Open Kernel Labs™
Be open. Be safe.

TECHNOLOGY WHITE PAPER

Virtualization for Embedded Systems

Gernot Heiser, PhD
Chief Technology Officer
Open Kernel Labs, Inc.

Document Number: OK 40036:2007
Date: November 27, 2007

Table of Contents

1 Introduction 5

2 Virtualization 6

2.1 What Is It? 6

2.2 How Is It Done? 7

2.2.1 Pure virtualization 8

2.2.2 Impure virtualization 9

3 Virtualization for Embedded Systems 10

3.1 Virtualization Benefits for Embedded Systems 10

3.1.1 Modern embedded systems software 10

3.1.2 Multiple concurrent operating systems 10

3.1.3 Security 11

3.1.4 Multicore chips 12

3.1.5 License separation 12

3.2 When “Virtualization” is not Virtualization 13

3.2.1 Security 13

3.2.2 License separation 13

3.3 Limits of Virtualization 14

3.3.1 Software complexity 14

3.3.2 Integration 14

3.3.3 Security policies 15

3.3.4 Trusted computing base 16

4 Microkernels — A Better Solution 17

4.1 Embedded Systems Requirements 17

4.2 Microkernels 17

4.2.1 What are microkernels? 17

4.2.2 General properties of microkernel systems 18

4.3 OKL4 Microkernel Technology 19

4.3.1 Low-overhead virtualization 19

4.3.2 Unbeaten IPC performance 20

4.3.3 Efficient resource sharing 20

4.3.4 Flexible scheduling 21

4.3.5 Security 21

4.3.6 Small trusted computing base 22

4.3.7 Open-source software 23

4.4 Virtualization with OKL4 — Best of Both Worlds 23

5 The Future: Many Cores, Many Components, Many Nines 25

5.1 The Challenges 25

5.2 Future-Proofing Embedded Technology 25

Bibliography 27

About the Author 28

About Open Kernel Labs 28

1 Introduction

Virtualization has been a hot topic in the enterprise space for quite some time, but has recently become an important technology for embedded systems as well. It is therefore important for embedded-systems developers to understand the power and limitations of virtualization in this space, in order to understand what technology is suitable for their products.

This white paper presents an introduction to virtualization technology in general, and specifically discusses its application to embedded systems.

We explain the inherent differences between the enterprise-systems style of virtualization and virtualization as it applies to embedded systems. We explain the benefits of virtualization, especially with regard to supporting embedded systems composed of subsystems with widely varying properties and requirements, and with regard to security and IP protection.

We then discuss the limitations of plain virtualization approaches, specifically as it applies to embedded systems. These relate to the highly-integrated nature of embedded systems, and the particular security and reliability requirements.

We present microkernels as a specific approach to virtualization, and explain why this approach is particularly suitable for embedded systems. We show how microkernels, especially Open Kernel's OKL4 technology, overcome the limitations of plain virtualization. We then provide a glimpse at the future of this technology.

2 Virtualization

2.1 What Is It?

A virtual machine provides a software environment which allows software to run as on bare hardware. This environment is created by a *virtual-machine monitor* or *hypervisor*.

Virtualization refers to providing a software environment on which programs, including operating systems, can run as if on bare hardware (Figure 2.1). Such a software environment is called a *virtual machine* (VM). Such a VM is an *efficient, isolated duplicate of the real machine* [PG74].

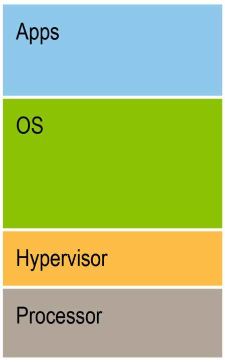

Figure 2.1.

A virtual machine. The hypervisor (or virtual-machine monitor) presents an interface that looks like hardware to the "guest" operating system.

The software layer that provides the VM environment is called the *virtual-machine monitor* (VMM), or *hypervisor*.

In order to maintain the illusion that is incorporated in a virtual machine, the VMM has three essential characteristics [PG74]:

1. the VMM provides to software an environment that is essentially identical with the original machine;
2. programs run in this environment show, at worst, minor decreases in speed;
3. the VMM is in complete control of system resources.

All three characteristics are important, and contribute to making virtualization highly useful in practice. The first (similarity) ensures that software that runs on the real machine will run on the virtual machine and vice versa. The second (efficiency) ensures that virtualization is practicable from the performance point of view. The third (resource control) ensures that software cannot break out of the VM.

The term virtual machine is also frequently applied to language environments, such as the Java virtual machine. This is referred to as a *process VM*, while a VM that corresponds to actual hardware, and can execute complete operating systems, is called a *system VM* [SN05]. In this paper we only deal with system VMs.

2.2 How Is It Done?

Most instructions of a virtual machine are executed directly on hardware. Instructions which access physical resources are interpreted by the virtual-machine monitor.

The efficiency feature requires that the vast majority of instructions be directly executed by the hardware: any form of emulation or interpretation replaces a single virtual-machine instruction by several instructions of the underlying *host* hardware. This requires that the virtual hardware is mostly identical to the physical hardware on which the VMM is hosted.

Small differences between the virtual and physical machines are possible. For example, the virtual machine may have some extra instructions not supported by the physical hardware. The physical hardware may have a different memory-management unit or different devices than the virtual hardware. The virtual machine may be an old version of the same basic architecture, and be used to run legacy code. Or the virtual machine may be a not yet implemented new version of the architecture. As long as the differences are small, and the differing instructions not heavily used, the virtualization can be about as efficient as if the hardware was the same.

Not all instructions can be directly executed. The resource-control characteristic requires that all instructions that deal with resources must access the virtual rather than the physical resources. This means such instructions must be *interpreted* by the VMM, as otherwise virtualization is broken.

Specifically, there are two classes of instructions that must be interpreted by the virtual machine:

control-sensitive instructions which modify privileged machine state, and therefore interfere with the hypervisor's control over resources;

behaviour-sensitive instructions which access (read) privileged machine state. While they cannot change resource allocations, they reveal the state of real resources, specifically that it differs from the virtual resources, and therefore breaks the illusion provided by virtualization.

Together, control-sensitive and behaviour-sensitive instructions are called *virtualization-sensitive*, or simply *sensitive* instructions.

There are two basic ways to ensure that code running in the virtual machine does not execute any sensitive instructions:

pure virtualization: ensure that sensitive instructions are not executable within the virtual machine, but instead invoke the hypervisor;

impure virtualization: remove sensitive instructions from the virtual machine and replace them with virtualization code.

2.2.1 Pure virtualization

Pure virtualization is the classical approach. It requires that all sensitive instructions are *privileged*. Privileged instructions execute successfully if the processor is in a privileged state (typically called *privileged mode*, *kernel mode* or *supervisor mode*) but generate an *exception* when executed in *unprivileged mode* (also called *user mode*), as shown in Figure 2.2. An exception enters privileged mode at a specific address (the exception handler) which is part of the hypervisor.

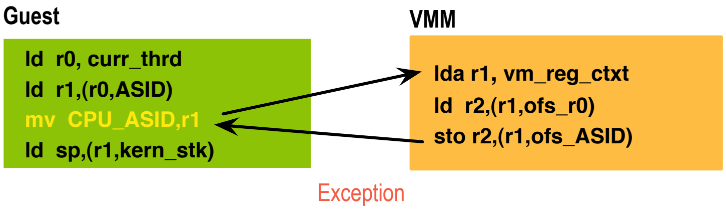

Figure 2.2.

Most instructions of the virtual machine are directly executed, while some cause an exception, which invokes the hypervisor which then interprets the instruction.

Pure virtualization then only requires executing all of the VM's code in non-privileged execution mode of the processor. Any sensitive instructions contained in the code running in the VM will trap into the hypervisor. The hypervisor interprets ("virtualizes") the instruction as required to maintain virtual machine state.

Until recently, pure virtualization was impossible on almost all contemporary architectures, as they all featured sensitive instructions that were not privileged (and thus would access physical rather than virtual machine state). Recently all major processor manufacturers have added *virtualization extensions* that allow the processor to be configured in a way that forces all sensitive instructions to cause exceptions.

However, there are other reasons why alternatives to pure virtualization are widely used. One is that exceptions are expensive. On pipelined processors, an exception drains the pipeline, resulting in delay in processing, typically one cycle per pipeline stage. A similar delay typically happens when returning to user mode. Furthermore, exceptions (and exception returns) are branches that are usually not predictable by a processor's branch-prediction unit, resulting in additional latency. These effects typically add up to some 10–20 cycles, more in deeply-pipelined high-performance processors. Some processors (notably the x86 family) have exception costs that are much higher than this (many hundreds of cycles).

2.2.2 Impure virtualization

Impure virtualization requires the removal of non-privileged sensitive instructions from the code executing in the virtual machine, as shown in Figure 2.3. This can happen transparently, by a technique called *binary code rewriting*: the executable code is scanned at load time, and any problematic instructions are replaced by instructions that cause an exception (or provide virtualization by other means, such as maintaining virtual hardware resources in user mode).

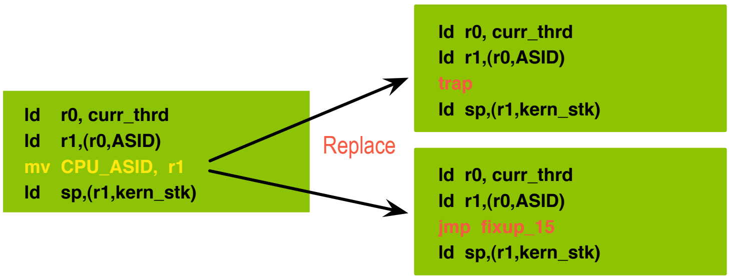

Figure 2.3.

Impure virtualization techniques replace instructions in the original code by either an explicit hypervisor call (trapping instruction) or a jump to user-level emulation code.

An alternative is to prevent problematic instructions from appearing in the executable code in the first place. This can be done at compile time by a mostly-automatic technique called *pre-virtualization* (also referred to as *afterburning*) [LUC$^+$05]. Alternatively, the source code can be manually modified to remove direct access to privileged state and instead replace such accesses by explicit invocations of the hypervisor (*"hypercalls"*). This approach is referred to as *para-virtualization*.

Para-virtualization replaces instructions of the original code by explicit VMM invocations. This not only has the advantage that it works on hardware that is unsuitable for pure virtualization, it also can have significant performance advantages.

The operation remains the same as in pure virtualization: The guest code runs in non-privileged execution mode of the processor, and a virtualization event is handled by invoking the hypervisor.

Para-virtualization and pre-virtualization have another advantage, besides being able to deal with hardware that is not suitable for pure virtualization: They can replace sequences of many sensitive instructions by a single hypercall, thus reducing the number of (expensive) switches between unprivileged and privileged mode. As such, impure virtualization has the potential to reduce the virtualization overhead, which makes it attractive even on fully virtualisable hardware.

3 Virtualization for Embedded Systems

Virtualization on its own is the wrong paradigm for embedded systems.

Virtualization, which originated on mainframes and finds increasing use on personal computers, has recently become popular in the embedded-systems space. In this chapter we will examine, not only the benefits virtualization brings to this application domain, but also the limitations which, in the end, imply that virtualization on its own is the wrong paradigm for embedded systems.

3.1 Virtualization Benefits for Embedded Systems

3.1.1 Modern embedded systems software

In order to understand the attraction of virtualization in the embedded-systems context, it is useful to recall the relevant features of modern embedded systems.

In the past, embedded systems were characterised by simple functionality, a single purpose, no or very simple user interface, and no or very simple communication channels. They also were *closed* in the sense that all the software on them was loaded pre-sale by the manufacturer, and normally remained unchanged for the lifetime of the device. The amount of software was small.

Modern embedded systems feature a wealth of functionality, open platforms, and code sizes measured in the millions of lines.

Many modern embedded systems, however, are very different — the mobile phone handset is a good representative. Such a system has a sophisticated user interface, consisting of input keys, possibly a touch screen, camera, audio and high-resolution video output. It combines many functions, including voice and data communication, productivity tools, media players and games. It supports different wireless communication modes, including multiple cellular standards, Bluetooth and infrared. It allows the user to load data and even programs. The total software running on the device is complex and large, measuring millions of lines of code.

3.1.2 Multiple concurrent operating systems

The key attraction of virtualization for embedded systems it that it supports the concurrent existence and operation of multiple operating systems on the same hardware platform.

Virtualization supports the concurrent use of several different operating systems on the same device. Typically this is used to run a RTOS for low-level real-time functionality (such as the communication stack) while at the same time running a high-level OS, like Linux or Windows, to support application code, such as user interfaces.

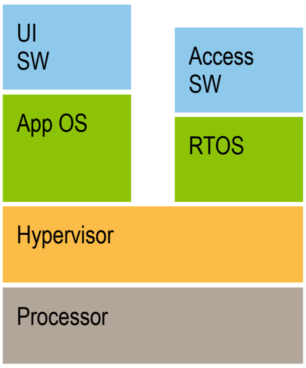

Figure 3.1.

Virtualization allows running multiple operating systems concurrently, serving the different needs of various subsystems, such as real-time environment vs. high-level API.

This is driven by the vastly different requirements of the various subsystems that provide separate aspects of the device’s functionality. On the one hand, there is real-time functionality that requires low and predictable interrupt latency. In the case of the mobile phone terminal, the cellular communication subsystem has such real-time requirements.

These requirements are traditionally met by a small and highly efficient *real-time operating system* (RTOS).

On the other hand, there is a large (and growing) amount of high-level application code that is similar (and often identical) to typical application code used on personal computers. Such code is typically developed by application programmers, who are not experts in low-level embedded programming.

Such application code is best served by a *high-level operating system* (also called *rich OS*, *application OS* or *feature OS*) that provides a convenient, high-level application interface. Popular examples are Linux and embedded versions of Windows.

Virtualization serves those different requirements by running appropriate operating systems concurrently on the same processor core, as shown in Figure 3.1. The same effect can be achieved by using separate cores for the real-time and application software stacks. But even in this case, virtualization provides advantages, which will be discussed in Section 3.1.4.

The ability to run several concurrent operating systems on a single processor core can reduce the bill of materials, especially for lower-end devices. It also provides a uniform OS environment in the case of a product series (comprising high-end devices using multiple cores as well as lower-end single-core devices).

An interesting aside relates to the concept of virtualizability in the embedded space: It is typically not particularly relevant to hide from a guest OS the fact that it is running in a virtual machine. Hence, in the embedded context it may be less of an issue if some behaviour-sensitive instructions are not privileged.

3.1.3 Security

Virtualization can be used to enhance security. A virtual machine encapsulates a subsystem, so that its failure cannot interfere with other subsystems. In a mobile phone handset, for example, the communication stack is of critical importance—if it were subverted by an attacker, the phone may interfere with the network by violating communication protocols. In the extreme case, the phone could be turned into a jammer which disables communication in the whole cell. Similarly, an encryption subsystem needs to be strongly shielded from compromise to prevent leaking the information the encryption is supposed to protect.

This is a significant challenge for a system running millions of lines of code, which inevitably contain tens of thousands of bugs, many of them security-critical. Especially in an open system, which allows owners to download and run arbitrary programs, the high-level OS is

Virtualization protects critical subsystems, such as the communications stack, from a compromised application OS. This is relevant even if the application OS runs on its own processor core.

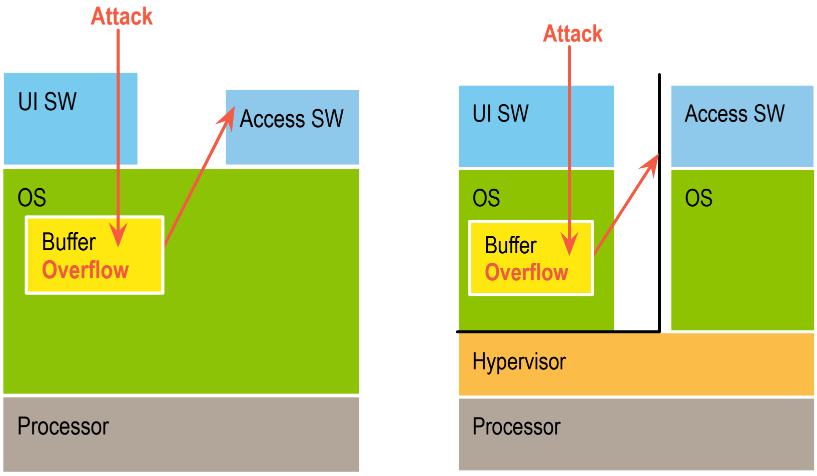

Figure 3.2.

Once the operating system is compromised (e.g. by an application program which exploits a buffer or stack overflow in the kernel), any software running on top can be subverted, as shown on the left. Encapsulating a subsystem into a VM protects other subsystems from a compromised OS.

subject to attacks, and is large enough (hundreds of thousands of lines of code) to contain of the order of a thousand bugs. In the absence of virtualization, the high-level OS runs in privileged mode, and therefore, once compromised, can attack any part of the system.

With virtualization, the high-level OS is de-privileged and unable to interfere with data belonging to other subsystems, as shown in Figure 3.2, and its access to the processor can be limited to ensure that real-time components meet their deadlines.

3.1.4 Multicore chips

The above threat scenario is not eliminated by running the application OS on a separate processor core. Unless the cores also have separated memory (which complicates system design and makes data transfer between cores expensive), a compromised application OS running in privileged mode can still access other subsystems' data, including kernel data structures.

This can be prevented by virtualization: the hypervisor partitions physical memory between virtual machines, and thereby prevents such interference.

3.1.5 License separation

Linux is a frequently deployed high-level OS. Its advantages are the royalty-free status, independence from specific vendors, widespread deployment and a strong and vibrant developer community and large ecosystem.

Linux is distributed under the GPL license, which requires that all derived code is subject to the same license, and thus becomes open source. There are legal arguments [Was07] that this even applies to device drivers that are loaded as binaries at run time into the kernel.

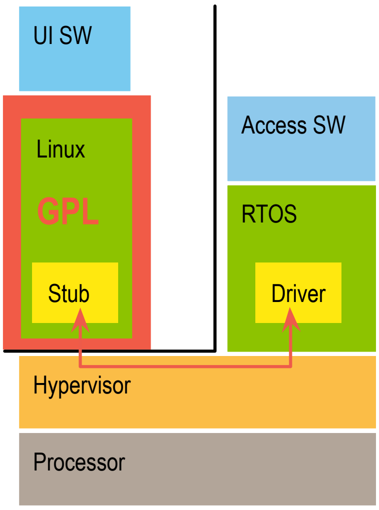

Figure 3.3.

Virtualization is frequently employed to segregate components subject to GPL from proprietary code.

Linux is licensed under the GPL which requires open-sourcing of all derived code. Virtualization is frequently employed to provide a proprietary software environment segregated from the GPL environment.

Virtualization is frequently employed to provide a proprietary execution environment for software that is to share the processor with a Linux environment. Linux and the proprietary environment are run in separate virtual machines. A stub (or proxy) driver is used to forward Linux driver requests to the real device driver, using hypercalls (see Figure 3.3).

3.2 When “Virtualization” is not Virtualization

In Section 2.1 we described the basic characteristics of virtualization. While these have been clearly understood for decades, the popularity of virtualization in recent years has lead some to apply the term to technologies that in reality are *not* virtualization in the established sense.

Does this matter? It does, as some of the benefits of virtualization are lost in technology that is not really virtualization. Let’s have a closer look...

Item 3 of the list of *essential characteristics of virtualization* given in Section 2.1 states that the *VMM is in full control of system resources*. In Section 2.2 we saw that this is achieved by running guest code in non-privileged mode, while the hypervisor runs privileged and has control over resources.

Pseudo-virtualization runs guest OSes at the highest privilege level, and thus forfeits some of the core benefits of virtualization, including security and possibly license separation.

This is exactly the point where some technologies claiming to provide virtualization actually fail to do so. Such technologies, which are called *pseudo-virtualization*, run guest operating systems in kernel mode, together with the hypervisor (the operating-systems literature calls this *co-locating* the guest with the hypervisor). In doing so, they forfeit some of the core benefits of virtualization.

3.2.1 Security

The security benefits discussed in Section 3.1.3 critically depend on the guest OS running de-privileged. If the guest runs in kernel mode, the hypervisor’s data structures are not protected from a misbehaving guest, and the guest can take complete control of the machine, including on multi-cores. From the security point of view, pseudo-virtualization offers absolutely nothing.

3.2.2 License separation

The license separation discussed in Section 3.1.5 depends on a clear separation of the GPL-ed code (the Linux kernel code) from the rest of the system, via non-GPLed interfaces. Does this separation still hold when the Linux kernel code runs in kernel mode, co-located with the hypervisor and other guest OSes?

This is a tricky legal question, on which lawyers are likely to disagree, as on many issues around the GPL.

The Free Software Foundation, guardians of the GPL, maintains a web page of answers to frequently-asked questions [Fre07]. The closest match in there seems to be this:

> **Question:** You have a GPL-ed program that I’d like to link with my code to build a proprietary program. Does the fact that I link with your program mean I have to GPL my program?
>
> **Answer:** Yes

Is a pseudo-virtualized Linux kernel co-located with the hypervisor and other guests a “GPL-ed program linked with a proprietary program” in the above sense? The argument can certainly be made. Will it prevail in court? No-one can say for sure at this stage. *Caveat emptor!*

3.3 Limits of Virtualization

While virtualization offers a number of compelling advantages, it is important to understand its limitations. These are particularly relevant to embedded systems. A closer examination will show that virtualization, on its own, is not sufficient to address the challenges of modern embedded systems. The main issues are granularity and integration.

In order to fully appreciate those issues, we will revisit some of the challenges facing modern embedded systems.

3.3.1 Software complexity

The complexity of modern embedded software poses formidable challenges to system reliability.

Modern embedded systems feature a wealth of functionality and, as a result, are highly complex. This is particularly true for their software, which frequently measures in the millions of lines of code and is growing strongly.

Systems of that complexity are, for the foreseeable future, impossible to get correct—in fact, they can be expected to contain tens of thousands of bugs.

This complexity presents a formidable challenge to the reliability of the devices. Even if we assume that the security threats can be controlled by virtualization, this is of limited use if failing subsystems degrade the user experience. It is necessary to construct embedded software so that it can detect faults and automatically recover from them. This is only possible if the effects of faults can be contained in relatively small components.

Virtualization is of very limited help here. The isolation provided by virtualization is by its nature coarse-grain — it provides the illusion of a complete machine for each subsystem. This means that each virtual machine is required to run its own operating system, making them relatively heavyweight. Increasing the number of virtual machines in order to reduce the granularity of the subsystems would create serious performance issues, and significantly increase the amount of code. This, in turn, not only requires increased memory size (and thus power consumption) but also more points of failure.

3.3.2 Integration

The subsystems of an embedded system are not independent, but must collaborate closely to achieve the system's mission.

Unlike a server that uses virtualization to run many independent services in their own virtual machines, embedded systems are highly integrated. Their subsystems are all required to co-operate closely in order to achieve the overall device functionality.

3.3.2.1 High-performance communication

This tight co-operation requires highly-efficient communication between subsystems, characterised by high bandwidth and low latency. This is the antithesis of the virtual-machine model, where each VM is considered a system of its own, which communicates with other systems via file systems or networks. The kind of communication required between components of an embedded system requires shared memory and low-latency signalling, requirements that simply do not fit the virtual-machine model.

Subsystems in embedded systems require highly-efficient communication. This requirement is fundamentally at odds with the virtual-machine approach.

This communication requirement has many aspects. One is bulk data transfer between subsystems, for example a media file that has been downloaded via the communications subsystem and is to be displayed by a media player. It is important for overall performance, as well as energy conservation, that such data is not copied unnecessarily, which is normally achieved by depositing it in a buffer that is shared (securely) between subsystems. This is not supported by the virtual-machine model.

3.3.2.2 Device sharing

The integration requires sharing of physical devices, which must be accessed (in a strictly controlled fashion according to some sharing policy) by different subsystems. A virtualization approach supports running device drivers in their native (guest) OS, but that means that a device is owned by a particular guest, and not accessible by others, and that the guest is trusted to drive the particular device.

A typical requirement for embedded systems is that a device must be accessible by several guests. For example, a graphic display may at times be partitioned with different subsystems accessing different sub-screens, while at other times subsystems are given access of the complete screen to the exclusion of all others. Other devices are not concurrently sharable but must be safely multiplexed between subsystems.

A straight virtualization approach can accommodate this by running the device driver inside the VMM. This requires porting all drivers to the hypervisor environment, with no re-use of guest OS drivers.

A straight virtualization approach runs device drivers inside a guest OS, limiting use of the device to a single VM, or as part of the hypervisor, requiring porting of the driver to the hypervisor environment.

A much better approach is to share a single driver between multiple VMs, without including it in the hypervisor. This requires that each participating subsystem has a device model for which it has a device driver. Typically the real device driver is contained in one of the participating subsystem, but a better (safer) solution is to separate it out into its own subsystem.

Access to such a device by each participating subsystem requires very low-latency communication across subsystems. This requirement is not served well by the virtual-machine model of network- or filesystem-based inter-VM communication. It requires a very lightweight (yet secure) message-passing mechanism.

3.3.2.3 Integrated scheduling

The tight integration of embedded systems is also visible at a very low level, that of the policy of scheduling many threads of execution on a single processor.

The virtual-machine approach to scheduling is inherently a two-level one: the hypervisor schedules virtual machines according to its resource-sharing policies. Whenever a particular VM is scheduled, its guest operating system schedules a particular thread according to its own policies. If the guest has no useful work to do, it schedules its idle thread. The hypervisor typically detects this special case and treats it as an indication that some other VM should be scheduled.

Scheduling of activities in an embedded system must be integrated and done according to a system-wide policy, not independent local policies of each virtual machine.

It is inherent in virtualization that the guest OS has no insight into what is going on in other machines. In particular, it has no notion of the relative importance of its own activities versus that of other VMs. The hypervisor can only associate an overall scheduling priority with each VM.

The implication of this is that low-importance (background) activities in a high-priority VM will always take priority over relatively high-importance activities in a lower-priority VM. In other words, the virtual-machine way of scheduling is inappropriate for embedded systems.

3.3.3 Security policies

Many embedded systems must meet critical security requirements. Virtualization alone does not help in addressing these requirements.

While it is essential that subsystems can communicate effectively and efficiently where needed, communication must be disabled where it is not needed or could lead to leakage of critical information. For example, bank-account access keys must be protected from disclosure, and licensed media content must be protected from copying.

This means that communication between components that is contrary to security requirements must not be permitted. Specifically, it must be possible to define system-wide security policies which define which communication is allowed across components, as indicated in Figure 3.4. For example, under digital rights management, a media player may only read but not write media content, and certain components are only allowed to communicate with the rest of the system via an encryption service.

In order to meet security requirements, it must be possible to define such security policies at system-configuration time, and it must be impossible for untrusted code to circumvent them.

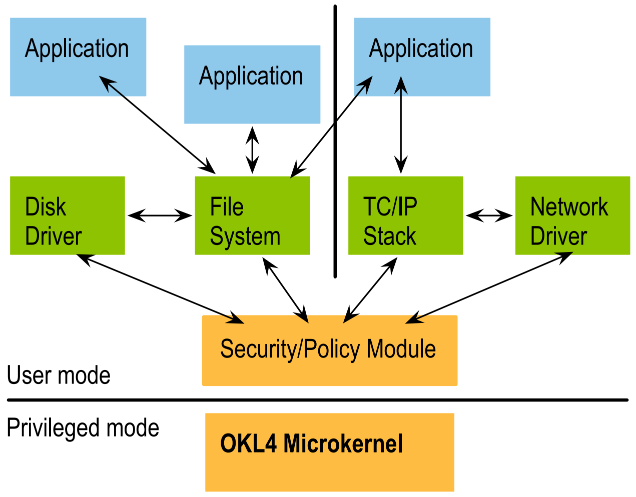

Figure 3.4.

Security mechanisms must allow separating subsystems in arbitrary ways according to a system-defined security policy. That policy defines which (if any) communications are allowed between sub-systems.

3.3.4 Trusted computing base

Many embedded systems contain components that are highly security-critical. An example would be an encryption service, which contains a driver for encryption hardware, and is used to secure financial transactions conducted through the device.

Such a service must obviously be particularly well protected from security compromises. Given the inherent bugginess of (almost) all software, it is important to minimise the security exposure of this service by minimising the amount of code on which it is dependent. The union of such code is called that service's *trusted computing base* (TCB). In general, a service's TCB is *the part of the system that can circumvent security*. The TCB must therefore be trusted to maintain security.

An application's TCB always includes all code that runs in the processor's privileged mode. This means that the kernel (or hypervisor, or VMM) is always part of the TCB. A pseudo-virtualized guest OS (which runs in privileged mode) is part of the TCB, even for applications which run outside that guest's virtual machine.

In the absence of a formal proof of correctness, the TCB must be expected to contain faults (bugs) like any other software. The best way to minimise exposure to such bugs is to *minimise the TCB*.

Secure subsystems, such as encryption services, require a minimal *trusted computing base*. This means their operation must depend on as little other code as possible. Virtualization *increases* the trusted computing base.

Virtualization does not support minimising the TCB. Compared to running a service on top of a native OS, running it in a virtual machine requires a hypervisor and a guest OS, both part of the TCB. Compared to a native OS, virtualization *increases the TCB*.

What is required is a framework in which trusted services can be built with a dependency on a minimal amount of other code. At the same time, the trusted service frequently has high performance requirements too, meaning that it must be able to communicate efficiently with the rest of the system. Virtualization does not serve this requirement.

4 Microkernels — A Better Solution

4.1 Embedded Systems Requirements

What would a suitable solution look like?

In order to best address the challenges discussed above, we would need a technology that has the following properties:

1. support for virtualization with all its benefits;
2. support for lightweight but strong encapsulation of medium-grain components that interact strongly, in order to build robust systems that can recover from faults;
3. high-bandwidth, low-latency communication, subject to a configurable, system-wide security policy;
4. global scheduling policies interleaving scheduling priorities of threads from different subsystems;
5. ability to build subsystems with a very small trusted computing base.

Property 5 is mandated by the *security principle of least authority* (POLA). As everything running in a privileged mode of the processor is inherently part of the TCB, POLA implies the need to minimise the amount of privileged code. Furthermore, it must be possible to provide a sufficient programming environment to support trusted service with a minimum of additional code, much less than a complete guest OS.

Property 3 means that we need the ability to share memory between components and we also need a highly-efficient low-latency mechanism for sending messages between components. Both must be subject to a configurable system-wide security policy.

Property 2 means that hardware mechanisms, particularly virtual-address mappings, must be employed in order to restrict the damage a component can do to its own data, and other data it has been explicitly given access to. Similarly, it means that a component's access to other system resources, such as devices and CPU time, is similarly controlled. This rules out running such components in privileged processor mode. It requires that it must be inexpensive to create, manage, schedule and destroy such components dynamically.

4.2 Microkernels

4.2.1 What are microkernels?

Microkernel technology provides the ideal foundation for meeting the above requirements. A microkernel is defined by Liedtke's *minimalism principle* [Lie95]:

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

A microkernel is a minimal privileged software layer that provides only general mechanisms. Actual system services and policies are implemented on top in user-mode components.

This minimality implies that a microkernel does not offer any services, only the mechanisms for implementing services. *Actual system services are implemented as components running in (unprivileged) user mode.* As such, a microkernel implements the principle of *separation of policy and mechanism* [LCC+75]: the kernel provides mechanisms that allow controlling resources, but the policies according to which resources are used are implemented in user-mode system components.

The microkernel approach leads to a system structure that differs significantly from that of classical "monolithic" operating systems, as shown in Figure 4.1. While the latter have a vertical structure of layers, each abstracting the layers below, a microkernel-based system

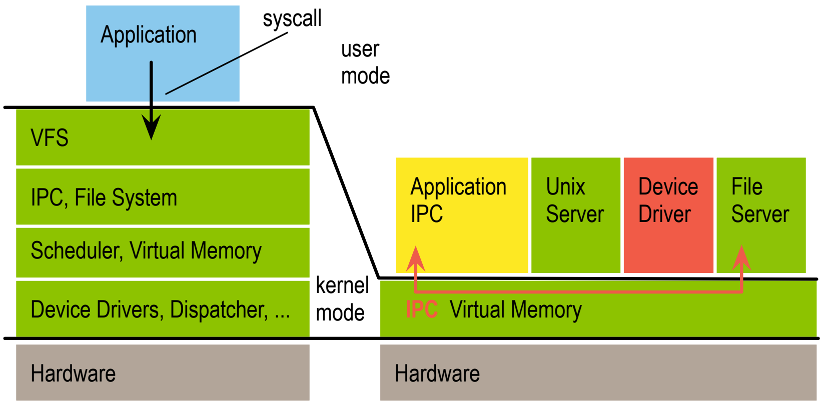

Figure 4.1. Structure of monolithic and microkernel-based systems

exhibits a horizontal structure. System components run beside application code, and are invoked by sending messages.

A main characteristic of a well-designed microkernel is that it provides a generic substrate on which arbitrary systems can be built, from virtual machines to highly-structured systems consisting of many separate (but interacting) components.

4.2.2 General properties of microkernel systems

A notable property of a microkernel system is that, as far as the kernel is concerned, there is no real difference between “system services” and “applications” — all are simply processes running in user mode. Each such user-mode process is encapsulated in its own hardware address space, set up by the kernel. It can only affect other parts of the systems (outside its own address space) by invoking kernel mechanisms, particularly message passing. In particular, it can only directly access memory (or other resources) if they are mapped into its address space via a system call.

The only difference between various kinds of processes is that some (generally a subset of system services) control resources, while others do not. Processes that control resources typically have them allocated via a system-configuration or start-up protocol.

This model is a good fit for embedded systems, where the distinction between “system services” and “applications” is frequently meaningless, due to the co-operative nature of the interaction of subsystems (cf. Section 3.3.2).

The minimal size of the kernel provides the basis for a *minimal trusted computing base*. A subsystem can be constructed such that it depends only on a small amount of support code (libraries and minimal resource management) besides the kernel.

The central mechanism provided by a microkernel is a message-passing communication mechanism, called *IPC*. In the horizontal system structure, IPC is used for invoking all system services, as well as providing other communication between subsystems. Due to its crucial importance, the microkernel’s IPC mechanism is highly optimised [Lie93, LES^{+}97] for minimal latency. A microkernel typically also provides mechanisms for setting up shared memory regions between processes, supporting high-bandwidth communication.

Most importantly in this context, a microkernel provides the right mechanisms for efficiently supporting virtualization. *The microkernel serves as the hypervisor, which catches virtualization traps. Unlike other virtualization approaches, the microkernel forwards the exception to a user-mode virtualization component, which performs the emulation (or signals a fault).*

4.3 OKL4 Microkernel Technology

OKL4 is Open Kernel's operating-system and virtualization technology. At its core is the OKL4 microkernel, the commercially-distributed and -supported member of the L4 microkernel family.

OKL4 is the world's most advanced commercial microkernel system, based on the OK team's 13 years of research leadership in the microkernel area, and hardened by several years of commercial deployments.

In this section we summarise the main characteristics of OKL4 technology as they relate to virtualization and beyond in embedded systems. Other Open Kernel white papers will cover specific aspects of OKL4 technology in more depth. Note that at the time of writing this white paper, not everything described in this section is fully supported by released OK products. However, the underlying technology exists and will be fully supported in OK products by mid-2008.

4.3.1 Low-overhead virtualization

L4 microkernels have a ten-year history of Linux virtualization. Performance is at par with specialised hypervisors.

For more than ten years L4 has been successfully used as a hypervisor for virtualizing Linux [HHL$^+$97, LvSH05]. While the approach used is essentially that employed years earlier by Mach [GDFR90, dPSR96], L4's vastly better IPC performance allowed it to succeed where Mach-based virtualization failed owing to intolerable overheads. The performance of OKL4-based virtual machines depends somewhat on the underlying processor architecture, but is generally within a few percent of the native performance. This overhead is about the same as that achieved by specialised hypervisors that lack the generality of the OKL4 platform.

A particularly interesting result is that of Linux virtualized on ARMv5 platforms. Here OK Linux (Linux para-virtualized on OKL4) outperforms native Linux in lmbench context-switching and other microbenchmarks, by factors of up to 50. This seemingly paradoxical result bears witness to the expertise of the OK kernel team. However, it also reflects the fact that it is much easier to thoroughly optimise a small code base of around 10,000 lines than a system of the size of the Linux kernel.

Figure 4.2 shows the structure of OK Linux. The hardware-abstraction layer (HAL) of Linux is replaced by a version that maps to the OKL4 "architecture". This OKL4-HAL is in fact mostly independent of the underlying processor architecture.

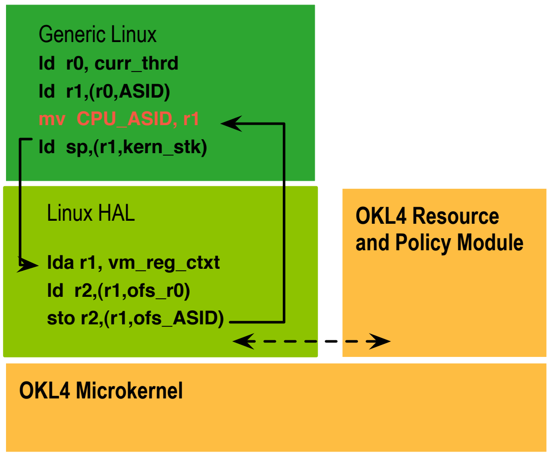

Figure 4.2. Virtualization in OK Linux.

A virtualization event is primarily handled inside the HAL: Either a sensitive instruction traps (as shown in Figure 2.2), invoking the hypervisor (OKL4 microkernel), which reflects the trap back into the HAL. Or the sensitive instruction was para-virtualized into a direct jump to virtualization code in the HAL (see Figure 2.3). The virtualization code then returns directly to the instruction following the virtualized one.

The HAL can handle some virtualizations directly because it holds copies of some virtual machine state, the real state is held outside the Linux kernel's address space for security reasons. Even where a virtualized instruction changes VM state, it is often possible to perform this action on the local copy, and synchronising with the master copy lazily on certain events (e.g. when the VM's time slice expires).

Some virtualization events require a synchronous change of the virtual state, e.g. where this changes the physical resource allocations. In such a case, the HAL invokes the resource- and policy-management module via an IPC message.

4.3.2 Unbeaten IPC performance

The key to performance of any system built on OKL4 is the high performance of its message-passing IPC mechanism. This is also the enabler for low-overhead virtualization: A system-call trap executed by a guest application in a virtual machine invokes the microkernel's exception handler, which converts this event into an IPC message to the guest operating system. The guest handles it like a normal system call. The system-call result is returned back to the guest application via another IPC message, which unblocks the waiting guest process.

Similarly, IPC is used to deliver interrupts to the guest OS's interrupt handler. It is also used to communicate with device drivers, and for communication and synchronisation between any components of the system, including between virtual-machine environments.

As the same mechanism is used for many different operations, it is highly optimised. Optimising IPC implicitly optimises the mechanism behind most critical system operations. As it is a relatively simple mechanism, it is possible to optimise it completely in virtually all of its aspects.

The core microkernel operation is message-passing IPC. The IPC performance of L4 kernels has not been beaten since the mid '90s.

IPC performance has been the hallmark of OKL4 and its predecessor L4 kernels since the beginning. *IPC performance data for those kernels has been published for years, and has never been beaten by other kernels.*

4.3.3 Efficient resource sharing

OKL4 provides mechanisms for efficient sharing of resources. Arbitrary memory regions can be shared by setting up mappings between address spaces. This is generally used to provide high-bandwidth communication channels between subsystems. Shared memory regions can be created with appropriate permissions. For example a buffer shared between processes in a producer-consumer relationship can be made accessible to the consumer read-only.

A typical scenario of communication via shared buffers is I/O via high-bandwidth devices. The device driver shares a buffer with a client in order to provide zero-copy I/O operations.

Another case of resource sharing is joint access to devices from separate subsystems, including virtual-machine environments. For example, a Linux system running in a virtual machine may need to access a device (touch screen, audio) that is also required by other subsystems.

As shown in Figure 4.3, a shared device will have a device driver which may live inside a virtual-machine environment, or in its own address space. The former allows reuse of the guest OS's native drivers (e.g. an unmodified Linux driver can be used), while the latter provides better security, as the driver is isolated from other code, leading to better fault isolation. In any case, *device drivers in OKL4 always run in user mode* (unless the hardware platform requires privileged execution).

Flexible, high-performance sharing of resources, in particular devices, is essential in embedded systems and is enabled by OKL4.

In such a scenario, other subsystems can access the device by communicating with the driver via an IPC protocol. In a virtual machine, this is achieved by inserting a *proxy driver* into the guest OS, which converts I/O commands into IPC messages to the real driver.

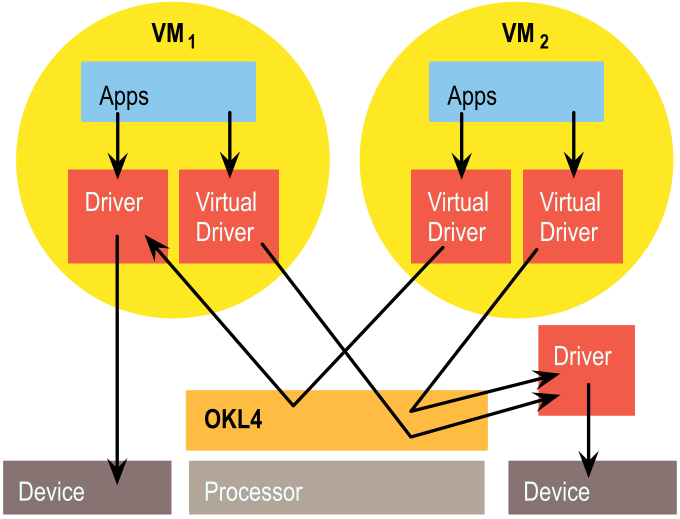

Figure 4.3.

Devices can be efficiently shared between virtual-machine environments, by the use of stub drivers. The real driver can either reside in a virtual machine (e.g. a native Linux driver) or run directly on OKL4 in its own protected address space.

4.3.4 Flexible scheduling

Normally it is important that the scheduling behaviour of a guest OS is not changed by virtualization. In OK Linux this is achieved by using the normal Linux scheduler to make scheduling decisions for Linux user processes. The microkernel's scheduler, in this case, is only used to schedule the complete Linux VM, and does not interfere in the scheduling of its internal tasks.

However, as indicated in Section 3.3.2.3, it is frequently desired to schedule some processes of a VM according to a different policy which takes the rest of the system into account. Figure 4.4 shows some examples:

- a high-priority VM (e.g. one that runs a real-time subsystem on top of an RTOS) may contain low-priority background threads which should only run when there is no other activity in the system;
- the system designer may prefer to run some real-time activities in an otherwise non-realtime VM (e.g. a Linux-based media player). Such an activity must be scheduled independently of the guest OS scheduler in order to achieve real-time performance.

This is achieved by allowing the guest operating system to select the appropriate global scheduling priority when scheduling its processes. This allows the guest operating system to run at a high priority when executing real-time threads, and a lower priority when executing background tasks. The range of priorities that a guest operating system can use is restricted so that it can not monopolise the access to the CPU. The mapping of operating system priorities to global system priorities is configured by the system designer.

4.3.5 Security

The OKL4 microkernel provides full mediation of resource allocation and communication according to a security policy defined by the designer of the embedded system.

The OKL4 microkernel mediates all resource access and communication in the system. A policy module controls who gets access to system resources (memory, devices, CPU), and who can communicate with whom. This policy module is outside the kernel (executes in user mode without hardware privileges), but is nevertheless a privileged part of the system (as it controls resources). All other code is subject to the policies imposed by this module.

Specifically, this policy module is responsible for mapping memory into address spaces (and virtual machines), giving it control over which memory can be shared and by whom.

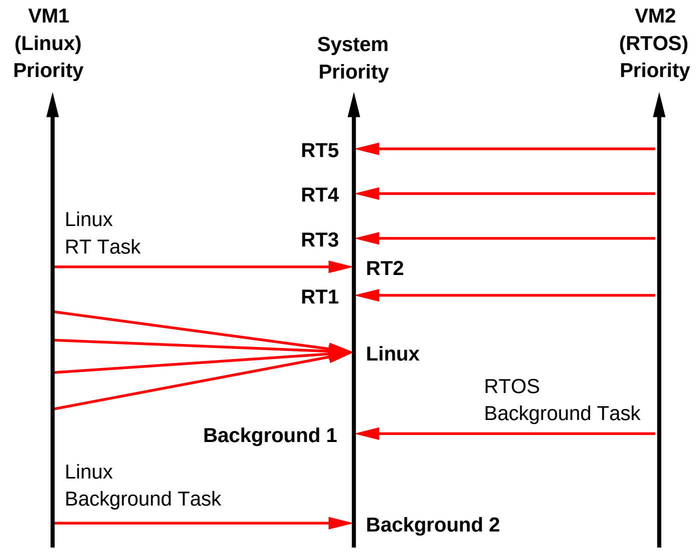

Figure 4.4.

Embedded systems require integrated scheduling. Linux real-time processes must have a global real-time priority, and low-priority background tasks of the real-time subsystem must run at lower global priority than Linux.

Also, devices are controlled by their drivers via memory-mapped I/O. By mapping device registers, the policy module controls who can drive a particular device.

The policy module also has a monopoly over operations that consume kernel memory; it can therefore control who is allowed to consume such kernel resources. This is important to prevent denial-of-service attacks on the system (e.g. by a rogue guest kernel).

Furthermore, the policy module controls the ability to send IPC messages across address spaces (and virtual machines). It can enforce policies governing which address spaces are allowed to communicate. For example, this allows encapsulating a subsystem such that it can only send messages to trusted subsystems. This can be used to prevent untrusted code from leaking data, such as sensitive personal data or valuable media content. The same mechanism is also used to confine applications of a particular virtual machine to that VM, e.g. restricting Linux processes to the Linux API and nothing else.

Finally, OKL4 runs all device drivers in user mode. This gives the system designer the ability to encapsulate drivers into separate address spaces, which limits the damage that can be done by a buggy or malicious driver, making it possible to use untrusted drivers. (In the case of bus-mastering DMA-capable devices this requires appropriate hardware support.) Note that this does not rule out the use of unmodified device drivers in the guest operating system (which itself runs in user mode).

4.3.6 Small trusted computing base

By keeping as much code as possible out of the kernel, the kernel itself can be made very small, around 10,000 lines, without restricting its universality. In fact, the strict separation of mechanisms (in the kernel) and policies (in user-mode components) ensures that the kernel can be used in arbitrary application scenarios and industry verticals.

A really big advantage of the small size of the kernel is that it allows minimisation of the amount of code that must be trusted, i.e., the system's trusted computing base. In contrast to plain virtualization approaches, which are designed to be always used with a guest OS underneath any other software, the amount of trusted user-mode code can be kept much smaller.

With OKL4, a minimal trusted computing base consists of the kernel, the user-mode policy module, and possibly some library code as required to support the security-critical code. The total TCB of a critical application can be kept as small as 15,000 lines, while concurrently running a large amount of untrusted code, as shown in Figure 4.5.

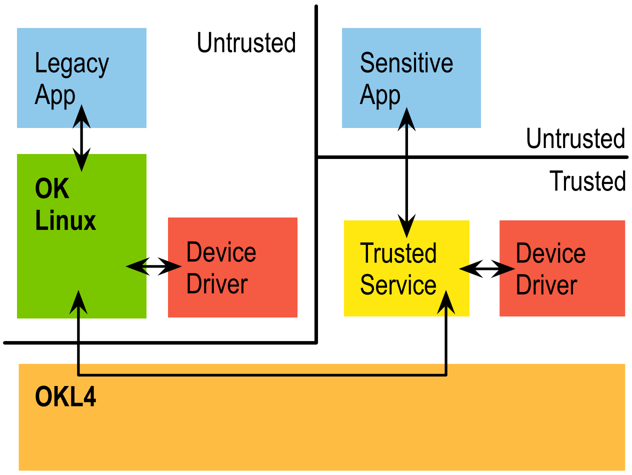

Figure 4.5.

Security-sensitive tasks can run in an environment with a minimal TCB, which includes the kernel, policy module and whatever functionality is required by the task, not more.

OKL4 allows the construction of components with a trusted computing base as small as 15,000 lines of code.

This means that the TCB can be made highly reliable. Standard software-engineering techniques, such as code inspections and systematic testing, can be used to reduce the number of bugs in such a small code base to maybe one or two dozen, a tiny fraction of the defects that must be expected for a hypervisor and guest OS combination that may be 100,000–300,000 lines in total.

However, even more is achievable. The small code size of OKL4 makes it possible to use mathematical methods to provide a formal *proof* of the correctness — more on that in Section 5.2.

4.3.7 Open-source software

Last but not least, OKL4 is open-source software. This means that the code is open for scrutiny, there is nothing to hide. The open source license allows evaluation, academic use, and use in the development of other open source software systems. Other uses of OKL4, including most commercial development uses will require a proprietary commercial license which is separately available from Open Kernel Labs.

4.4 Virtualization with OKL4 — Best of Both Worlds

In this paper we provided an introduction to virtualization and what it means in the context of embedded systems. *We pointed out the shortcomings of virtualization, and discussed why this means that a plain virtualization approach does not match the requirements for modern embedded systems designs.*

We then introduced microkernel technology in general, and Open Kernel's OKL4 microkernel in particular. We showed that on the one hand, OKL4 forms a suitable base for virtualization, but on the other hand overcomes the shortcomings of pure hypervisors.

OKL4 supports the construction of hybrid systems, containing virtual machines as well as highly-componentised code that runs in a native environment.

Specifically, OKL4 supports the construction of hybrid systems that combine virtualization with other approaches to system structure. OKL4 supports a large design space ranging from virtual machines with monolithic guest OSes on the one end, to highly-structured componentised designs [**?**] at the other end, as indicated in Figure 4.6.

Most importantly, both extremes (and everything in between) can be used in the same system. This can be used to integrate a monolithic guest in an otherwise highly-structured design, but also to evolve a monolith step-by-step into a more structured design.

For example, a media player, originally hosted in a VM with Linux as the guest OS, can be ported across to run in its own address space as a native OKL4 application. This can then run side-by-side with the Linux system (that still supports other applications), but also with a

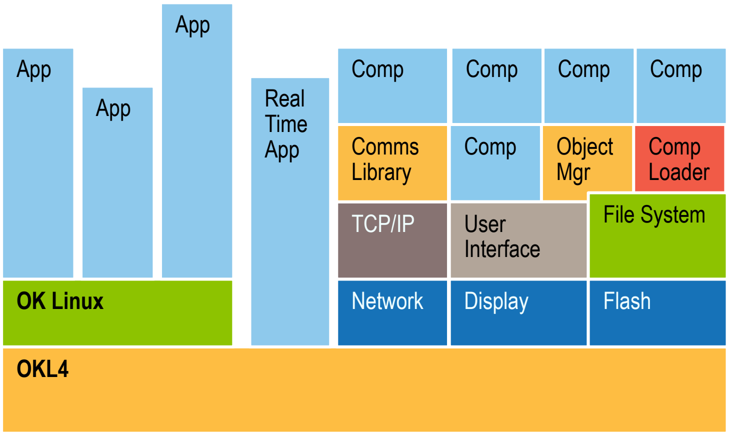

Figure 4.6.

A hybrid system contains virtual machines as well as code that runs in a native OKL4 environment (and can be highly componentised).

trusted crypto service which runs in a minimum-TCB environment. Over time, more components can be extracted from their monolithic environments (be it a high-level OS or an RTOS running a communications stack) into their own protected compartments. This includes device drivers, network stacks, file systems and other functional components.

Such an approach can dramatically improve the robustness of the system, by introducing internal protection boundaries which confine the damage caused by bugs.

OKL4 future-proofs embedded-system designs by providing a migration path towards highly componentised designs that exhibit fault containment, minimal trusted computing base, component reuse, and leverage the benefits of formal verification.

Even if initially a straight virtualization approach is seen to be sufficient for the designer's requirements, using OKL4 as the virtualization platform future-proofs the design: It allows the designer to move to a more componentised design over time, and the design will benefit from the unprecedented reliability that will be achieved with the formally-verified OKL4 kernel (see Section 5.2).

In this sense, OKL4 technology represents the best of all worlds for the design of embedded software systems.

5 The Future: Many Cores, Many Components, Many Nines

Finally, we do not want to conclude this paper without taking a glimpse of what is to come. The relevant question is how the current trends in embedded systems impact on virtualization and microkernel technology. Will it become more or less relevant, what is needed to keep it relevant, and is the technology heading in the right direction?

5.1 The Challenges

As the section title indicates, we see the future challenges imposed on embedded virtualization technology as many cores, many components, many nines. Let's examine what this means.

Many cores: this seems obvious. Multicore chips are already common place in high-end embedded systems, and "manycores" (chips containing 16 or more CPUs) are only a few years away. Legacy operating systems will find it increasingly harder to scale to such chips. Virtualization will be required to partition the chip into sub-domains containing a small or moderate number of processors that can be handled by a single guest OS. Obviously, this means that the virtualization technology itself must scale to the required number of cores.

Many components: embedded software will continue to grow in functionality and complexity. The use of modern software-engineering technology will become even more important, specifically component technology supporting fault isolation and reuse. This is where standard virtualization technology alone will increasingly become insufficient.

Many nines: the robustness requirements on embedded systems will grow (five, six, seven, eight "nines"?) At the same time, the increasing size and complexity of embedded software will make this level of reliability harder to achieve. Component technology and encapsulation will help, but only if the underlying software substrate that maintains the encapsulation (i.e., the trusted computing base) satisfies at least the overall system reliability goal. Present software technology cannot guarantee this, and much stricter assurance is required.

5.2 Future-Proofing Embedded Technology

Operating-system and virtualization technology is the lowest layer of software on which everything else is built, and on which everything else depends. It is therefore essential that embedded-system developers understand how this technology will meet the challenges of the future. This is particularly important for developers who are employing virtualization technology for the first time, and are therefore making a decision that will impact their future business for many years.

Future-proofing your technology requires virtualization technology that will adapt to future challenges.

In other words, it is important for developers to future-proof their technology, by choosing virtualization technology that will adapt to future challenges.

OKL4 is unique in this respect: The technology has a long track record of research leadership that is unmatched by competing products; at the same time the technology is proven in end-user deployments. OKL4 is also unique for the comprehensiveness and ambition of the present portfolio of R&D projects [**?**] conducted jointly by Open Kernel Labs and NICTA. Here we list a few highlights:

Scalability: The code base of the OKL4 microkernel was designed from the beginning to enable high multiprocessor scalability (among others by minimising global data structures). Recent research has demonstrated how this code base can be made to scale to 100s of processors [**?**].

Component technology: A new, light-weight component technology aimed specifically at embedded systems has been developed [**?**]. This work forms the basis for providing a modern software-engineering framework on top of OKL4 that will support high performance, strong encapsulation, fault tolerance, code reuse and real-time analysis.

Verification: The holy grail of system reliability and security is a mathematical proof of its correct operation. No system has such a proof at present, but OKL4 is closer than any other. In fact, a formal correctness proof of the kernel is a core part of Open Kernel's R&D roadmap [**?**], and work is on track to deliver a proof of the correctness of an implementation of the kernel by mid-2008. Verification is enabled by the small size and disciplined design of the OKL4 microkernel and will enable unprecedented reliability and security, to the benefit of all users of the technology.

Real-time guarantees: Real-time guarantees are difficult to establish for code running in user-mode. They require a complete timing analysis of the underlying privileged code. Present industry practice of comprehensive benchmarking cannot guarantee worst-case latencies. Work is in progress [**?**] on a complete, sound and reliable evaluation of the timing behaviour of OKL4, something that has never been achieved for any general-purpose kernel supporting memory protection.

Security: Work is highly advanced [**?**] on a revision of the API that will support highest security requirements, such as formal proofs of separation properties. Customers will have a smooth upgrade path to this advanced technology.

OKL4 is the future of embedded virtualization technology.

The best way for developers to future-proof their technology is to base it on the technology of the future. OKL4, proven in the present, is the future of embedded-systems virtualization technology.

Bibliography

[dPSR96] Françcois Barbou des Places, Nick Stephen, and Franklin D. Reynold. Linux on the OSF Mach3 microkernel. In *First Conference on Freely Distributable Software*, Cambridge, MA, USA, 1996. Free Software Foundation. Available from `http://pauillac.inria.fr/~lang/hotlist/free/licence/fsf96/mklinux.html`.

[Fre07] Free Software Foundation. Frequently asked questions about the GNU GPL. `http://www.fsf.org/licensing/licenses/gpl-faq.html`, 2007. Last visited July 2007.

[GDFR90] David Golub, Randall Dean, Allesandro Forin, and Richard Rashid. Unix as an application program. In *Proceedings of the 1990 Summer USENIX Technical Conference*, June 1990.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of µ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on OS Principles*, pages 66–77, St. Malo, France, October 1997.

[LCC+75] R. Levin, E.S. Cohen, W.M. Corwin, F.J. Pollack, and W.A. Wulf. Policy/mechanism separation in HYDRA. In *ACM Symposium on OS Principles*, pages 132–40, 1975.

[LES+97] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for extensibility). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 28–31, Cape Cod, MA, USA, May 1997.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 175–88, Asheville, NC, USA, December 1993.

[Lie95] Jochen Liedtke. On µ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[LUC+05] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report PA005520, National ICT Australia, October 2005.

[LvSH05] Ben Leslie, Carl van Schaik, and Gernot Heiser. Wombat: A portable user-mode Linux for embedded systems. In *Proceedings of the 6th Linux.Conf.Au*, Canberra, April 2005.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):413–421, 1974.

[SN05] James E. Smith and Ravi Nair. The architecture of virtual machines. *IEEE Computer*, 38(5):32–38, 2005.

[Was07] LKM's should not be used to evade the GPL. `http://www.wasabisystems.com/LKM/summary/`, 2007. Last visited June 2007.

About the Author

Dr Gernot Heiser is co-founder and Chief Technology Officer of Open Kernel Labs (OK). As Chief Technology Officer, his specific responsibility is to set the strategic direction of the company's research and development, in order to maintain and further expand OK's technology leadership.

Prior to founding OK, Dr. Heiser created and lead the Embedded, Real-Time and Operating Systems (ERTOS) research program at NICTA, the Australian national centre of excellence for information and communications technology, and has established ERTOS as a recognised world leader in embedded operating-systems technology. Dr Heiser continues in this position on a part-time basis, in order to ensure the strategic alignment of OK and ERTOS, and the smooth transfer of ERTOS research outcomes for commercialisation in OK.

Prior to NICTA's creation in 2003, Dr Heiser was a full-time faculty member at the University of New South Wales (UNSW), where he created a suite of world-class OS courses, lead the development of several research operating systems, and built the group that provided the foundation for ERTOS and later OK. He still holds the position of Professor for Operating Systems at UNSW, the only such chair in Australia, and continues to teach advanced-level courses and supervise a large number of PhD students.

Gernot Heiser holds a PhD in Computer Science from ETH Zurich, Switzerland. He is a senior member of the IEEE, and a member of the ACM, Usenix and the Australian Institute of Company Directors.

About Open Kernel Labs

Open Kernel Lab (OK) is a leading provider of embedded systems software and virtualization technology. Spun out from NICTA, Australia's prestigious centre of excellence for information and communications technology, OK is focussed on driving the state of the art in embedded operating systems. OK's technology aims at improving the reliability, safety and security of embedded devices.

OK believes that the best technology should have nothing to hide, and consequently distributes its code as open source. The company also believes that dramatic improvements in system reliability are possible in the near future, and to this end collaborates closely with NICTA and other research institutions on creating and commercialising the next generation of embedded operating-systems technology. For more information on OK and its products visit `http://www.ok-labs.com`.