

The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors

Gernot Heiser
Open Kernel Labs and NICTA and
University of New South Wales
Sydney, Australia
gernot@ok-labs.com

Ben Leslie
Open Kernel Labs
Sydney, Australia
benno@ok-labs.com

ABSTRACT

We argue that recent hypervisor-vs-microkernel discussions completely miss the point. Fundamentally, the two classes of systems have much in common, and provide similar abstractions. We assert that the requirements for both types of systems can be met with a single set of abstractions, a single design, and a single implementation. We present partial proof of the existence of this convergence point, in the guise of the *OKL4 microvisor*, an industrial-strength system designed as a highly-efficient hypervisor for use in embedded systems. It is also a third-generation microkernel that aims to support the construction of similarly componentised systems as classical microkernels. Benchmarks show that the microvisor's virtualization performance is highly competitive.

Categories and Subject Descriptors

D.4.7 [Operating Systems]: Organization and Design—*Real-time systems and embedded systems*

General Terms

Design, Performance

Keywords

Microkernels, hypervisors, virtual machines, real-time systems and embedded systems

1. INTRODUCTION

The merits of microkernels have been debated for a long time. Popular in the 1980s, they fell into disrepute in the early '90s as systems built on top failed to perform. Arguments that the performance problems were inherent in the microkernel approach [CB93] were refuted by Liedtke [Lie95] identifying design and implementation shortcomings of these first-generation kernels, and it was demonstrated that the second-generation (2G) L4 microkernel could be used as a hypervisor to virtualize Linux with an overhead of about 5–7% [HHL⁺97]. Commercial microkernels are now deployed at a large scale, including QNX [LG09] for high-availability, Green Hills Integrity [GHS] for high security, and the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

APSys 2010, August 30, 2010, New Delhi, India.

Copyright 2010 ACM 978-1-4503-0195-4/10/08 ...\$10.00.

high-performance OKL4 microkernel for mobile devices [OKL4] (with more than 750 million instances shipped to date).

Curiously, this did not stop a new debate starting a number of years back, this time pitting microkernels against hypervisors, as if they were disjoint classes of systems. Some of these arguments [HWF⁺05] we have earlier shown to be based on false premises [HUL06], others [AG09] are based on an outdated, 1980's view of microkernels, oblivious to the last 15+ years of microkernel research.

We assert that these arguments completely miss the point. Microkernels and hypervisors are both designed as low-level foundations for larger systems, although with different objectives. This does not mean that these objectives are irreconcilably at odds. In fact, we argue that we can construct a type of kernel that satisfies the combined objectives of microkernels and hypervisors. We call such a kernel a *microvisor*, and present the OKL4 microvisor as a representative.

In the next section we revisit the motivations for microkernels and hypervisors, and the resulting objectives and designs. In Section 3 we explore the intersection of these objectives and present the OKL4 microvisor as a representative. Section 4 presents initial performance data and Section 5 discusses related work.

2. MICROKERNELS AND HYPERVISORS

2.1 Microkernels

While the term *microkernel* was not coined until a decade and a half later, the basic concept goes back to Brinch Hansen's *nucleus* [BH70]. The basic idea is to reduce the kernel code to fundamental mechanisms, and implement actual system services (and policies) in user-level servers. The microkernel is supposed to be general enough to support the construction of arbitrary systems on top.

The modern microkernel concept is captured in Liedtke's Minimality Principle: *A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent implementation of the system's required functionality* [Lie95].

The original driver behind microkernels is generality and flexibility, based on a separation of policy and mechanisms [LCC⁺75]. A more modern motivation is the design of systems with a minimal trusted computing base, in order to aid assurance of safety- and security-critical systems [HPS04]. The formal verification of the seL4 microkernel [KEH⁺09] shows that microkernels can be small enough make a formal proof of functional correctness feasible.

Moving services out of the kernel into servers makes client-server communication performance-critical, and as such a microkernel needs a very fast inter-process communication (IPC) mech-

anism. Consequently, IPC performance used to be the focus of microkernel research [Lie93].

2.2 Hypervisors

Hypervisors are even older, they go back to the 1960s. The original motivation was an early form of legacy re-use: With the advent of more powerful computers, the original single-user single-task model of operating systems was no longer sufficient. Virtualization supported the concurrent execution of programs by putting each into a separate virtual machine (VM), together with its single-tasking executive. Virtualization went out of fashion in the '70s with the widespread availability of multitasking and multi-user OSes, until its renaissance in the last decade, mostly driven by the resource-management deficiencies of most modern OSes.

According to the classic definition, a *VM is an efficient, isolated duplicate of a real machine* [PG74]. The VM is implemented by a hypervisor (or virtual-machine monitor, VMM) which provides virtual resources either by (temporarily) mapping them to physical resources, or emulating them. The former is efficient, the latter results in overhead, and so is avoided where possible.

While originally virtualization was understood to be *pure* (virtual resources are essentially indistinguishable from real ones), hardware limitations and efficiency reasons led to widespread use of *para-virtualization*, which presents a modified hardware ABI to which the guest OS is ported. A hypervisor supporting para-virtualization provides extra APIs, called *hypercalls*, which generally are more high-level than the hardware ABI.

2.3 Comparison

In summary, microkernels aim to provide a minimal layer of privileged software, while hypervisors aim to replicate and multiplex hardware resources. Both have an inherent need to abstract the hardware, although with different emphasis.

For a hypervisor it is fundamentally important that the abstract (“virtual”) resources look as much like the concrete (“real”) ones as possible, while implementation size is not a primary concern. For a microkernel, similarity of real and abstract resources is not a driver, but in reality the semantic gap cannot become very large without violating the minimality requirement.

It should now not be overly surprising that the abstractions end up being quite similar. Fundamentally, both types of kernels must provide abstractions for:

Memory: The hardware managing memory is the MMU, and hypervisors provide a virtual MMU, or virtualize the MMU’s software interface, the page tables. Microkernels typically abstract memory via the more OS-like concept of an address space. However, in (2G) L4 microkernels, address spaces are simply containers for mappings, and as such present a minimal abstraction of page tables.

Execution: The CPU must be multiplexed between different VMs, and therefore hypervisors provide the concept of virtual CPUs, which are multiplexed on the real CPU (by scheduling VMs). 2G microkernels typically abstract execution time as threads (as in L4) or scheduler activations (as in K42 [KAR⁺06]). All are minimal abstractions that allow scheduling of activities. Both views must associate similar attributes with their CPU abstractions: priorities and time slices. The main difference is that the hypervisor abstraction retains the concept of multiple (virtual) privilege levels while a microkernel treats all user-level activities as peers.

I/O Hypervisors traditionally virtualize devices by exporting a (usually simplified) device interface to the guest OS, with

the actual device driver residing inside the hypervisor—this approach supports multiplexing the device (like other resources). Sometimes a device is made directly accessible to the guest by mapping device registers into the guest’s memory and virtualizing interrupts (the hypervisor up-calls the guest’s interrupt handler routine). This makes multiplexing of stateful devices difficult or impossible and, in the case of DMA-capable devices, can only be done securely with additional hardware support (I/O MMU).

Microkernels, in contrast, tend to run device drivers as separate user-level processes, which communicate with the rest of the system using IPC. This is effectively a combination of the approaches discussed for hypervisors, and supports device sharing without making drivers part of the microkernel, but comes at the expense of additional context switches.

Communication Subsystems need to communicate. VMs communicate like real machines—via networks. Hence, hypervisors traditionally provide virtual networks, which are based on the existing virtual-device abstraction, running standard networking stacks. Microkernels, in contrast, are designed for systems decomposed at a much finer granularity, and therefore offer highly-optimised message-based IPC primitives. Following Liedtke [Lie93], 2G microkernels typically minimise overheads by providing a synchronous (blocking and unbuffered) IPC primitive.

In summary, there is not much difference in the first three points. As far as I/O is concerned, hypervisors have in recent years moved closer to microkernels. Xen [BDF⁺03], for example, runs its administrative functions in a privileged VM called Domain0, which hosts a Linux guest. Domain0 is also used as a source of device drivers, which therefore run in user mode, as with a microkernel (although all the Domain0 drivers run in the same address space, that of the Linux guest). The driver (excuse the pun) behind this development is the cost of re-implementing or adapting device drivers: Domain0 uses unmodified drivers from the Linux guest.

This in turn puts pressure on inter-VM communication costs. Only a few years back, hypervisor proponents belittled the microkernel community [HWF⁺05] for their efforts in optimising IPC. During the discussion following the presentation of that paper, a prediction was made that within two years the hypervisor community would be publishing papers on improving inter-VM communication overheads. In fact, a string of papers have since proposed such inter-VM communication primitives. Some of these [ZMRG07, WWG08] showed performance that was well below what had been achieved in comparable microkernel-based setups [LCFD⁺05, LUSG04] while others sacrifice isolation for performance [BSR09].

2.4 Trends

As the above discussion shows, microkernels and hypervisors are fundamentally nowhere near as different as some of the literature would make us believe. And in fact, the similarities are growing.

On the one hand, hypervisors are becoming more microkernel-like. The tendency to move drivers into userland, as a way to reuse legacy drivers, is one indication. Furthermore, the hypervisor community is becoming increasingly aware of the attack surface offered by a big hypervisor, and the high rewards for cracking it [Rut08, SK10].

On the other hand, microkernels are increasingly used to support virtualization. The reason is also legacy support: even highly security- or safety-critical systems increasingly face the need to

support legacy OS environments—in order to support GUIs, provide familiar application environments, and run standard networking and file system stacks. There is barely a commercial microkernel which does not double as a hypervisor.

3. ENTER THE MICROVISOR

3.1 Combining the models

Under the circumstances it makes sense to revisit the microkernel-hypervisor issue. Specifically, we ask whether it is possible to achieve the objective of *both* classes of kernels with a single set of abstractions, a single design, and a single implementation. Specifically, can we build a single kernel which provides platform virtualization with the efficiency of the best hypervisors, while at the same time achieving the core microkernel goals of generality (ability to support the construction of arbitrary systems, especially highly-componentised systems) and minimality?

A partial answer has recently been given by NOVA [SK10], although that system is foremost a hypervisor for x86 platforms and requires hardware that is trap-and-emulate virtualizable [PG74]. NOVA also provides multiple abstractions for the same hardware resources (e.g. threads as well as vCPUs).

We provide another data point in the form of the *OKL4 microvisor*, a system explicitly designed to serve as a hypervisor as well as replacing our microkernel, and is aimed at performance-sensitive, memory-constrained embedded systems. The OKL4 microvisor is designed to be portable across a wide range of processor architectures (although the present product only supports ARM processors).

The OKL4 microvisor is a third-generation (3G) microkernel of L4 heritage (as indicated by the name). It grew out of our experience with large-scale commercial deployment of the OKL4 microkernel in mobile wireless devices and the growing demand for low-overhead platform virtualization in embedded systems.

The microvisor is also strongly influenced by our experience with the design, implementation and formal verification of seL4 [KEH⁺09]. It shares with seL4 the use of capabilities for access control on all resources, and a design for formal verification. It uses a more traditional approach to kernel resource management than seL4, but in other ways departs more aggressively than seL4 from the classical L4—by neither providing a synchronous message-passing IPC primitive nor kernel-scheduled threads.

3.2 The microvisor model

In line with the goal of supporting virtualization with the lowest possible overhead, the microvisor’s abstractions are designed to model hardware as closely as possible. Specifically,

- the microvisor’s execution abstraction is that of a virtual machine with one or more *virtual CPUs* (vCPUs), on which the guest OS can schedule activities;
- the memory abstraction is that of a *virtual MMU* (vMMU), which the guest OS uses to map virtual to (guest) physical memory;
- the I/O abstraction consists of memory-mapped *virtual device registers* and *virtual interrupts* (vIRQs);
- communication is abstracted as vIRQs (for synchronisation) and *channels*. The latter are bi-directional FIFOs with a fixed (configurable per channel) buffer allocated in user space (you can run TCP/IP on a channel if you *really* want to).

The asynchronous communication model not only maps better to hardware (including SMP) than the traditional L4 model. It also reflects our experience with mapping large, real-world embedded-systems code (such as multi-MLOC modem stacks and mobile-device application environments) to the L4 microkernel—the asynchronous model turned out to be a better match to the requirements of such systems.

L4’s synchronous IPC model necessitated the support for kernel-scheduled threads. With only an asynchronous IPC primitive, that need goes away, and the microvisor provides multiple vCPUs per VM solely for the purpose of allowing a VM to use multiple physical CPUs. Multiplexing of a single CPU between multiple activities within a VM is left to the OS. The model does not prevent creation of VMs with more vCPUs than physical cores, but there is no benefit in doing so.

The vMMU contains a virtual TLB which, like a real TLB, is a limited-size cache for mappings. The vTLB is typically much bigger than the real TLB (to make up for the higher access cost), and is implemented as a page table which is traversed by the microvisor on a page fault.

3.3 Implementation

The OKL4 microvisor is a clean, from-scratch design and implementation. It shares no code with the early commercially-deployed version of the L4 microkernel (but shares code modules with the presently shipping OKL4 microkernel). Like its predecessor, it is designed to support a mixture of real-time and non-real-time software running on top. The implementation comprises about 8.6 kLOC of ANSI C and about 1.2kLOC of assembler; it compiles to about 35 KiB of text. It is less complex (and smaller) than our earlier microkernels, which is one indication of an improved API.

In particular, the use of vIRQs as the communication primitive lead to dramatic simplifications compared to the synchronous IPC model traditionally used by L4 microkernels (even though that model had been significantly simplified over the years). As a consequence, it has no need for an “IPC fastpath”—there is really only a single code path in the vIRQ implementation, and it is much shorter than that of any synchronous IPC primitive.

The microvisor has a total of 30 hypercalls. This is more than the typical number of system calls of L4 microkernels (between seven and twenty, depending on L4 version). However, L4 system calls tend to be heavily overloaded (the OKL4 microkernel version 3.0 system header files contain over 200 APIs) while the microvisor hypercalls are all simple.

4. BUT DOES IT WORK?

4.1 Evaluation issues

A complete proof of our claim that the OKL4 microvisor represents the convergence point of microkernel and hypervisor technology would have to consist of two parts: firstly a demonstration that it is as good as any hypervisor in supporting virtual machines, and secondly a demonstration that it is as good as any microkernel in supporting microkernel applications.

The former is easier to do than the latter, as there is a lack of readily-accessible microkernel-based systems that could be used for comparison, and the lack of standardised APIs makes it difficult to build directly comparable systems. As the microvisor is a fairly new system, we have not yet had the opportunity to perform such an analysis.

Therefore, we can only look at the other part of the proof at present—comparing the performance with hypervisors. Even this

is not as easy as it may seem: OKL4 is designed for embedded systems and presently only available for ARM processors. While there exist a number of virtualization solutions which should be directly comparable to OKL4, these are proprietary, are not readily accessible even in binary form, and performance and even APIs are protected by NDAs. The most we can say here is that several prospective commercial users have performed competitive performance evaluations, from which the OKL4 microvisor has always emerged as the winner.

The only comparable system for which performance data is publicly available is a port of Xen to the ARM architecture which was performed by Samsung, and performance data was published for an ARM9 CPU [HSH⁺08].

Benchmark	Native	Virtualized	Overhead	
null syscall	0.6 μ s	0.96 μ s	0.36 μ s	60 %
read	1.14 μ s	1.31 μ s	0.17 μ s	15 %
write	0.98 μ s	1.22 μ s	0.24 μ s	24 %
stat	4.73 μ s	5.05 μ s	0.32 μ s	7 %
fstat	1.58 μ s	2.24 μ s	0.66 μ s	42 %
open/close	9.12 μ s	8.23 μ s	-0.89 μ s	-10 %
select(10)	2.62 μ s	2.98 μ s	0.36 μ s	14 %
select(100)	16.24 μ s	16.44 μ s	0.20 μ s	1 %
sig. install	1.77 μ s	2.05 μ s	0.28 μ s	16 %
sig. handler	6.81 μ s	5.83 μ s	-0.98 μ s	-14 %
prot. fault	1.27 μ s	2.15 μ s	0.88 μ s	67 %
pipe latency	41.56 μ s	54.45 μ s	12.89 μ s	31 %
UNIX socket	52.76 μ s	80.90 μ s	28.14 μ s	53 %
fork	1,106 μ s	1,190 μ s	84 μ s	8 %
fork+execve	4,710 μ s	4,933 μ s	223 μ s	5 %
system	7,583 μ s	7,796 μ s	213 μ s	3 %

Table 1: Lmbench results for OKL4 on Beagle Board

4.2 Virtualized Linux performance

There are presently no standardised hypervisor benchmarks for embedded systems. The most widely-used performance measure is comparing lmbench scores of virtualized and native Linux. Table 1 shows this comparison on the Beagle Board, a popular platform featuring a TI OMAP3530 processor based on an ARM Cortex-A8 core (ARM v7 architecture) clocked at 500MHz. The data for this table is taken from a customer’s evaluation report, and can therefore be considered independently validated (and agrees fully with our own measurements).

Most of the lmbench benchmarks time a single Linux system call, which in turn requires a single OKL4 hypercall to virtualize. These show that the basic overhead is around 0.3 μ s (150 cycles) per hypercall. The IPC (pipes and sockets) and process-creation (fork, exec, system) benchmarks are complex and require multiple hypercalls, and therefore show higher absolute overheads.

The relative overheads of the process-creation benchmarks (3–8 %) are remarkably low. For comparison, the OKL4 microkernel on an ARM9 processor (ARM v5 architecture) showed an overhead of 35 % for fork and 27 % for fork+execve [Hei09]. The corresponding figures for Xen are around 250 % [HSH⁺08], although it must be noted that these are from a research project and not a product, and the available performance data is by now 2.5 years old.

An anomaly are the negative apparent overheads for open/close and signal handling, and the high apparent overhead for protection fault. These result from the changed memory layout in the virtu-

alized system changing the patterns of cache conflicts misses, and indicate that not too much should be read into an individual result.

Type	Benchmark	Native	Virt.	O/H
TCP	Xput [Mib/s]	651	630	3 %
	Load [%]	99	99	0 %
	Cost [μ s/KiB]	12.5	12.9	3 %
UDP	Xput [Mib/s]	537	516	4 %
	Load [%]	99 %	99 %	0 %
	Cost [μ s/KiB]	15.2	15.8	4 %

Table 2: Netperf on Beagle Board

Table 2 shows the result of running netperf on a loopback device on both systems. Both systems showed a fully-loaded CPU, and the throughput degradation (and thus increase in cost per byte transmitted) of the virtualized system was three to four percent.

To the best of our knowledge, these are the lowest overhead figures ever published for a virtualized Linux system, even when comparing results published on other architectures (but we note that the ARM architecture features much lower trap costs than x86 and is therefore much more virtualization-friendly, so not too much can be read into comparisons across these architectures).

5. RELATED WORK

Closely related work is NOVA [SK10], a hypervisor designed for minimal size. NOVA is designed solely to support pure virtualization on hardware which is fully trap-and-emulate virtualizable, and for performance is strongly dependent on virtualization extensions to the x86 architecture.

NOVA minimises the kernel code (which the authors call the hypervisor) by moving much of the virtualization support, such as instruction emulation, to user level (they call this part the VMM). The VMM is replicated for each VM, thus removing it from the trusted computing base of “native” code (code executing without a guest OS on virtual bare hardware).

The NOVA hypervisor does not actually provide a minimal set of abstractions, but exhibits redundancy in its API. For example, NOVA offers threads as well as vCPUs, tasks as well as VMs, and synchronous IPC.

While the OKL4 microvisor is designed to make use of architectural support for virtualization, most cores for embedded use do not presently provide such support, and therefore require para-virtualization.

Comparing NOVA with our microvisor actually illustrates frequently-overlooked benefits of para-virtualization: the hypervisor can be much simpler. A para-virtualized guest on top of the microvisor does not require any instruction emulation, as all privileged instructions are replaced by explicit hypercalls (and typically each hypercall replaces many lines of guest source). This completely removes the need for the NOVA-like VMM, which adds 20 kLOC to the 9 kLOC of the NOVA kernel, compared to a total size of 10 kLOC for OKL4 (but of course, this comes at the expense of having to para-virtualize the guest).

Obviously, comparisons of code size between the two systems need careful interpretation, as the functionality is different (pure vs para-virtualization) and the ARM architecture is much cleaner than x86. If in the future ARM cores become fully virtualizable, utilising the respective hardware mechanisms will require extra code in the microvisor.

6. CONCLUSIONS

We hope to have convinced the reader that the microkernel-vs-hypervisor debate is pointless, that in reality the two concepts have significant overlap, and the two models can, should and will converge to this common subset. We have presented one such convergence point in the form of the OKL4 microvisor, and have demonstrated that it meets the hypervisor objective of minimal overhead for virtualization as well as the microkernel objective of minimal size. Proof of meeting the other core microkernel objective (generality) is in the works.

Acknowledgements

We would like to thank the OK Labs engineering team, especially Carl van Schaik, for their contributions to designing and implementing the OKL4 microvisor. 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.

7. REFERENCES

- [AG09] F. Armand and M. Gien. A practical look at micro-kernels and virtual machine monitors. In *6th IEEE Consumer Comm. & Networking Conf.*, Las Vegas, NV, USA, Jan 2009.
- [BDF⁺03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *19th SOSP*, pages 164–177, Bolton Landing, NY, USA, Oct 2003.
- [BH70] P. Brinch Hansen. The nucleus of a multiprogramming operating system. *CACM*, 13:238–250, 1970.
- [BSR09] A. Burtsev, K. Srinivasan, and P. Radhakrishnan. Fido: Fast inter-virtual-machine communication for enterprise applications. In *2009 USENIX*, San Diego, CA, USA, Jun 2009.
- [CB93] 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.
- [GHS] Green Hills Integrity. <http://www.ghs.com/products/rtos/integrity.html>.
- [Hei09] G. Heiser. Hypervisors for consumer electronics. In *6th IEEE Consumer Comm. & Networking Conf.*, pages 1–5, Las Vegas, NV, USA, Jan 2009.
- [HHL⁺97] 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.
- [HPHS04] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors. In *11th SIGOPS Eur. WS*, Leuven, Belgium, Sep 2004.
- [HSH⁺08] J.-Y. Hwang, S.-b. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *5th IEEE Consumer Comm. & Networking Conf.*, pages 257–261, Las Vegas, NV, USA, Jan 2008.
- [HUL06] G. Heiser, V. Uhlig, and J. LeVasseur. Are virtual-machine monitors microkernels done right? *ACM Operat. Syst. Rev.*, 40(1):95–99, Jan 2006.
- [HWF⁺05] S. Hand, A. Warfield, K. Fraser, E. Kottsovinos, and D. Magenheimer. Are virtual machine monitors microkernels done right? In *10th HotOS*, pages 1–6, Sante Fe, NM, USA, Jun 2005. USENIX.
- [KAR⁺06] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *1st EuroSys Conf.*, pages 133–145, Leuven, Belgium, Apr 2006.
- [KEH⁺09] 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.
- [LCC⁺75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in HYDRA. In *5th SOSP*, pages 132–140, 1975.
- [LCFD⁺05] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. R. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, Sep 2005.
- [LG09] P. Laroux and B. Graham. Secure by design: Using a microkernel rtos to build secure, fault-tolerant systems. White paper, QNX, <http://www.qnx.com/download/feature.html?programid=19358>, Apr 2009.
- [Lie93] J. Liedtke. Improving IPC by kernel design. In *14th SOSP*, pages 175–188, Asheville, NC, Dec 1993.
- [Lie95] J. Liedtke. On μ -kernel construction. In *15th SOSP*, pages 237–250, Copper Mountain, CO, Dec 1995.
- [LUSG04] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *6th OSDI*, pages 17–30, San Francisco, CA, USA, Dec 2004.
- [OKL4] Open Kernel Labs. OKL4 community site. <http://okl4.org>.
- [PG74] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *CACM*, 17(7):413–421, 1974.
- [Rut08] J. Rutkowska. Security challenges in virtualized environments. <http://www.invisiblethingslab.com>, Apr 2008.
- [SK10] U. Steinberg and B. Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *5th EuroSys Conf.*, Paris, France, Apr 2010.
- [WWG08] J. Wang, K.-L. Wright, and K. Gopalan. XenLoop: A transparent high performance inter-VM network loopback. In *17th HPDC*, pages 109–118, Boston, MA, USA, Jun 2008.
- [ZMRG07] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin. XenSocket: A high-throughput interdomain transport for virtual machines. In *8th Middleware*, pages 184–203, Newport Beach, CA, USA, Nov 2007.