Hypervisors for Consumer Electronics

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

***Abstract*—Virtualization, well established in enterprise computing, is finding its way into embedded systems. However, the use cases differ dramatically between the domains, and this results in significant differences in the requirements on the virtual-machine technology.**

This paper examines a number of typical virtualization use cases from the CE domain, and the resulting requirements imposed on the hypervisor. We find that enterprise-style hypervisors are ill-matched to the requirements of the embedded domain, which are characterised by low-overhead communication, real-time capability, small memory footprint, small trusted computing base, and fine-grained control over security. We present the OKL4 hypervisor, a member of the L4 microkernel family, designed for embedded-systems use. We outline OKL4's relevant properties with an emphasis on its security mechanisms, and compare its performance to a version of Xen that has recently been promoted for CE use. We conclude that OKL4 is superior to enterprise-style hypervisors for use in CE devices.

I. Introduction

Virtual machines have become mainstream in enterprise computing over the last few years. More recently, virtualization is gaining significant interest in the embedded domain, including deployments in mobile wireless communication devices, multimedia devices and network infrastructure, and proposals for use in automobiles [1].

In this paper we examine a number of use cases for virtualization in CE devices. Based on these, we establish the requirements on virtualization technology imposed by the nature of CE devices, and discuss how they imply a different approach to virtualization compared to the enterprise domain. We then translate this into requirements on the underlying virtual-machine (or hypervisor) technology and show that they are poorly matched by the kinds of hypervisors employed in the enterprise space.

We then present OKL4, a commercial, microkernel-based hypervisor targeted for use in embedded systems. We discuss how OKL4 matches the requirements of virtualization for CE which we identified earlier. We examine OKL4's security model, which is based on capabilities [2]. We also compare OKL4's performance and size to that of the ARM port of Xen [3], which has been advocated as a hypervisor for mobile wireless devices [4]. The results indicate that Xen fails to match the requirements of virtualization for CE. The same arguments mostly apply to other hypervisors designed for enterprise-style virtualization.

II. Virtualization Use Cases

The primary use case for virtual-machine technology in CE devices is that of *running several operating systems concurrently on the same processor*. This is motivated by the dramatic growth in the functionality of CE devices, which makes many of them similar to PCs (smart phones and mobile internet devices (MID) are the obvious examples). The rich functionality, together with the move to open devices, is creating a demand for familiar, open and high-level programming interfaces, similar to the ones provided by desktop operating systems. Consequently, this need is increasingly being served by stripped-down versions of desktop OSes, especially Windows and Linux.

The high-level OS is frequently not suitable for supporting all of the embedded software. Low-level real-time software (eg. for low-latency, low-overhead network processing) is traditionally supported by a lean real-time OS (RTOS). Continued support of the RTOS environment is important, both for retaining the value of the legacy software, and because the real-time capabilities of the high-level OS are in many cases not sufficient for the real-time requirements of the device.

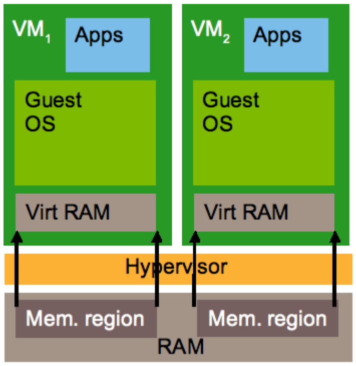

Fig. 1. Virtual machines

Obviously, such a dual-OS setup requires partitioning of the system's physical resources (processors, memory and devices) between the OS environments. A simple approach would be to use separate physical resources, but this is not always feasible. For performance reasons, the two environments typically need to share memory buffers, which rules out separate processors each with their own memory. Using a hypervisor to partition the resources avoids the need for physical separation, as in Figure 1.

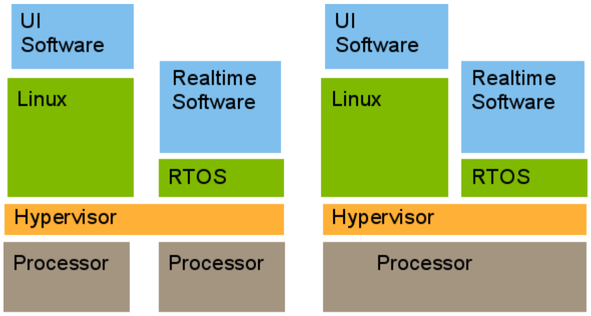

Fig. 2. Architectural abstraction provided by virtual machines

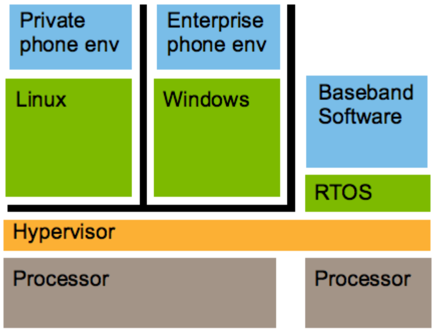

Fig. 3. Several concurrent UI environments in different virtual machines

A further benefit of such a setup is that it provides *architectural abstraction*: The system designer now has the freedom to map the two subsystems onto individual processor cores (with shared physical memory), or share a single core between them, depending on the processing needs of the particular product (see Figure 2). The resource abstraction provided by the hypervisor means that most software is unaffected by such a change. This can provide a significant cost advantage for a product series consisting of models of different capabilities but running mostly the same system software.

The same basic setup can also be used for a more *dynamic partitioning* of resources. For example, during periods of high processing demands of the UI software, the high-level OS (assuming it is multiprocessor-capable) can be given a share of the processor normally used for real-time processing, or, during periods of low overall demand, both systems can be migrated to the same core, shutting down the other core completely to save power.

For devices (such as mobile phones) which require certification [5], the isolation afforded by virtualization can be used to *reduce certification costs*. Product innovation tends to be predominantly in the user-facing functionality, which therefore tends to change much faster than the real-time subsystem (which tends to be the part requiring certification). By isolating the certified subsystem from any changes in the rest of the system, the certification remains valid.

This enables further design choices, including completely *opening up* the UI part of the devices, allowing users to completely change their OS environment (as done on the OpenMoko phone or a number of MIDs) without the hardware cost of complete physical separation. Some (trusted) versions of the UI stack could be given higher privileges, as long as the hypervisor is able to validate a certificate of trust.

A variant of this theme is the OSTI proposal [6], which proposes *two independent user environments* on a single mobile phone, one tightly controlled for business use (which integrates with the enterprise IT system) and an open one for private use (which holds private data, such as address books and bank account access codes, and allows the user to install software of their choice). Each environment runs its own operating system (they could be different, eg. Windows and Linux) and the environments can be switched by pressing a key. Given that the two OSes never run at the same time, reserving a separate physical processor for each is clearly a waste, and virtualization is the obvious approach, as shown in Figure 3.

This basic scenario is not restricted to mobile phones. A MID might also be used to *access an enterprise IT system*. Given the openness of the device (and the likelihood of it getting lost) the company might want to restrict access to an encapsulated trusted module. A hypervisor can provide the encapsulation while maintaining the overall openness.

Mobile devices are increasingly used for obtaining third-party services. This use creates increasing demand for fast and simple payment processes, eg. by using credit card information stored on the devices, which, in turn, results in concerns about fraud. Credit card companies require strong *isolation of pin-entry devices* (PED) [7], which tend to require separate hardware components. Virtualization can be used to avoid this expense.

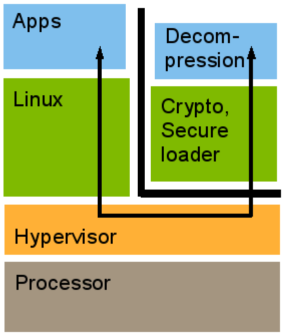

Fig. 4. Virtualization for IP protection

A related use case is that of *IP protection*, shown in Figure 4. A device (a concrete, real-life example that will ship this year is a HD IPTV set-top box) contains code that must be protected against theft (as it implements proprietary decompression algorithms). The device can easily be dismantled and hardware probes attached, so it must be protected against such attacks. The solution is to ship the code encrypted, with the decryption key kept in on-chip non-volatile memory. A secure boot process initialises the device to run a certified copy of the hypervisor in on-chip memory, and makes the decryption key available to the hypervisor. The hypervisor then decrypts and loads the proprietary code into a virtual machine which executes from on-chip memory, protected by the hypervisor from any other code running on the system, including the main OS (Linux) which operates the device. The main OS communicates with the protected decompression service via communication channels provided by the hypervisor. Similar setups can be used for protecting media contents (digital rights management).

Firmware over the air (FOTA) upgrades represent another interesting use case. While there exist widely-deployed approaches for FOTA that do not depend on virtualization, the level of indirection provided by the hypervisor can significantly simplify the upgrade and reduce the amount of infrastructure required to support it. Also, it may be possible to avoid a reboot of the complete systems after the upgrade, limiting the restart to the upgraded component.

III. Hypervisor Requirements

What are the requirements posed on the hypervisor from the above use cases?

The basic use case of co-existing high-level and real-time OSes inherently requires that the hypervisor is *real-time capable*. This means short and bounded interrupt latencies, a requirement that is quite different from expectations put on a hypervisor in the enterprise space.

The *isolation* provided by virtual machines is critically important for most of the use cases. However, this raises an interesting issue, as embedded systems are by their nature tightly integrated — all subsystems cooperate closely to meet the system's overall mission.

Tight integration requires highly-efficient (low-latency, high-bandwidth) *communication channels* (IPC), very much unlike enterprise-style virtualization use cases. The established cross-VM communication channels (essentially virtual network interfaces) are too expensive for embedded-systems use — the hypervisor needs to provide secure access to shared-memory regions and low-overhead message-passing mechanisms. This also implies a need for *fast inter-VM switches*, and, as argued elsewhere [8], a need for a global approach to scheduling (where priorities of tasks from different VMs are interleaved) as opposed to the hierarchical scheduling approach that is inherent in the virtual-machine model.

The IP-protection use case requires a hypervisor that is small enough to share on-chip memory with the security-critical application. No security-critical code must ever leave the chip (except when signed). This puts extremely stringent condition on the memory footprint of the hypervisor. Even in cases where this particular requirement does not apply, the hypervisor of a secure system needs to be kept small in order to minimise the size of the *trusted computing base* (TCB), in accordance with established security principles [9].

The FOTA use case has similar requirements. The benefits of indirection are voided if provision must be made for upgrading the hypervisor once deployed on the device. This means that the FOTA use case will only work if the hypervisor can be relied on to operate correctly for the lifetime of the device without requiring an upgrade. This implies a very high degree of *freedom from faults*, which is only achievable with a very small code base.

IV. Enterprise vs. Embedded Hypervisors

A case has been made that an enterprise-style hypervisor, specifically Xen [3], can be adapted for embedded-systems use [4]. However the above observations are in conflict with such an approach.

Xen, like other enterprise-style hypervisors, has been built without any regard for real-time performance. Real-time capability is very difficult to retrofit into an operating-system code base, because it affects all code. Either the kernel has to be made (almost completely) preemptible, or all kernel operations have to be short and bounded. It took Linux years (in spite of heavy investment by large corporations) to achieve a reasonable degree of real-time performance, and it still is not considered suitable for serious real-time work, else we would not bother with the Linux-RTOS use case. Hence, making Xen (or any other enterprise hypervisor) real-time capable should present a formidable challenge.

Similarly, the communication requirement: Xen was not designed to support fast IPC [10], and the history of microkernels shows that this is also very difficult to retrofit. Liedtke concluded that a kernel needs to be designed from the beginning for high IPC performance [11].

Then there is size. Xen has, by embedded-systems standards, a very large memory footprint. It contains a complete Python interpreter, and uses a dedicated Linux guest just as a driver container and for management functions. This is in addition to any guest OS required for the actual operation of the embedded device. For the typical setup of Linux+RTOS, we can roughly estimate that Xen alone (including the Dom0 OS) requires about as much memory as the two guest OSes combined. Given that memory is a significant cost factor in many embedded devices, and the power consumption of RAM a significant factor limiting battery lifetime, this does not seem a feasible approach. And it obviously rules out running all security-critical code in on-chip memory, as required in the IP-protection use case.

A size argument can also be made for the TCB. The critical measure here is lines of code, as defect numbers grow at least linearly with code size [12]. The complete OKL4 embedded hypervisor is less than 12 kLOC (lines of code). Hwang et al. report that the port of Xen to ARM required 23.4 kLOC of new or changed code [4], which is about twice the size of the complete OKL4 [13] hypervisor!

V. OKL4

OKL4 is a virtualization product from Open Kernel Labs. The OKL4 hypervisor is a member of the L4 microkernel family which originated with Liedtke's original L4 [14]. It has been successfully deployed in CE devices, including an estimated 250 million mobile phones.

A. Overview

OKL4 has been designed to meet the requirements of virtualization for embedded systems. Specifically, it features high-performance IPC of less than 200 cycles for a one-way message-passing operation on an ARM9 processor and provides efficient mechanisms for setting up shared memory regions. It has low interrupt latencies, able to deliver an interrupt to a driver running in a virtual machine within a few microseconds. Its memory footprint is less than 64KiB, and the complete code size for ARM is less than 12 kLOC.

OKL4 supports para-virtualized high-level OSes as well as RTOSes. OK Linux, the para-virtualized version of Linux on OKL4 is in fact a port to an "OKL4 architecture", meaning that a new architecture subtree is introduced into the Linux source tree. This can simply be dropped into a kernel.org source, otherwise only needing a small patch affecting about 20 lines of architecture-independent Linux kernel code (some of them bug fixes).

OKL4 is more than just a hypervisor, it is in fact a small yet general-purpose OS platform. It provides an optional light-weight Posix environment, which supports running individual programs "native", without the need of a virtualized guest OS. This establishes an ideal minimal-TCB environment for executing safety- or security-critical code. Examples are the PED and IP-protection use cases. However, the native environment is also useful for other purposes. For example, running browsers, media players, TPC/IP stacks or video/audio codecs in a native environment my be advantageous for configuration management, as well as for making those components independent of a particular guest OS.

B. Security

Recently, mandatory access control (MAC) mechanisms have been implemented in versions of Xen [15], [16]. OKL4 also supports mandatory access control via an abstraction called *Secure HyperCell*$^{\text{TM}}$ (SHC). SHCs are isolation domains of varying granularity, from individual processes (eg. a device driver or a codec module) up to complete virtual machines.

SHCs are based on an underlying fine-grained access-control model using *capabilities* [2]. OKL4 capabilities are kernel objects which control all other kernel objects, namely virtual and physical memory (including device registers), address spaces, threads, and IPC.[1] Capabilities can thus be used to delegate authority over resources to subsystems — a SHC is a subsystem associated with a particular set of capabilities. CPU time is controlled via scheduler threads using capability-mediated mechanisms to allocate time slices to threads.

Capabilities are therefore a mechanism for enforcing mandatory access and communication control between SHCs. As indicated in Figure 5, communication is allowed between cells if they have a capability for an information channel (IPC endpoint or shared memory), and prevented otherwise. This allows implementing virtually arbitrary security policies, including those required by the security use cases discussed earlier.

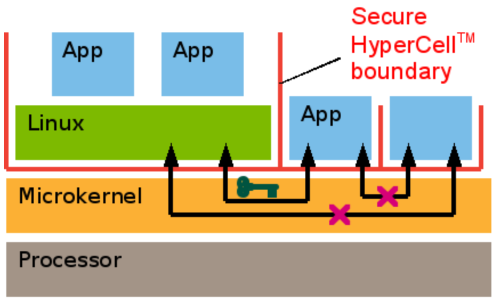

Fig. 5. Capabilities used for communication control

The fine-grained protection management provided by capabilities is particularly powerful in combination with the minimal native execution environment mentioned above. It allows running a security- or safety-critical component with a minimal TCB, consisting only of the microkernel and whatever Posix functionality the component needs. The TCB on the deployed device may be as small as 15 kLOC, and the SHC framework ensures that only code that has been given a capability can interact in any way with the protected component.

Capability-based protection combined with small code size is an enabler for a truly exciting next step towards real trustworthiness [17]: mathematical proof that the security properties claimed for the system hold for the actual implementation (i.e. the C and assembler code of the kernel). Such a proof is in progress for a close relative of OKL4 and is expected to complete around the time of publication of this paper [18]. The techniques will then be applied to OKL4 itself.

C. Performance

Table I shows performance measurements using lmbench latency benchmarks. OKL4 benchmarks were performed with the OKL4 3.0 release of October 2008 and OK Linux based on Linux 2.6.24. They were run on an XScale PXA255 at 200 MHz. This platform is somewhat slower than the 266 MHz ARM926 processor used in the Xen work [4], and has a different memory architecture. Hence, care must be taken in comparing results from the two platforms, and not too much can be read into smaller differences.

The "native" and "virt./OKL4" columns of the table compare native with OKL4-virtualized Linux performance on the XScale (small is good). The "rel. perf." columns show the performance degradation experienced by OKL4- and Xen-based virtualization (large is good).

In the cases where virtualized Linux outperforms native (rel. perf > 1), the former profits from the fast context-switching implementation provided by OKL4, which avoids flushing caches (ARMv5 cores use virtually-addressed caches). This optimisation (which could also be done in native Linux) is an

[1] The present 3.0 release of OKL4 does yet subject all resources to capability-mediated access control, some are still managed by other mechanisms. This will change in the near future.

TABLE I
BENCHMARK PERFORMANCE OF OKL4 AND XEN.

	Latencies [μs]		Rel. Perf.	
Benchmark	Native	Virt./OKL4	OKL4	Xen
pipe	756.59	84.84	8.92	0.58
fork	6469	8742	0.74	0.29
fork+exec	59715	75515	0.79	0.30
semaphore	261.6	21.08	12.41	0.56
unix	1292.2	115.01	11.24	0.59
signal handler	14.26	54.76	0.26	0.55
null syscall	1.14	5.40	0.21	0.40
read syscall	3.34	8.45	0.40	0.52

indication that it is easier to optimise a 10 kLOC than a 200 kLOC kernel.

The cases where the performance degradation under Xen is less than that under OKL4 are mostly a result of the far less invasive para-virtualization approach used in OK Linux. Those figures could be improved, but have to date not become bottlenecks.

Another interesting comparison is the cost of security. The Xen work reports the cost of security decisions (which must be done at each domain crossing) to be on average 7.69 μs [16]. In contrast, the security check in OKL4 is done implicitly when the capability passed to a system call is validated. The extra cost of this is of the order of 10–20 cycles (50–100 ns). The total cost of a cross-address-space IPC is around 1–1.5 μs in OKL4, including the security check.

VI. CONCLUSION

We have argued that virtualization for embedded systems has requirements which enterprise-style virtual machines do not satisfy, and which are structurally hard to retrofit. Specifically these include efficient IPC, real-time capability, small memory footprint and small TCB.

We based our assessment on Xen, not only because it is a typical and widely-used representative of an enterprise-style hypervisor, but also because it is the only such hypervisor that has been ported to ARM, the leading processor architecture on mobile wireless devices and other kinds of CE devices. Moreover, Xen has been specifically advocated for CE use [4].

From the above discussion it should be evident that the unsuitability of Xen is not a result of Xen-specific features. To the contrary, Xen is fairly representative of enterprise-style hypervisors, as it shares many relevant properties with them. Most properties which make Xen unsuitable for CE use are shared with other hypervisors in its class, such as KVM or VMware. Specifically these systems all have a large TCB, large memory requirements(by embedded standards), and are not designed to support real-time systems and low-latency cross-VM communication. They also adhere to the traditional black-box model that implies hierarchical scheduling and power management, both of which are unsuitable for embedded systems [8].

We presented OKL4, a microkernel-based hypervisor specifically designed to meet the requirements of virtualization for CE. While a number of other commercial hypervisors are marketed for embedded use, claiming many of the relevant characteristics, this is in fact hard to verify due to their closed nature. OKL4 is *open source*, and thus easy to access and evaluate. However, OKL4 goes much further than any competitor offerings in its security focus: Its unique *Secure HyperCell*$^{\text{TM}}$ technology, based on kernel-protected capabilities, provides a very low-overhead mechanism for mandatory information-flow control, and thus provides the basis for a security-oriented systems design. And the kernel is small enough to be suitable for complete formal verification, for ultimate trustworthiness.

For all those reasons we believe that OKL4 presents a superior approach to addressing the virtualization needs of next-generation CE devices.

ACKNOWLEDGMENT

The author would like to thank James Arth and Jean-Pierre Seifert for helpful feedback, and Jonathan Sokolowski for benchmarking the pre-release system.

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.

REFERENCES

[1] A. Hergenhan and G. Heiser, “Operating systems technology for converged ECUs,” in *6th Emb. Security in Cars Conf. (escar)*. Hamburg, Germany: ISITS, Nov 2008.

[2] J. B. Dennis and E. C. Van Horn, “Programming semantics for multiprogrammed computations,” *CACM*, vol. 9, pp. 143–155, 1966.

[3] 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*, Bolton Landing, NY, USA, Oct 2003, pp. 164–177.

[4] 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.*, Las Vegas, NV, USA, Jan 2008, pp. 257–261.

[5] FCC, “Ruling 07-66,” http://edocket.access.gpo.gov/2007/07-2684.htm, Apr 2007.

[6] NTT DoCoMo and Intel Corp, “Open and secure terminal initiative (OSTI) architecture specification,” http://www.nttdocomo.co.jp/english/corporate/technology/osti/, Oct 2006.

[7] “PCI security standards,” http://www.pcisecuritystandards.org.

[8] G. Heiser, “The role of virtualization in embedded systems,” in *1st IIES*. Glasgow, UK: ACM SIGOPS, Apr 2008, pp. 11–16.

[9] *Trusted Computer System Evaluation Criteria*, US Department of Defence, 1986, doD 5200.28-STD.

[10] S. Hand, A. Warfield, K. Fraser, E. Kottsovinos, and D. Magenheimer, “Are virtual machine monitors microkernels done right?” in *10th HotOS*. Sante Fe, NM, USA: USENIX, Jun 2005, pp. 1–6.

[11] J. Liedtke, “Improving IPC by kernel design,” in *14th SOSP*, Asheville, NC, USA, Dec 1993, pp. 175–188.

[12] L. Hatton, “Re-examining the fault density - component size connection,” *Softw.*, vol. 14, no. 2, pp. 89–97, 1997.

[13] Open Kernel Labs, “OKL4 community site,” http://okl4.org.

[14] J. Liedtke, “On μ-kernel construction,” in *15th SOSP*, Copper Mountain, CO, USA, Dec 1995, pp. 237–250.

[15] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn, “Building a MAC-based security architecture for the Xen open-source hypervisor,” in *21st ACSAC*, 2005.

[16] S.-M. Lee, S. bum Suh, B. Jeong, and S. Mo, “A multi-layer mandatory access control mechanism for mobile devices based on virtualization,” in *5th IEEE Consumer Comm. & Networking Conf.*, Las Vegas, NV, USA, Jan 2008, pp. 251–256.

[17] G. Heiser, “Trustworthy ⇐ trusted ⇐ proof,” in *1st Conf. Future Trust Comput.* Berlin, Germany: Vieweg-Verlag, Jul 2008.

[18] NICTA, “L4.verified,” http://ertos.nicta.com.au/research/l4.verified/.