space. The receiver can either block on a notification or poll. The feature was motivated by the realisation that the rendezvous model forces a multi-threaded design on many otherwise relatively simple systems.

Asynchronous notifications avoid introducing copy overhead by keeping the payload restricted to (a subset of) a single word. Each thread has a single-word notification field. A send operation specifies a mask of bits, which is OR-ed into the receiver's notification field. The receiver can wait on the notification bits to change or can poll the notification word.

Asynchronous notification thus is quite similar to hardware interrupts (with the slight generalisation that it can transfer multiple flags, while an interrupt generally just indicates an interrupt level or number). Consequently, current L4 versions model IRQs as asynchronous notifications.

Added: Asynchronous notifications.

3.1.6 Hierarchical task management and communication control

The original L4 had a process hierarchy, where there was a (large but fixed) set of *task IDs*, which were essentially process-management capabilities; they could be transferred to child address spaces. They doubled as a mechanism for communication control: IPC could only be sent to siblings (the "clan") or the parent (called "chief"), any IPC directed outside the clan was transparently re-directed to the chief (which could forward it in a way transparent to the receiver). Similarly, a message directed into a sibling's clan was re-directed to the sibling.

While Liedtke [1995] argued that the clans-and-chiefs model only added two cycles per IPC operation, this only applies where direct communication is possible. Once messages get re-directed, each such re-direction adds two messages to a (logically) single round-trip IPC, a significant overhead. Furthermore, the strict process hierarchy was unwieldy in prac-

tice (and was probably the L4 feature most cursed by people trying to build L4-based systems). It is a prime example of kernel-enforced policy (address-space hierarchy) limiting the design space.

As a consequence of these drawbacks, many L4 implementations did not implement clans-and-chiefs (or disable it 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 introduction of capability-based access control for all kernel objects (together with a clean model for controlling their propagation) in more recent L4 kernels provides a more principled (and in the end more efficient) model: Endpoint capabilities support communication control, and address-space capabilities support managing the address-space resource.

Abandoned: Clans-and-chiefs.

3.2 Design and implementation tricks

Liedtke [1993a] list a set of design decisions and implementation tricks which helped to make IPC fast in the original i386 version, although some 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 ones in more detail.

3.2.1 Strict process orientation and virtual TCB array

The original L4 had a separate kernel stack for each thread (allocated inside the thread's TCB). This allowed obtaining the present thread's TCB by masking the low-order bits off the kernel stack pointer.

Process-orientation was tightly coupled with allocating all TCBs 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 destination thread ID: If the caller supplies an invalid ID, the lookup may access an unmapped TCB, triggering a page fault, which the kernel handles by aborting the IPC. If no fault happened, the validity of the thread ID can be established by comparing the caller-supplied value with the one found in the TCB. (Note that original L4's thread IDs had version numbers, which changed when the thread was destroyed and re-created. This was done to make thread IDs unique in time. Recording the current ID in the TCB allowed detecting stale thread IDs.)

Both features come at a cost: The many kernel stacks dominate the per-thread memory overhead, 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. TLB pressure is not much of an issue on x86 processors with untagged TLBs, as any cross-address-space IPC will implicitly flush the TLB anyway. However, RISC processors had tagged TLBs early on.

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

Haeberlen [2003] built the first event-based (single-stack) variant of L4 based on Hazelnut while experimenting with designs for utilizing virtual memory in the kernel. As such, the Pentium-based evaluation was not focused substantiating an event-based kernel, however the results were promising with a reduction in kernel memory consumption, and improvement of 20% in IPC micro benchmarks compared to a (still immature) version of Pistachio.

Later, Warton [2005] performed a specifically targeted analysis of the performance tradeoff of event vs. process kernels, and implemented an event-based (single-stack) kernel with continuations in Pistachio. An evaluation on an ARMv5 processor showed that while the process kernel outperformed the event kernel on micro benchmarks by small margins (generally within 1%), on a multi-tasking workload (AIM7) the event kernel had a performance advantage of about 20%.

Warton also showed that while the TCB size (ignoring the stack) of the single-stack kernel was more than twice that required for the process kernel (the storage overhead of continuations), taking the stack into account, the event kernel's per-thread memory use is a quarter of that of the process kernel. Note that this space problem was less severe in the original L4, as it was all assembler code and did not use standard stack frames.

Similarly, Nourai [2005] analysed the trade-offs of virtual vs. physical addressing of TCBs. He implemented physical addressing, also in Pistachio, although on a MIPS64 processor. He found little if any differences in IPC performance in micro-benchmarks, but significantly better performance of the physically-addressed kernel on workloads that stressed the TLB. The 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 superpage mappings. Still Nourai's results convincingly indicate that there is no significant performance benefit from the virtually-addressed TCBs.

Together, these results lead to L4-embedded moving to an event-based design with physically-addressed kernel data, and seL4 followed suit.

Abandoned: Process kernel and virtual TCB addressing.

3.2.2 IPC timeouts

Any IPC operation in the original L4 had a timeout value to protect against denial of service. (This feature could also be used for timed sleeps by waiting on a message from a non-existing thread.)

An IPC syscall actually specified 4 timeouts: one to limit blocking until start of the send phase, one to limit blocking in the receive phase (measured from the time of completion of the send), and two more to limit blocking on page faults during the send and receive phases (of long IPC). Timeouts values were encoded in a compressed floating-point format that supported the values of zero, infinity, and a range of finite values ranging from one millisecond to weeks.

Timeouts added significant complexity for decoding the floating-point values as well as for managing wakeup lists. They were an implied requirement of long IPC: 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. The server could use the page-fault timeout to protect against this denial-of-service attack.

Practically, however, timeouts were of little use. There is no theory, or even good heuristics, for choosing timeout values in a non-trivial system, and in reality, only the values zero and infinity were used: A client sends and receives with infinite timeouts, while a server waits for a request with an infinite but replies with a zero timeout. (The semantics of the `call` operation specifies an atomic switch from send to receive phase, so a client using `call` to invoke the server is guaranteed to be ready to receive.) Traditional watchdog timers can be used for detecting unresponsive IPC interactions much simpler than through IPC timeouts.

Having abandoned long IPC, L4-embedded 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.

Abandoned: Timeouts.

3.2.3 Lazy scheduling

In the rendezvous model of IPC, a thread's state frequently alternates between runnable and blocked. This implies frequent queue manipulations, moving threads between the run queue and a waiting 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 run queue, in the hope that it will unblock soon. Similarly, threads which become runnable are not dequeued from their timeout wakeup queue and not entered into the run queue.

When the scheduler is invoked on a time-slice preemption, it traverses the run queue until it finds a thread that is really runnable, and removes the ones that are not.

The drawback of lazy scheduling was discovered 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!

However, the removal of IPC timeouts (and thus wakeup queues) enabled an equivalent optimisation, referred to as *Benno scheduling*, which does not suffer from pathological timing behaviour: While a thread which blocks during IPC is removed from the run queue, a thread which is unblocked during an IPC is *not* inserted into the run queue. At any time, there can be only one such thread (per processor) with is runnable but not in the ready queue: the currently running one. At preemption time, the kernel inserts the preempted thread into the run queue if needed, which re-establishes the invariant that the run queue contains all runnable threads and no others.

Replaced: Lazy scheduling by Benno scheduling.

3.2.4 Direct process switch

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

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

On the other hand, if a server replies to a waiting client (execution the `reply-and-wait` syscall), and the server has a request waiting from another client, it makes sense to continue the server 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.

Replaced: Direct process switch subject to priorities.

3.2.5 Register messages

In its desire to minimise IPC costs through zero-copy message transfer (see Section 3.1.5), original L4 packed as many message words into registers, and for cases were these were insufficient, supported the "long IPC" semantics with buffers pointed to by the register messages.

The size of the "direct" (in-register) message was therefore highly dependent on the architecture. It also changed between ABI versions, as changes to system-call arguments consumed or freed additional registers.

Pistachio avoids this problem by specifying a moderate-size (configurable in the range 16–64 words) set of *virtual message registers*. The implementation mapped some of these to physical registers, the rest was contained in a pinned part of the address space called the *user-level TCB* (originally introduced to support ultra-fast user-level IPC [Liedtke and Wenske 2001]). The pinning ensures a register-like semantics without the possibility of a page fault. Inlined access

functions hide the distinction between physical and memory-backed registers from the user. seL4 uses a variant of this approach, with a pinned message buffer per thread.

Replaced: Physical by virtual message registers.

3.2.6 Non-standard calling convention

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

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

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

The assembler fast path approach was discarded in the OKL4 Microvisor because of the high maintenance cost of assembler code, which was not justified by a performance improvement of typically less than 10%. It was also not suitable for seL4, as the verification framework could only deal with C code [Klein et al. 2009], and the team wanted to verify the kernel's functionality as completely as feasible. Assembler code was restricted to the bare minimum, and therefore calling-convention conversions were ruled out as well, forcing the team to adopt the tool chain's standard calling conventions.

seL4 is also highly dependent on fast-path code to obtain competitive IPC performance, but the fast paths must now be implemented in C. However, it turned out that by carefully hand-crafting the fast path, including providing hints to the compiler about (verified) invariants that the compiler is unable to determine by static analysis, highly-competitive IPC latencies can be achieved [Blackham and Heiser 2012].

In fact, the finally achieved latency of 185 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! While this is partially a result of the simplified seL4 ABI and IPC semantics, it shows that assembler implementations are no longer justified by performance arguments.

Abandoned: Non-standard calling conventions and assembler code for performance.

3.2.7 Non-portability

Liedtke [1995] makes the point that a microkernel implementation should not strive for portability, as a hardware abstraction introduces overheads and hides hardware-specific optimisation opportunities. He cites subtle architectural changes between the "compatible" i486 and Pentium processors resulting in shifting tradeoffs and implying significantly changes in the optimal implementation.

This argument was debunked by Liedtke himself, with the high-performance yet portable Hazelnut and Pistachio kernels. Highly-tuned (and sometimes micro-architecture-specific) fast paths can achieve performance that is at par with the original L4 (see Table 1) while keeping 80–90% of the implementation architecture-agnostic.

Abandoned: Non-portable implementation.

3.3 Is it still L4?

Having abandoned so many of the design and implementation features that seemed to define the original L4, one naturally asks "in which sense is it still L4?"

On the one hand, the answer to this question is probably not terribly important. On the other hand, seL4 is very much considered part of the "L4 family", both inside the seL4 team and outside, so there is probably some reason for this sentiment.

First of all, seL4 is the (presently) latest data point in the evolution of high-performance microkernels derived from the original L4, even though seL4 presents the arguably largest step in this evolution, given its significant departure from traditional L4 ways in regards to resource management.

Then seL4 still follows the original philosophy of minimality of concepts and mechanisms without foregoing generality, and tight coding aimed at optimising critical IPC operations. In some ways, seL4 is probably more economical in its abstractions than any previous L4 kernel, although this is hard to quantify.

4. seL4 Design

While L4 microkernels generally had a strong average-case performance focus (and, in the case of Fiasco, a focus on real-time performance [Härtig and Roitzsch 2006]), security (or more specifically, the design and assurance of it) has traditionally received only modest attention. seL4, in contrast, is designed from the beginning to support formal reasoning about security and safety, while maintaining the L4 tradition of minimality and performance.

Specific high-level requirements were:

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

2. Data access and authority can be confined.
3. The kernel itself (for its own data structures) adheres to the authority distributed to applications, including the consumption of physical memory. We examine this more closely in Section 4.2.
4. All kernel objects can be reclaimed independent of any other kernel objects. For example, page-table memory can be reclaimed without having to destroy the corresponding address space. We'll discuss this in Section 4.3.
5. All operations are "short" in execution time, or are preemptible in short time (to support real-time analysis). Here, *short* either stands for constant time or linear in a small number. For example, object initialisation is linear in size, where size is less than a page. This is a requirement for reasoning about the safety of real-time systems, and is examined further in Section 4.4.
6. Performance is not significantly worse than the fastest L4 kernels (say within 10%).

We now examine goals 1–5 (performance has been discussed in the previous section), discuss our approach to achieving them, and the degree to which we succeeded.

4.1 Security Focus

Requirements (1) and (2) above are well understood in the security community, and seL4 borrows from capability systems such as CAP [Needham and Walker 1977], KeyKOS [Bromberger et al. 1992] and EROS [Shapiro et al. 1999]. All seL4 system calls involve a capability invocation, which authorizes the call to the kernel.[6] They are also relatively simple by virtue of being low-level operations on few kernel objects.

User-mode memory access and instruction execution are explicitly (although indirectly) included in the security model: user-mode access is mediated by page tables which map physical memory, and the security model treats page tables as capability storage (and page table entries as capabilities).

To facilitate revocation of resources, seL4 tracks capability derivations in a *capability derivation tree* (CDT); the tree pointers are stored inside the capabilities themselves to avoid memory allocation for book keeping.

The most characteristic aspect of seL4's security design is its approach to connecting the kernel's memory consumption with authority, which we examine in Section 4.2. It effectively makes kernel memory subject to the user-mode authority mechanisms, and thus extends user-mode authority separation to kernel resources. The success of this approach was recently demonstrated, when it enabled first a proof of integrity enforcement [Sewell et al. 2011], followed by a proof of non-interference (and, by implication, confidentiality enforcement) [Murray et al. 2013], both a first for a general-purpose OS kernel.

One of the more encouraging outcomes is summarised in the following statement, which implies that the design was sufficiently clean and easy to understand to get it right on the first attempt:

> During this proof of noninterference we did not find any information-flow problems in the seL4 kernel that required code changes, as we had hoped given the previous intensive work on proving functional correctness and integrity [Murray et al. 2013].

4.2 Memory Management

4.2.1 The kernel memory problem

The advantage of having all authority represented by capabilities is that it facilitates reasoning about the security state of a system configuration by reasoning about the distribution of capabilities. However, this is only sufficient if capabilities represent *all* authority in the system.

The kernel's own memory management is an area where there is a potential gap between the authority to use a kernel service, and the memory consumed in providing the service. For example, the ability to map virtual pages into an address space may indirectly allocate page tables to record the mappings.

The consequence is a potential for denial-of-service attacks by forcing the kernel into excessive memory consumption. Original L4 had this problem [Liedtke et al. 1997b], as do other systems, where it is not possible to precisely reason about in-kernel memory consumption based on authority to access kernel services. As a result, reasoning about application authority alone is insufficient to strictly enforce in-kernel memory isolation.

Kernels that manage memory as a cache of user-level content only partially address this problem. While caching-based approaches remove the opportunity for denial-of-service 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. A precise relationship between the authority (in our case capabilities) and the consumption of kernel memory is required to connect reasoning about authority (and isolation thereof) and the consumption of in-kernel memory.

In essence, this is the third in the above list of design requirements.

4.2.2 Approach

To connect the authority an application possesses with kernel memory consumption, we (1) make all in-kernel allocated objects first-class objects in the ABI, and (2) ensure that no objects change their size after creation. This integrates kernel objects into the general protection system, and makes them subject to capability-mediated authority. The approach is inspired by hardware-based capability systems, specifi-

[6] The single exception is the `yield()` system call, which triggers the scheduler.

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

Table 3. seL4 kernel objects.

cally CAP [Needham and Walker 1977] where hardware-interpreted capabilities directly refer to memory.

The approach is made feasible by sL4's microkernel design, with a small set of abstractions and thus object types. Table 3 summarises seL4's object types.

Higher-level concepts, such as a *process*, are user-level abstractions created by associating different objects. For a process this might be TCBs, Cnodes, PageTables, a PageDirectory and Frames.

4.2.3 Memory allocation

Making all kernel data structures first-class objects in the API alone is insufficient to align authority with memory consumption; a model of creation of those objects is required. The model must be able to represent the authority to memory regions that are used by an application to create the kernel objects. Additionally, *the model must ensure the kernel integrity requirement that no two objects overlap.*

In seL4 we enforce this by the requirement that any objects are allocated from Untyped Memory (UM) objects. UM represents power-of-2 sized and aligned regions of physical memory. An UM capability represents the authority to a region of memory, that an application can use to create a *typed* memory object, i.e. an object that supports a specific kernel service. The `retype()` method of a UM capability is invoked with the type of the required memory object. In response, the kernel creates a new object and a capability with full authority over the object (provided that the request satisfies the integrity condition that the new object is fully contained within the UM and does not overlap with other objects).

At boot time, seL4 preallocates the memory required for its own code, data, and stack (which is strictly bounded). It then creates an initial "process" (consisting of a minimal set of objects according to a startup protocol). The kernel deposits in that process's Cnode the capabilities to the process' own objects, as well as UM caps to all remaining (so far unused) memory. The process then has all the authority in the system. It can, for example, partition its UM, and start up new processes in each of the partitions, thus delegating management of each partition to its initial process.

Bedsides the requirement of non-overlapping objects, *the kernel enforces a second key integrity property: User-mode code cannot uncontrollably modify kernel objects.* The kernel ensures this by ensuring that only Frame objects can be mapped into the virtual address space of applications (i.e. inserted into a PageTable). The kernel never accesses Frame objects (except for the buffer of virtual registers, which is used for passing system-call arguments between kernel and user).

4.2.4 Memory Re-use

For simple static systems, the model presented thus far is sufficient for allowing applications to allocate kernel objects for obtaining services, to distribute that authority to clients, and to explicitly manage and limit the consumption of kernel memory. Dynamic systems furthermore need to be able to reclaim and re-use memory.

Reclamation implies a further integrity requirement: no reference that would allow access to an object must remain after reclamation. Besides imposing invariants on kernel data structures, this means that the kernel must ensure that there are no capabilities left to a typed object that is being reclaimed.

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

Owing to the derivation hierarchy, revocation is a long-running operation (bounded only by the amount of physical memory). It must therefore be preemptible, more on that in Section 4.4.

4.3 Object Independence

The resource management model introduced above is sufficient for reasoning about the distribution of both authority and memory using the distribution of capabilities. The creation of kernel objects is strictly limited by access to UM, and memory can be reclaimed safely by revocation of capabilities.

However, the power of this model, together with the low-level nature of seL4 kernel services, requires a further invariant: *every object must be reclaimable in isolation.* Ker-

nel objects inherently reference other objects. For example, a TCB belongs to an address space, and therefore references a PageDirectory, which references PageTables.

While it is intuitive that removing all capabilities to an object will result in the object being reclaimed, it is not obvious how all references from other kernel objects will be removed if an object is destroyed. While the CDT allows locating all of an object's caps, some of these internal references may not be caps but just normal pointers.

A traditional OS avoids this problem by defining a particular order of reclamation of various in-kernel data structures. For example, killing a process might require first freeing physical frames, then leaf page tables, then the page directory, prior to freeing the process control block. In seL4, such an ordering is not enforced (or enforceable), as reclamation of objects is at the discretion of user level code, and the kernel cannot rely on sane userland behaviour.

In order to ensure object independence, composable objects adhere to the following invariant. *The capabilities (authority) used to couple objects must also facilitate decoupling the objects when authority is revoked.* The seL4 design has three scenarios relevant to maintaining this invariant.

Objects may refer to each other with internal pointers. For example, if a thread blocks waiting on an Endpoint, the TCB is enqueued into the Endpoint's doubly-linked waiting list. As a result, the TCB has a reference to the Endpoint, and vice versa. If either object is reclaimed, the reference must be removed and the surviving object made consistent. This approach is only applicable where the object format is not dictated by hardware.

Objects contain capabilities to other objects. In this case, the objects are coupled by inserting a copy of a capability, e.g. a TCB contains a PageDirectory capability. In this case, revocation of the capability automatically decouples the object.

The capability contains the book-keeping data. Some capabilities have space that can be used for book-keeping. For example, when a Frame is mapped into an address space, the Frame capability is updated to record the location of the PageTable entry (PTE) affecting the mapping. A Frame cap can only hold one such PTE reference, so if Frames are shared, their caps must be explicitly copied.

Capability-based book keeping alone is insufficient to decouple the composition should a PageTable be reclaimed. The book-keeping data of the cap of each Frame mapped in the PageTable must be updated to remove the reference to the PageTable. Thus we need to locate the Frame cap using the physical frame address contained in the PTE. Older L4 kernels had an analogous problem with their *mapping database*, which contained *shadow page tables*, which in turn held back-links to the mappings used to establish the PTE.

In seL4 we can solve this problem by using the existing capability derivation tree, in which capabilities are stored in a total order based on physical address, depth of derivation, and in the case of a Frame, PTE address. Given a particular PTE, we can find the Frame cap used to install the mapping by probing the CDT, a $\log(n)$ operation, where n is the number of capabilities in the CDT.

The capability-based book keeping approach is only used for PageTable objects and Frame objects, as the format of the PageTable object is dictated by hardware and does not support explicit object references.

4.4 Preemption

Like most L4 kernels, the seL4 kernel executes with interrupts disabled. Historically this approach has been used to maximise average-case performance. In the case of seL4, there is an additional reason for this choice: it keeps concurrency out of the kernel, and was needed to make formal verification tractable [Klein et al. 2009].

A key target domain for seL4 deployment is that of safety-critical systems, many of which are of a hard real-time nature, requiring hard bounds on the latency of interrupt delivery. The seL4 design therefore uses preemption points to allow potentially long-running operations to be postponed while a higher-priority thread executes. As a consequence, all global kernel invariants must be re-established by the end of kernel operations, *and* when any operation is preempted. This requires kernel careful design to ensure that suitable preemption points can be inserted into long-running operations.

The kernel operations that are potentially long-running are:

- object initialisation (especially large memory frames or Cnodes);
- revocation of capabilities;
- decoupling of objects from reclaimed objects, e.g. removing all the threads waiting on an IPC endpoint, or removing all frames from a page table.

With the right design, revocation and decoupling operations (all related to object deletion paths) have natural preemption points. For example, the CDT in seL4 is designed to ensure that deletion can be decomposed into a series of constant-time operations, where all kernel invariants are maintained between each step. We call such a design *incrementally consistent*.

Some versions of seL4 have contained data structures that do not exhibit incremental consistency, resulting in long-running operations that cannot be decomposed. Such operations (e.g., choosing a new thread in lazy scheduling) are sources of large worst-case interrupt latencies which are detrimental to real-time performance. Refinements of seL4's design have strived to eliminate these by employing the incremental consistency principle [Blackham et al. 2012].

In seL4, system calls are made restartable by storing progress explicitly in either the disabled capability (called a *zombie* in seL4) or the object itself, or implicitly by the

entries remaining on a queue. Using restartable system calls, all pre-conditions required for the operation (which may have changed while being preempted) are re-established on restart.

4.5 Notifications

The seemingly innocuous semantics of asynchronous notification demonstrates how easy it is to get a design wrong. In the initial version, the send operation OR-ed a sender-supplied bit mask with a single-word notification field in the TCB. The receiver could poll that field, or block until a notification was received. In the latter case, the notification word was atomically delivered to the receiver and simultaneously reset to receive new notifications.

With the move to endpoints as IPC destinations, an explicit asynchronous endpoint capability invocation is required to receive a notification. This meant that a thread is able to either wait for synchronous or for asynchronous IPC, but not both concurrently. This design was a mistake: A server dealing with asynchronous I/O and synchronous clients required multiple threads with complex synchronization, even for otherwise trivially simple systems.

As a consequence we changed the design. An asynchronous endpoint is now *bound* to a particular thread, allowing that thread to be notified of pending notifications while blocked on a synchronous endpoint. This feature enables event-based service implementations that manage both synchronous clients and asynchronous I/O.

4.6 Open issues

4.6.1 Time

One challenge that has been around since the early day of L4 is finding an appropriate abstraction for *time*. Fundamentally, the kernel abstracts space (memory) and time (execution on the CPU). The seL4 abstraction for the former is satisfactory. For time, seL4 has not really progressed from the original L4.

The scheduling model of the original L4, hard-priority round-robin, is still alive, despite being a gross heresy against the core microkernel religion of policy-freedom. All past attempts to export scheduling policy to user level have failed, generally due to intolerable overheads.

On the one hand, 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 environment-specific plugins. On the other hand, the formal verification of seL4 creates a powerful disincentive to changing the kernel, it really reinforces the desire to have a single platform for all usage scenarios. Hence, a policy-free approach to dealing with time is as desirable as it has ever been.

4.6.2 Multicore and Verification

Operating systems and concurrency usually go hand-in-hand. However, formal software verification using interactive theorem proving avoids concurrency wherever possible. The seL4 design explicitly avoids concurrency using an event-based execution model to improve the tractability of verification. However, the ubiquity of multicore processors creates a challenge: how can one utilize the available cores while retaining some, if not all, of the formal guarantees?

Our first step to resolving this issue is the *clustered multikernel*, a hybrid of a big-lock kernel, and a restricted variant of a multikernel [Baumann et al. 2009] (no memory migration is permitted between kernels). The clustered multikernel avoids concurrency in the majority of kernel, which enables some of the formal guarantees to continue hold under some assumptions. The most significant assumptions are that (1) the lock itself, and any code outside of it, is correct and race free, and that (2) the kernel is robust to any concurrent changes to memory shared between the kernel and user-level (for seL4 is this is only block of virtual IPC message registers).

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

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

5. Conclusions

It is rare that a research operating system has both a significant developer community 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 more than half a dozen from-scratch implementations. We see this as a great opportunity to reflect on the principles and know-how that has stood the test of time, and what has failed to survive increased insights, changed deployment scenarios and the evolution of CPU architectures.

We find that the most general principles behind L4, minimality and a strong focus on performance, still remain relevant and foremost in the minds of developers. Specifically we find that the key microkernel performance metric, IPC latency, has remained essentially unchanged (in terms of clock cycles), as far as comparisons across vastly different ISAs and micro architectures have any validity, in stark contrast to the trend identified by Ousterhout [1990] just a few years before L4 was created. Furthermore, and maybe most surprisingly, the code size has essentially remained constant, a rather unusual development in software systems.

In terms of systems design, device drivers at user-level, highly optimised *fast paths* for performance-critical common cases, and asynchronous notifications augmenting synchronous IPC, are uncontroversial and are common features of modern L4 variants.

The rise of virtualization (Linux as standard middleware), memory-limited embedded applications, and the general desire to simplify, has left its traces. A number of approaches and features have been dropped and others have evolved to better support current use cases. Specifically, recursive address spaces, large amounts of assembly code, and complex IPC features, such as timeouts, have been removed. Various implementation tricks have become obsolete as better (often simpler) approaches have been discovered. Other implementation techniques have lost relevance with the evolution of CPU architecture. However, TLB footprint has a greater impact on microkernel performance than it did 20 years ago, giving rise to techniques that favor minimizing that footprint.

Security focus and formal verification has driven the evolution of the most recent L4 variant: seL4. This kernel introduces a model for explicit user-level control of kernel memory, and it has been formally verified to be functionally correct and to possess the desired high-level security and safety properties. We think it is a great testament to the brilliance of the 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].

A case in point is the user-level management of kernel objects, which is the key to provable isolation provided by seL4. The requirement to be able to reclaim any object independently of any others, even where objects are coupled by in-kernel data structures, is feasible due to the small number of object types resulting from a consistently simplicity-focussed design. It is hard to imagine how this cold be achieved in a kernel providing a more complex API.

There is one concept that has, so far, resisted any satisfactory abstraction: *time*. Modulo minor variations, L4 kernels still have a priority-based round-robin scheduler—the last major holdout of policy in the kernel. This probably represents the largest limitation of generality of L4 kernels. We hope it will not take another 20 years to find a solution.

References

M. T. Alexander. Organization and features of the Michigan terminal system. In *AFIPS Conf. Proc., 1972 Spring Joint Comp. Conf.*, pages 585–591, 1972.

A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *22nd SOSP*, Big Sky, MT, USA, Oct 2009. ACM.

B. Blackham and G. Heiser. Correct, fast, maintainable – choose any three! In *3rd APSys*, pages 13:1–13:7, Seoul, Korea, Jul 2012. doi: 10.1145/2349896.2349909.

B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser. Timing analysis of a protected operating system kernel. In *32nd RTSS*, pages 339–348, Vienna, Austria, Nov 2011. doi: 10.1109/RTSS.2011.38.

B. Blackham, Y. Shi, and G. Heiser. Improving interrupt response time in a verifiable protected microkernel. In *7th EuroSys Conf.*, pages 323–336, Bern, Switzerland, Apr 2012. doi: 10.1145/2168836.2168869.

P. Brinch Hansen. The nucleus of a multiprogramming operating system. *CACM*, 13:238–250, 1970.

A. C. Bromberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The KeyKOS nanokernel architecture. In *USENIX WS Microkernels & other Kernel Arch.*, pages 95–112, Seattle, WA, USA, Apr 1992.

J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *14th SOSP*, pages 120–133, Asheville, NC, USA, Dec 1993.

M. Condict, D. Bolinger, D. Mitchell, and E. McManus. Microkernel modularity with integrated kernel performance. Technical report, OSF Research Institute, Jun 1994.

D. Elkaduwe, P. Derrin, and K. Elphinstone. Kernel design for isolation and assurance of physical memory. In *1st WS Isolation & Integration Emb. Syst.*, pages 35–40, Glasgow, UK, Apr 2008. ACM SIGOPS. doi: 10.1145/1435458.

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

A. Haeberlen. Managing kernel memory resources from user level. Diploma thesis, System Architecture Group, University of Karlsruhe, Germany, Apr 2003. URL http://os.ibds.kit.edu/english/97_639.php.

H. Härtig and M. Roitzsch. Ten years of research on L4-based real-time systems. In *8th Real-Time Linux WS*, Lanzhou, China, 2006.

H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *16th SOSP*, pages 66–77, St. Malo, France, Oct 1997.

G. Heiser and B. Leslie. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *1st APSys*, pages 19–24, New Delhi, India, Aug 2010.

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

T. Jaeger, K. Elphinstone, J. Liedtke, V. Panteleenko, and Y. Park. Flexible access control using IPC redirection. In *7th HotOS*, Rio Rico, AZ, USA, Mar 1999.

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

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM. doi: 10.1145/1629575.1629596.

A. Lackorzynski and A. Warg. Taming subsystems: capabilities as universal resource access control in L4. In *2nd WS Isolation &*

Integration Emb. Syst., pages 25–30, Nuremburg, Germany, Mar 2009.

R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in HYDRA. In *5th SOSP*, pages 132–140, 1975.

J. Liedtke. Improving IPC by kernel design. In *14th SOSP*, pages 175–188, Asheville, NC, USA, Dec 1993a.

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

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

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

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

J. Liedtke and H. Wenske. Lazy process switching. In *8th HotOS*, pages 15–20, Schloss Elmau, Germany, May 2001.

J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay. Two years of experience with a μ-kernel based OS. *ACM Operat. Syst. Rev.*, 25(2):51–62, Apr 1991.

J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger. Achieved IPC performance (still the foundation for extensibility). In *6th HotOS*, pages 28–31, Cape Cod, MA, USA, May 1997a.

J. Liedtke, N. Islam, and T. Jaeger. Preventing denial-of-service attacks on a μ-kernel for WebOSes. In *6th HotOS*, pages 73–79, Cape Cod, MA, USA, May 1997b. IEEE.

T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symp. Security & Privacy*, Oakland, CA, May 2013.

R. Needham and R. Walker. The Cambridge CAP computer and its protection system. In *6th SOSP*, pages 1–10. ACM, Nov 1977.

A. Nourai. A physically-addressed L4 kernel. BE thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Mar 2005. Available from publications page at http://ssrg.nicta.com.au/.

J. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *1990 Summer USENIX*, pages 247–56, Jun 1990.

T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. seL4 enforces integrity. In M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *2nd ITP*, volume 6898 of *LNCS*, pages 325–340, Nijmegen, The Netherlands, Aug 2011. Springer. doi: http://dx.doi.org/10.1007/978-3-642-22863-6_24.

J. S. Shapiro. Vulnerabilities in synchronous IPC designs. In *IEEE Symp. Security & Privacy*, Berkeley, CA, May 2003. URL citeseer.ist.psu.edu/shapiro03vulnerabilities.html.

J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *17th SOSP*, pages 170–185, Charleston, SC, USA, Dec 1999. URL http://www.eros-os.org/papers/sosp99-eros-preprint.ps.

M. Warton. Single kernel stack L4. BE thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Nov 2005.