# Virtualization for Embedded Systems

*Is an open source solution right for you?*

6/26/2013

Julia Keffer

# Table of Contents

# Table of Figures

# Introduction

Embedded computers are part of our everyday lives, from smart phones, to cars, to gaming consoles. Virtualization was predominantly used first in the server market, but today it has come to the embedded computer. Undoubtedly, you have used an embedded computer that employs virtualization technology.

This paper explains what virtualization is, how different virtualization technologies work, and how virtualization is applied in embedded applications. It examines a set of criteria for choosing a virtualization solution and evaluates three open source implementations against each of the criteria.

## What is Virtualization?

The Computer Desktop Encyclopedia defines a virtual machine (VM) as “An operating system that runs like a "machine within a machine", and functions as if it controls the entire computer. The operating systems in each VM partition are referred to as guest operating systems or partitions, and they communicate with the hardware via a program called a virtual machine monitor (VMM), which is also referred to as a hypervisor. The hypervisors "virtualizes" the hardware for each guest operating system (1).” Figure 1 shows the difference between non-virtualized and virtualized systems.

This paper uses the terms hypervisor and guest to refer to the VMM and the guest operating system, respectively.

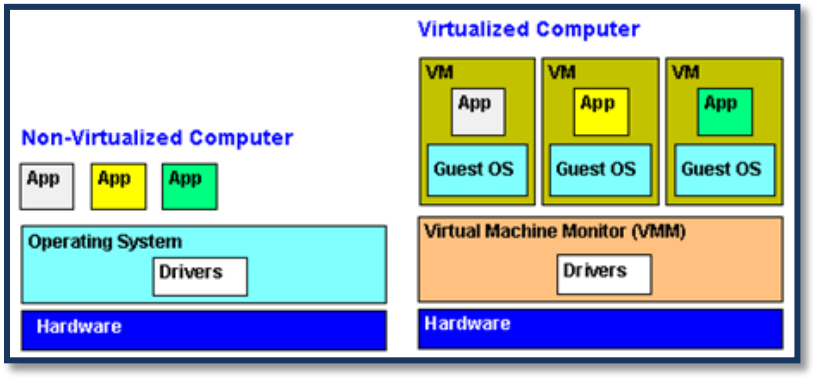


Figure 1: Non-Virtualized and Virtualized Computer

The hypervisor operates in a privileged environment, also referred to as kernel mode or supervisor mode, where it has access to low level system calls to mediate resource access.

The entire guest operates in a non-privileged environment, also referred to as user mode. Guests communicate with the hardware only through the hypervisor and cannot use low level system calls directly.

The number of guests that can run on a single hardware platform is constrained by the available hardware resources, typically the amount of memory. Each guest can run a different operating system.

# Virtualization Applications

Virtualization is useful either to consolidate multiple computer systems on the same hardware to reduce costs or to isolate programs running on the same hardware. This section describes three situations where it is useful to run multiple isolated operating systems on the same hardware in embedded systems.

### Operating Systems with Different Run-time Requirements

Virtualization provides the ability to run different types of operating systems on the same hardware, such as a full featured OS for user interface functions, and a real-time OS for time critical applications. For example, in an automobile, the computer that controls the anti-lock brake system has real-time requirements, while the infotainment system does not. Without virtualization an automobile required two different computers; using virtualization both systems can run on the same hardware, reducing costs.

### Isolate Security Conscious Applications

Virtualization can isolate security-conscious applications from insecure applications. For example, if an application running on a smart phone introduces a computer virus, it is necessary to protect the wireless protocol stack to ensure that the system can still make phone calls. One way to do this is to run each of these components inside separate operating systems in a virtual environment.

### Open Source Compliance

Open source licenses typically allow proprietary code to interact with open source code if the two communicate only via a messaging interface. If the open source and proprietary code run in separate operating systems, the hypervisor fulfills this requirement.

# Virtualization Architectures

There are different virtualization technologies; this paper focuses on type 1, or bare metal hypervisors, where the hypervisor software runs between the hardware and the guest. There are three categories of type 1 hypervisors: full virtualization, paravirtualization, and a microkernel.

With full virtualization, a whole system is emulated (basic input/output system (BIOS), disk, processor, network interface) and a guest runs unmodified on a hypervisor that provides the abstraction of the underlying computer system. The guest is not aware of the hypervisor. The hypervisor intercepts hardware access instructions from the guests and invokes the instructions on behalf of the guest. Full virtualization requires hardware extensions in the computer processor, such as Intel's VT-x technology.

Figure 2 shows a paravirtualized system in which the guest requires modifications to work in a virtual machine (2) and communicate with the hypervisor. Specifically, some or all of the device drivers in the guest are modified to replace the privileged instructions with direct requests to the hypervisor, which are referred to as hypercalls.

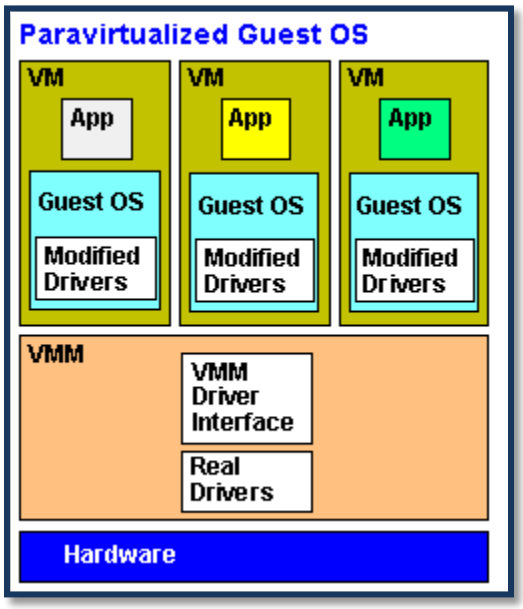


Figure 2: Paravirtualized System

In theory, any operating system which provides access to the source code, such as Linux, can be paravirtualized, unlike proprietary operating systems, such as Windows. Paravirtualization is typically used in systems with hardware that does not support virtualization, although it may still be advantageous to use it for performance reasons. Full virtualization is often not as efficient as paravirtualization, because of the extra step required to intercept the privileged instructions.

Although not originally designed for virtualization, it is possible to implement virtualization on top of a microkernel. A microkernel is a reduced version of a regular operating system kernel that provides a set of policies and mechanisms to access hardware resources.

Any component can run on top of the microkernel. An operating system is a type of component, but it can run alongside a standalone application, such as a special device driver that runs directly on top of the microkernel.

As with a hypervisor, the microkernel software runs in kernel mode between the hardware and the guest. Unlike a hypervisor, a microkernel does not perform the instructions on behalf of the guest. Instead of calling the privileged instructions directly, the microkernel forwards the request to a user mode virtualization component, which interprets the request. The component may reside inside a guest or it may be a standalone component.

The mechanism used to forward the request is called inter-process communication (IPC). As with paravirtualization, the device drivers in the guest must replace hardware access instructions with IPC messages.

# Key Criteria for a Virtualization Solution

Before considering virtualization in an embedded system, some of the factors you should consider are:

- hardware support
- operating system support
- resource allocation and sharing
- memory isolation
- processor scheduling
- guest communication
- size of the trusted computing base

## Hardware Support

The hypervisor or microkernel software runs directly on top of the hardware, and therefore must support the instructions required by the hardware architecture. Common architectures in embedded systems are x86, ARM, PowerPC, and Sparc. The x86 processor is typically used in industrial and medical applications. Smart phones and tablets almost exclusively use ARM processors. The Sparc architecture is common in military and avionics systems. Gaming consoles use PowerPC architectures.

To support full virtualization, the hardware must include virtualization extensions. Intel and AMD both include virtualization support in their x86 processors. The ARM Cortex A15 and A7 processors also support virtualization.

Full memory isolation requires a memory management unit (MMU).

## Operating System Support

Hardware and operating system support are closely related. As mentioned previously, proprietary operating systems (such as Microsoft Windows) need hardware support because they cannot be paravirtualized.

If paravirtualization is necessary or desirable, the operating system source code must be available. FreeBSD, NetBSD, and all variants of Linux freely distribute their source code. Some Linux operating systems already include the paravirtualized drivers (3).

## Resource Allocation and Sharing

Guests must share some hardware resources, such as disks and network interfaces. An application in the system may require dedicated access to a particular device, such as a USB port, which means that the hypervisor must provide a mechanism to assign exclusive access to the device to a specific guest.

## Memory Isolation

The memory allocation scheme must ensure that guests cannot access memory outside their own address range. It is important to note that any truly secure implementation requires hardware support by an MMU to guard against a malicious device driver that uses direct memory access (DMA). Both Intel and AMD processors have MMU support, as do some ARM processors.

## Processor Scheduling

Execution isolation is important to prevent a rogue application on a guest from monopolizing the CPU, which essentially functions as a denial-of-service attack on the rest of the guests. If one of the guests requires a real-time response, the hypervisor must use a scheduling algorithm that can assign it a higher priority. It is also desirable to have a way of ensuring that lower priority tasks in one guest do not preempt higher priority tasks in another guest.

## Guest Communication

Guests may want to exchange information. For example, a component may need to provide status information for a user interface to display. If there is a mechanism to enable guests to communicate, the solution must prevent a security breach using this mechanism. Any risk typically results from the mechanism the guests use to store the data to exchange.

## Size of the Trusted Computing Base

The size of the trusted computing base that implements virtualization affects system robustness. All code has a certain number of defects and the smaller the trusted computing base, the fewer defects there are likely to be. Because the code runs in privileged mode, it must be possible to contain the faults to the virtualization code without affecting the guests. The code with access to the privileged instructions is referred to as the trusted computing base (TCB).

# Open Source Solutions

Open source solutions are often appealing because the code is free and can be modified according to the needs of the system. An ideal open source solution has active development and community members willing to informally support users.

One potential drawback of open source is the requirements under GNU GPL[1] to release the source code for any derived work. If is necessary to make proprietary changes to the virtualization code, open source code may not be an appropriate solution.

The open source solutions described in Table 1 are licensed under either GPL or a proprietary license with the same conditions as GPL.

Table 1: Summary of Virtualization Solutions

| Criteria | Xen | Xtratum | OKL4 |
|---|---|---|---|
| **Hardware support** | Intel x68, AMD, ARM v5-v7, ARM CortexA15 (experimental) | LEON3 (Sparc V8) and Intel Itanium-64 processor | ARM v5/v6 and Intel i386 processor; requires an MMU |
| **Operating system support** | Any paravirtualized guest, proprietary OS on Intel x86 and AMD | Any paravirtualized guest | Any paravirtualized guest |
| **Resource allocation and sharing** | Domain 0 mediates shared access; exclusive access using PCI pass-through interface on Intel VT-x or AMD-V hardware | Exclusive access using configuration settings; shared access requires user-implemented communication protocol | Mediated using IPC messages; exclusive access via policy module configuration |
| **Memory isolation** | Hypervisor memory tracking, Domain 0 grant tables for each guest | Configurable memory area statically assigned to each guest, no shared memory | Static and shared access configured in the resource and policy module |
| **Processor scheduling** | Configurable weight and CPU cap, no exclusive guest CPU access | Configurable timeslot and duration with a fixed, cyclical scheduling algorithm | Global policy which assigns process priorities across the guests |
| **Guest communication** | Virtual network interface in domain 0 using standard communication protocols | Port-based communication using a predefined protocol | The basis of the implementation |
| **Size of the trusted computing base** | Large – Domain 0 is an entire operating system | Small hypervisor code base | Small microkernel code base |

[1] Refer to (17) for more information.

## Xen

Xen is a bare metal hypervisor. There are two parts to the hypervisor implementation: a hypervisor and a special paravirtualized guest (see Figure 3). The hypervisor code that runs directly on top of the hardware is responsible for virtualizing the CPU, memory, and input/output (I/O) control, including interrupt handling.

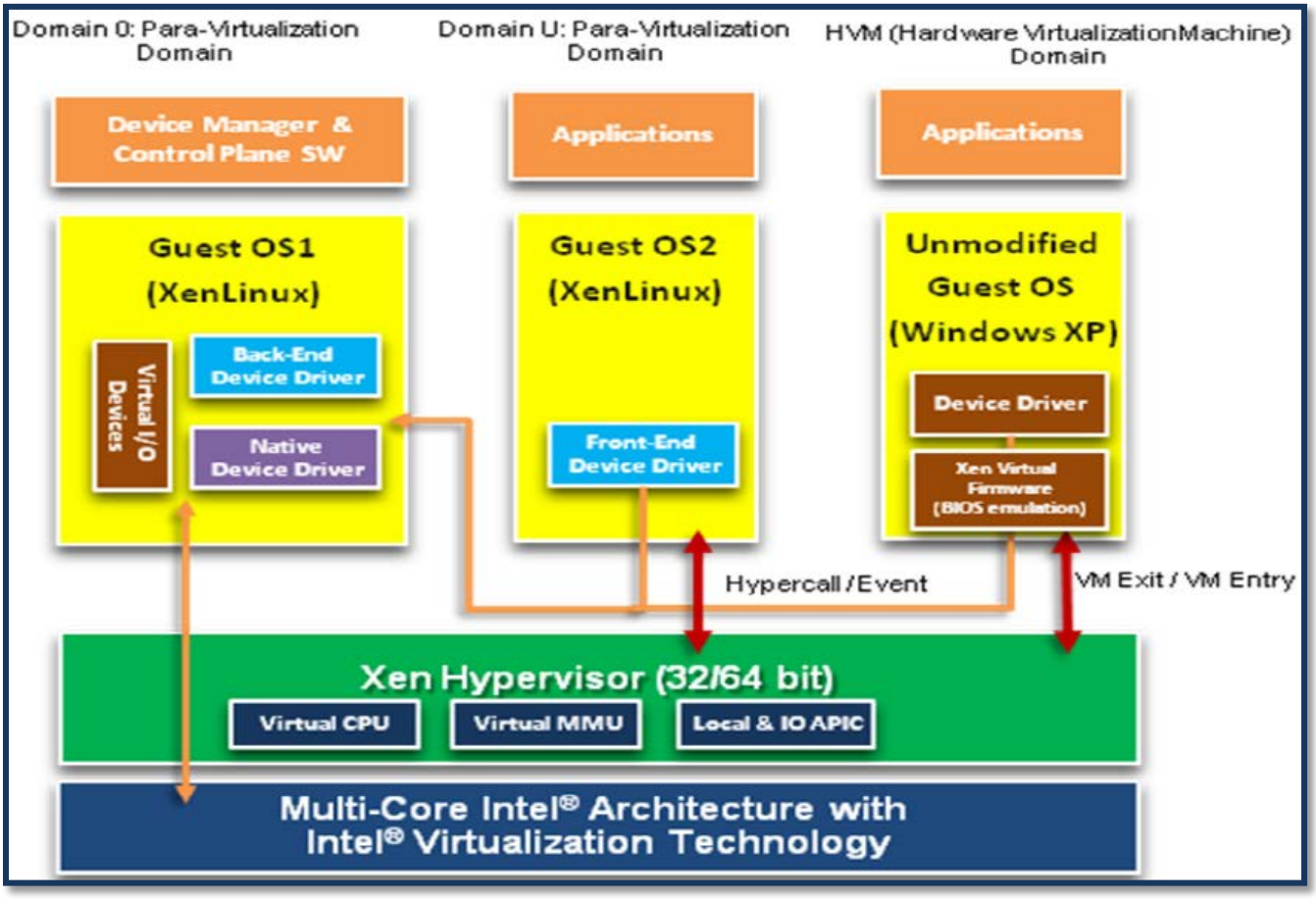


Figure 3: Xen Hypervisor

The special paravirtualized guest (referred to as Domain 0) has privileged access to the hardware. It manages processor and memory sharing, network and disk access, and communication between guests. Domain 0 can run any paravirtualized operating system, but it is typically a variant of Linux, as many Linux distributions include native Xen support.

Xen supports both full virtualization and paravirtualization. Xen has an active development community. Refer to (4) for more information.

**Hardware Support** - Xen can run on x86 processors from Intel and AMD. Xen support for ARM processors is a project led by Samsung which delivers and maintains Xen support for a range of ARM processors (ARM v5 - v7) for mobile devices. The project is also working on providing real-time guarantees in a virtualized environment and multi-processor support. Refer to (5) for information. An experimental version of Xen which uses the virtualization support introduced for the ARM Cortex A15 is underway. An experimental project to port Xen to PowerPC was abandoned.

**Operating System Support –** Xen supports any guest that can be paravirtualized. It supports full virtualization for any guest that runs on Intel or AMD x86 hardware with virtualization extensions. Many Linux distributions include the virtual device drivers to support paravirtualization on Xen.

**Resource Allocation and Sharing –** Domain 0 mediates access to I/O devices by receiving requests from the guests on a virtual channel. Xen also supports a PCI pass-through interface (see Figure 4) to allow guests direct and exclusive hardware access to PCI devices, such as the network interface. This hardware access method requires Intel VT-x or AMD-V hardware support, and can be used with paravirtualized and fully virtualized guests. The guest must have a native device driver for the device and the Domain 0 guest must have a "pciback" version of the driver.

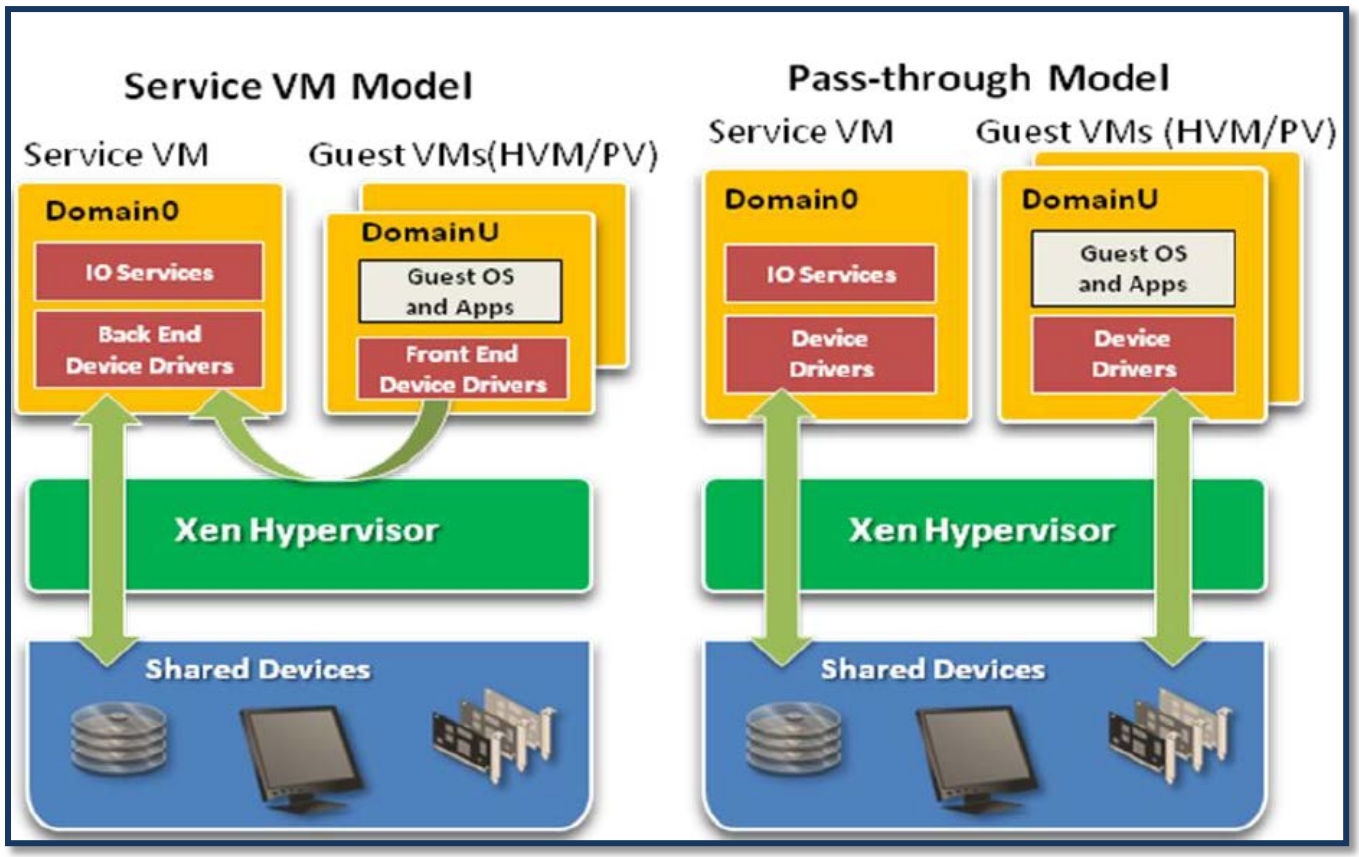


Figure 4: PCI Pass-Through

The PCI pass-through feature must be enabled in both the BIOS and in Xen, and is a potential security risk if a guest runs a malicious application or has defects in its device driver.

Additional pass-through support is available for USB devices and selected graphics devices. A mechanism called SR-IOV allows devices to be assigned to guests but shared among them. Refer to (6) and (7) for information about pass-through functionality.

**Memory Isolation –** Xen assigns a static area of memory to each guest. The hypervisor uses shadow pages to translate virtual memory access requests from the guests into the physical address. It tracks which guest owns the memory to enforce isolation (8). Newer releases of Xen implement a feature to allow identical guests to share physical memory for common binaries and libraries. This feature is still in the beta stage and does not have security support. Xen uses shared memory to implement guest communication. Domain 0 manages grant tables which grant access to guests on a per page basis to ensure safe memory sharing (9).

**Processor Scheduling –** Xen has settings to configure the CPU usage across guests. It load balances across CPUs using a weight and cap (limit) for each guest. It is possible to configure a guest to use only to a specified set of CPUs; however, a guest cannot be assigned exclusively to a single CPU. CPU configuration takes into account physical devices and hyperthreading.

**Guest Communication –** Guests communicate through a virtual network interface in domain 0, which implements routing and bridging functionality. Communication is based on standard networking protocols. The default implementation of guest communication requires significant overhead. The XenLoop and XenSockets projects attempt to address this issue (10).

**Size of the Trusted Computing Base –** Although the portion of Xen that resides on top of the hardware is small, domain 0 is a complete operating system. The TCB for Xen is quite large and susceptible to defects in many areas. A newer version of Xen on ARM is experimenting with moving device drivers outside of domain 0 to isolate them from the rest of the computing base.

## Xtratum

Xtratum is a bare metal hypervisor that runs in privileged mode and virtualizes the CPU, memory, interrupts, and any devices that endanger isolation. A guest, which Xtratum documents refer to as a partition, must be paravirtualized to run on Xtratum and replace calls to privileged instructions with hypercalls. See Figure 5.

Xtratum supports a special system guest with extra privileges. It can use a special set of hypercalls to manage and monitor system resources, and stop, start, or reset partitions. The access rights of a system guest are set in the Xtratum configuration file. System guests do not have direct hardware access.

The scheduling and IPC mechanisms are modeled on the Avionics Application Standard Software Interface (ARINC) 653 standards, although its goal does not include compliance to the specification. The Universidad Politecnica de Valencia in Spain developed Xtratum. Refer to (11) for more information.

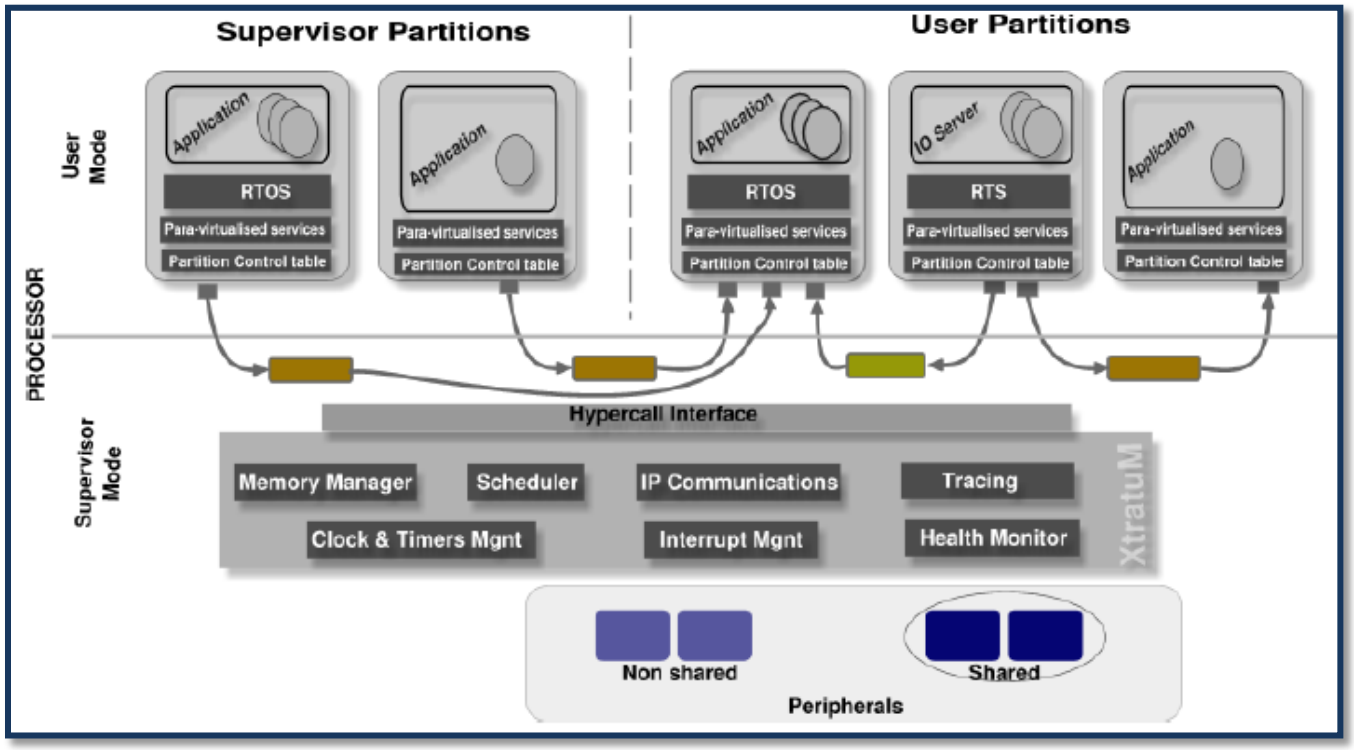


Figure 5: Xtratum Architecture

**Hardware Support –** Xtratum supports the LEON3 (Sparc V8) and Intel Itanium-64 processor, neither of which have virtualization extensions. A theoretical paper was published about porting Xtratum to PowerPC but was never implemented (12).

**Operating System Support –** Any paravirtualized guest can run on Xtratum.

**Resource Allocation and Sharing –** I/O ports and interrupts that the hypervisor does not manage are assigned exclusively to guests in the configuration file. The device driver resides in the guest. To share devices, the system designer must implement an I/O server partition which receives requests from other guests via IPC and processes them according to its policy configuration.

**Memory Isolation –** The configuration file defines the memory area statically assigned to the guest. There are no shared memory regions. Xtratum will run without an MMU, in which case, there is a risk of unauthorized memory access.

**Processor Scheduling –**Xtratum uses a fixed, cyclical scheduling algorithm based on the timeslot and duration settings for each guest in the configuration file. Each guest may define multiple scheduling plans and can notify the hypervisor to switch plans using a hypercall. For example, a guest may want to switch into a maintenance mode when it has only low priority tasks to do, freeing the processor for use by other guests with higher priority tasks. A system guest can also change the scheduling plan of a normal guest.

**Guest Communication –** The hypervisor implements a port-based communication mechanism. Guests send and receive messages from each other or the hypervisor on a channel that links two ports. The protocol is specific to the sending and receiving parties. Both broadcast and direct messaging modes are available. Channels, ports, maximum message sizes, and maximum number of messages (queuing ports) are defined in the configuration file. Data exchange relies on buffer copying mechanisms, as there are no shared memory regions.

**Size of the Trusted Computing Base –** The critical code for Xtratum is limited to the small hypervisor code base. It uses a health monitor feature to detect and react to errors to contain them within the proper scope: process, guest, hypervisor, or firmware.

## OKL4

OKL4 3.0[2] is a microkernel implementation of virtualization. The microkernel runs on top of the hardware in kernel mode and uses IPC to mediate requests for interrupts and device drivers between guests. See Figure 6. Separate components running in user mode provide system services; the microkernel does not provide them.

A separate resource and policy model, which runs outside the microkernel in user space, holds the configuration of the CPU scheduling policy and the memory allocation. The microkernel runs in privileged mode and the guests run in user mode. Guests must replace hardware access instructions in their device drivers with IPC messages. Open Kernel Labs, which is owned by General Dynamics, sponsors the OKL4 project. Refer to (13) for more information.

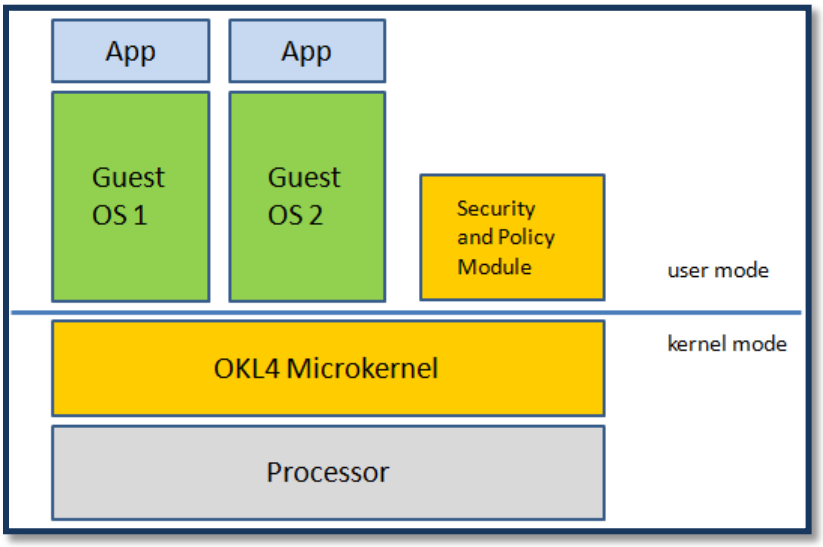


Figure 6: OKL4 Architecture

**Hardware Support -** OKL4 supports ARM v5/v6 and Intel i386 processors, none of which have virtualization extensions. OKL4 must run on a processor with an MMU.

**Operating System Support –** Any paravirtualized guest can run on OKL4. Open Kernel Labs provides a paravirtualized version of Linux to use as a guest.

[2] Not to be confused with the OKL4 4.0 microvisor, which requires a commercial license

**Resource Allocation and Sharing –** OKL4 mediates device access using IPC messages (Figure 7). It relays requests for device access from a guest's virtual driver to the physical driver, which may be either a standalone driver, or reside in another guest. The policy module controls which guest can drive a particular device by mapping device registers.

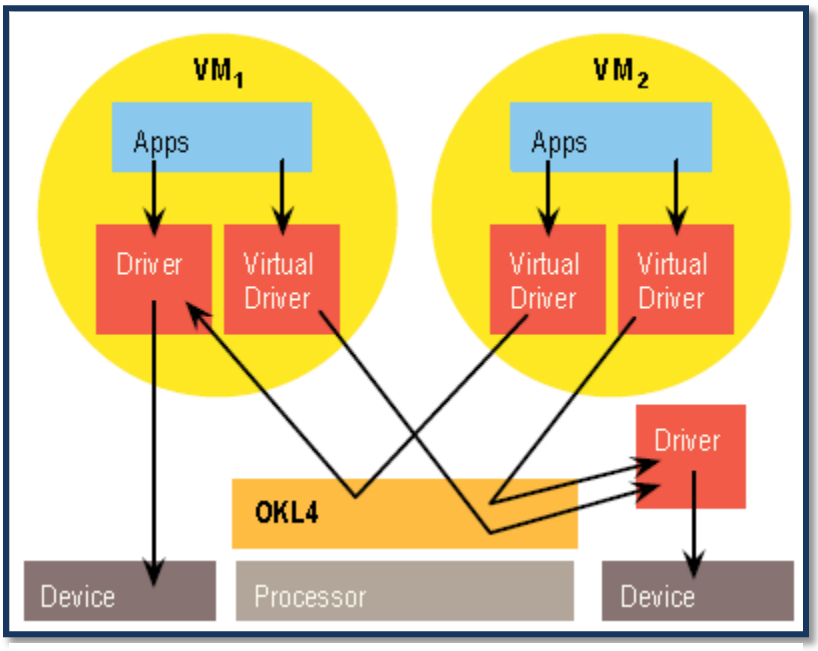


Figure 7: OKL4 IPC Model

**Memory Isolation –** The memory allocated for each guest is statically configured in the resource and policy module. There are also configuration settings for shared memory regions and policy settings to determine which guests can access the shared regions. The policy module has a monopoly over operations that consume kernel memory; it can control which guest is allowed to consume such kernel resources to guard against denial-of-service attacks on the system, for example, by a rogue guest kernel.

**Processor Scheduling –** The system designer configures the CPU scheduling algorithm in the security and policy module. The policy can schedule the entire guest or an individual process within a guest. A global policy which assigns process priorities across the guests ensures that lower priority maintenance tasks in one guest do not block higher priority processes in another guest. The range of priorities the guest can use is restricted to ensure that it does not monopolize the system.

**Guest Communication –** Guest communication is the heart of the OKL4 implementation and performance has been highly optimized. In addition to device access requests to the microkernel, guests can exchange information based on an agreed-upon protocol. Guests may make use of the shared memory regions to facilitate buffer allocation and access to shared data.

**Size of the Trusted Computing Base –** The microkernel code base is very small and therefore, less likely to have defects. Much effort has gone toward providing a formal proof of correctness of the kernel. Its separation from the policy module, which is located in user space, allows flexibility while limiting the code that has access to privileged activities.

# Conclusions

The low cost of an open source solution may seem appealing; however, an open source solution is suitable only for certain applications and development organizations. Table 2 summarizes the benefits and limits of each solution described in this paper.

Table 2: Solution Evaluation

| | Xen | Xtratum | OKL4 |
|---|---|---|---|
| **Pros** | Supports full virtualization on selected hardware<br><br>Many Linux versions include Xen drivers, which may result in lower development costs<br><br>Active community development | Health monitor limits the scope of errors<br><br>Secure guest communication | Extremely flexible scheduling and guest communication<br><br>Small, formally verified code base<br><br>Policy module effectively isolates guest memory regions and protects against CPU monopolization |
| **Cons** | Large TCB<br><br>Default guest communication mechanism requires significant overhead | Limited hardware support<br><br>Resource sharing requires development of a guest communication protocol | Limited hardware support<br><br>Will not run without an MMU |
| **Best for** | Applications that require proprietary guest operating systems<br><br>Applications that need limited guest communication<br><br>Users that do not want or need to implement specialized features | Applications that require specialized features and that have the flexibility to use any hardware architecture<br><br>Experienced users; they need to implement at least the paravirtualized drivers | Applications that require specialized features and that have the flexibility to use any hardware architecture<br><br>Applications that require the high levels of security<br><br>Experienced users with low-level programming expertise |

The main consideration for choosing a virtualization solution for an embedded system is the hardware platform. With an open source solution, while it is possible to implement one of the solutions on a different platform, it may require significant effort.

Closely tied to the hardware solution is the choice of operating system. Most Linux distributions work well on any of the solutions, although Linux does not have native real-time support (requires a kernel patch). To save time and effort, you may also want to choose an operating system that has paravirtualized drivers already available.

Once you choose the hardware and operating system, the needs of the application drive the choice of the virtualization solution. You need to consider how the implementation of the virtualization solution affects the design of your application. You may need to implement an I/O server to share devices, or you may be able to take advantage of hardware support to have exclusive access to a device. A particular CPU sharing mechanism may be a better fit, depending on the scheduling constraints of your application. The use of a shared memory region to exchange data between guests may be a benefit or a concern.

Choosing the right solution is important to ensure that you can create an application that is easy to design and maintain. The information about the available open source options can help you decide if any of the options suit your needs, and you have a base of comparison for a commercial solution.

# Works Cited

1. Computer Desktop Encyclopedia. [Online] http://www.computerlanguage.com/.

2. Paravirtualization explained. Search Server Virtualization. [Online] TechTarget, 2007. http://searchservervirtualization.techtarget.com/tip/Paravirtualization-explained.

3. Paravirtualization. Wikipedia. [Online] http://en.wikipedia.org/wiki/Paravirtualization.

4. Xen Wiki. [Online] http://wiki.xen.org.

5. Xen ARM Wiki. Xen ARM Wiki. [Online] Samsung Corp. http://wiki.xen.org/wiki/XenARM.

6. **Amit Aneja.** Designing Embedded Virtualization Intel Platform. s.l. : Intel Corporation, 2011.

7. **Intel Corporation.** Intel® Virtualization Technology for Directed I/O Architecture Description. s.l. : Intel Corporation, 2011.

8. Memory Allocation - Xen 3.0 Virtualization Interface Guide. Linuxtopia. [Online] http://www.linuxtopia.org/online_books/linux_virtualization/xen_3.0_interface_guide/linux_virualization_xen_interface_9.html.

9. Inter-Domain Communication - Xen 3.0 Virtualization Interface Guide. Linuxtopia. [Online] http://www.linuxtopia.org/online_books/linux_virtualization/xen_3.0_interface_guide/linux_virualization_xen_interface_52.html.

10. **Tomlinson, Allan and Gebhardt, Carl.** Challenges for Inter Virtual Machine Communication. Mathemetics, Royal Holloway, University of London. Surrey : s.n., 2010.

11. **Miguel Masmano, Ismael Ripoll, Alfons Crespo.** XtratuM Hypervisor for LEON3 Volume2: User Manual. s.l. : Universidad Politecnica de Valencia, February 2011.

12. **Zhou, Rui.** Partitioned System with XtratuM on PowerPC. Universi. s.l. : Universidad Politecnica de Valencia, 2009.

13. Open Kernel Labs Community Wiki. [Online] http://wiki.ok-labs.com/.

14. **Gernot Heiser.** Virtualization for Embedded Systems. s.l. : Open Kernel Labs, 2007.

15. Xen Web site. [Online] Citrix. http://wiki.xen.org.

16. Xen. Wikipedia. [Online] http://en.wikipedia.org/wiki/Xen.

17. GNU General Public License. [Online] http://www.gnu.org/licenses/gpl.html.

# Glossary

AMD – Advanced Micro Devices

ARINC – Avionics Application Standard Software Interface

ARM – Originally Acorn RISC (Reduced Instruction Set Computer) Machine

BIOS – Basic Input/Output System; the program that runs when the computer starts and loads the operating system

CPU – Central Processing Unit

DMA – Directory Memory Access

GNU – Gnu's Not Unix; a free open source UNIX-like operating system

GPL – General Public License; an open source license

Guest Operating System – An operating system that runs in a virtualized environment

Hyperthreading – INTE's technology to emulate a multi-processing environment on a single processor

Hypervisor – A virtual machine monitor program

I/O – Input/Output

IPC – Inter-Process Communication

LEON3 – a microprocessor based on the SPARC-V8 RISC architecture and instruction set

Microkernel – A reduced version of a regular operating system kernel

MMU – Memory Management Unit

Paravirtualization – A virtualization technology that replaces the device drivers with virtualization-aware device drivers

PCI – Peripheral Component Interface

PCI-Back – A device driver with a PCI pass -through interface to allow guests direct and exclusive hardware access to PCI devices

TCB – Trusted Computing Base

USB – Universal Serial Bus

Virtualization Extensions – Hardware support for virtualization, such as Intel VT-x and AMD-V

VM – Virtual Machine

VMM – Virtual Machine Monitor