A Practical Look at Micro-Kernels and Virtual Machine Monitors

François Armand, Michel Gien, *Member, IEEE*

Abstract — **In this paper, we look at two different approaches used to provide embedded system support for virtualization and virtual machine monitors for consumer electronics and mobile devices. We compare the micro-kernel approach, which has been a popular choice for building embedded operating systems with the Virtual Machine Monitor (VMM) or hypervisor approach widely deployed in general purpose computing environments such as desktops and data center servers. Comparison criteria are based on virtualization use cases that are typical of Consumer Electronics (CE) systems such as mobile devices and IPTV. These approaches are further evaluated based on performance and on their ability to allow re-use of existing (often real-time) software as well as modern open operating systems such as Linux while remaining as transparently as possible. Such transparency can come through different paths, including: leveraging of hardware virtualization support, minimal modifications to the original operating system internals (kernel, device drivers, etc.), and the ability to use existing operating system applications as-is and without the need to port them to a new environment. An analysis of the fundamental principles behind each approach is presented with a discussion of their impact on existing operating environments, together with practical performance results based on existing micro-kernels and real-time hypervisor benchmarks. We conclude that mapping the VMM (hypervisor) approach used in data centers to the needs of embedded systems is a better option for the support of complete operating systems (as guests) than extending micro-kernels for such functionality.**

Keywords—**Virtual machine monitors for CE systems such as mobile devices and IPTV, Virtualization for security and fault-isolation on CE systems, Embedded operating system support for virtualization, Virtualization architecture for real-time applications**

***Index Terms*—Embedded system, hypervisor, micro-kernel, Operating System (OS), virtualization, Virtual Machine Monitor (VMM).**

I. Introduction

After several years of availability in the enterprise and server space, virtualization is becoming a hot topic in embedded systems, particularly for consumer electronics systems such as mobile devices and IPTV set-top boxes. One of the main drivers is the need to run new or feature-rich open system software while maintaining existing legacy software that has been already tested and validated in their own operating environment. Such open software commonly includes Linux and more established operating systems such as Windows or Symbian where developers want to run the operating system unchanged while also extending their device security and manageability at all levels.

Co-existence of several operating environments on the same hardware platform is one of the main purposes of hardware virtualization software, made possible by the provision of a virtual image of the hardware to each operating system, which believes it is running alone on the underlying hardware. Mobile phone platform products that are now been deployed with such virtualized operating environments include the NXP 7210 platform used in the Purple Magic single core Linux reference phone design [1], [2].

Hardware virtualization is not a new concept. It has been used in data centers for years. However, its introduction in embedded systems, in particular mobile devices is relatively new. Several approaches are being used to provide hardware virtualization in embedded devices. One extends the data center Virtual Machine Monitor (VMM) or hypervisor approach to fit the constraints of embedded systems. Another approach extends micro-kernels that have traditionally been used in embedded systems so they can support full operating system kernels in addition to their own embedded applications.

This paper examines foundation concepts of each approach and discusses their implications on the expected benefits of virtualization in embedded devices. A parallel can be made with the arguments exchanged in two papers issued by stakeholders from each "camp", ie, Xen [3] and L4 [4].

Finally actual measurements and performance benchmarks of the L4 microkernel and the VirtualLogix VLX hypervisor illustrate the implications of the two approaches in real-world scenario.

II. Virtual Machine Monitors and Hypervisors

A. Virtualization in the Data Center

The common term, virtual machine monitor (VMM), refers to software that controls and virtualizes the system "physical" hardware and reflects that this software generates and monitors "virtual machines" in which operating systems (OS) are executed. An operating system that executes in the context of a virtual machine is often referred to as a "guest OS," as

Manuscript received October 10, 2008.

François Armand is with VirtualLogix, 6 avenue Gustave Eiffel, 78180 – Montigny-le-Bretonneux, France (phone: +33 1 39 44 74 00; fax: +33 1 30 57 00 66; (e-mail: Francois.Armand@VirtualLogix.com).

Michel Gien is with VirtualLogix, 6 avenue Gustave Eiffel, 78180 – Montigny-le-Bretonneux, France (phone: +33 1 39 44 74 22; fax: +33 1 30 57 00 66; e-mail: Michel.Gien@VirtualLogix.com).

opposed to a "native OS" that has full control over the hardware resources (i.e. the "real machine"). The VMM may sometimes be called a "hypervisor" by analogy to the term "supervisor" used to designate the function of the inner part of an OS called the OS kernel, which supervises the execution of applications and their access to hardware resources (computing, memory, storage, network, etc.). In the case of "virtualization technology," the hypervisor provides access to the hardware resources for the operating systems via fully or partially virtualized interfaces. This adds one layer of hardware management which multiplexes usage of the “real” hardware to several operating systems or several instances of the same operating system, which can thus share that same hardware.

The first hypervisor, called Control Program 67 (CP-67) [5], was introduced in the summer of 1966 by IBM in its famous System/360 family of mainframe computers, the 360/67. It then became VM/370 on the S/370 and was used to run various operating systems such as "single user" (e.g. CMS) or "batch" operating systems (e.g. OS 360) simultaneously and securely on the same computer. Essentially VM/370 and hardware cooperate so multiple instances of any operating system, each with protected access to the full instruction set, can peacefully and concurrently coexist. This is one of the more successful hypervisors and still sees wide use in IBM mainframes today.

Since then, hypervisors have gained increasing popularity in the computing space with many commercial and public options now available. The most widely used hypervisor on x86-based computers is VMware [6], which is offered by a subsidiary of the EMC Corporation. The University of Cambridge Computer Laboratory has developed a competing open source project called Xen [7] that is supported by various computer manufacturers including IBM, Sun Microsystems and HP to run several instances of Linux and other operating systems primarily on x86-based servers. Microsoft recently announced its own hypervisor software named Hyper-V [8].

Hypervisors provide "hardware virtual machines" and should not be confused with interpreters providing "application virtual machines." Interpreters isolate the application used by the user from the computer it is running on. Because versions of the virtual machine are written as interpreters for various computer platforms, any application written for the virtual machine can be operated on any of the platforms, instead of having to produce separate versions of the application for each computer and operating system. One of the best known examples of an application virtual machine is Sun Microsystem's Java Virtual Machine [9].

B. Virtualization in a Consumer Device

The amount of processing power and memory available in today’s mobile handsets enable them to run high-level, general purpose operating systems such as Linux, Windows or Symbian providing rich sets of services and applications. Smart phones increasingly resemble laptops in terms of features that they offer to users. Similar to what happened with personal computers in past years, today’s convergence of increased processing power and operating system support allows virtualization technology to be applied to mobile handsets and provide them with a software architecture that allows them to consolidate different operating environments on the same hardware platform.

Running along side of a feature-rich open operating system, a real-time operating system (RTOS) is used in mobile handsets to support deterministic tasks, such as running wireless protocol stacks and core phone services. Aside from the operating system choices, mobile handsets present a further challenge to virtualization solutions in their support for real-time operating systems and for many peripherals such as screen, touchpad, keyboard, audio I/O, camera, flash memory, disk, Bluetooth, USB, WiFi as well as wireless telecom network protocols.

Emerging from these market and technical requirements is a new generation of hypervisors, sometimes referred to as using "real-time virtualization" technology targeting embedded devices by allowing a guest OS of the hypervisor to be a real-time operating system (RTOS) without compromising its native real-time characteristics and behavior. VirtualLogix VLX is an example of such a “real-time virtualization” hypervisor [10].

Like other hypervisors in the traditional computing space, real-time hypervisors ensure strict isolation and secure communication between multiple virtualized execution environments. In addition, real-time hypervisors support the ARM processor family powering the vast majority of mobile phones and when it is made available by the silicon manufacturer, these hypervisors are able to leverage hardware support for virtualization, on Intel VT-based architectures for example.

"Real-time virtualization" can be used to securely consolidate a RTOS and a rich operating system – which would otherwise run on independent processors – on a single CPU lowering the cost of smart phones and bringing more functionality into the high-volume mass market. It can also be used to support a trusted execution environment (TEE), compliant with the requirement standards of the Open Mobile Terminal Platform (OMTP) industry organization [11].

Unlike virtualization technologies used in IT servers in the data center, such as provided by VMware, Xen, or Microsoft, VirtualLogix VLX has been specifically designed to address the requirements of devices such as mobile handsets, by meeting stringent cost, performance and memory constraints. Such hypervisor-based virtualization products can be summarized as providing real-time deterministic execution environments dedicating specific hardware resources to particular guest OS's for performance reasons, and supporting embedded processors such as the ARM and low power Intel Atom processor families.

They support Linux, Windows, Symbian, several commercial real-time operating systems and a variety of legacy "home-grown" proprietary embedded OS's. Their real-time virtualization technology provides a flexible architecture that enables various tradeoffs between resource isolation and sharing policies, based upon use case requirements.

III. MICRO-KERNELS

A. Operating system complexity

Complexity of operating systems has been increasing in a rather exponential way during the last 20 years. Using a measure as simple as the number of lines of code, Unix and Linux systems have evolved from 10 000 of lines of C code (Unix V7, in the early 80's) to more than 8.5 million lines of code in the Linux 2.6.24 kernel[1] [12]. Services provided by an operating system have extended and cover a much larger scope than 20 years ago. Peripheral devices are much more diverse and complex than they used to be and operating systems needs to abstract an ever larger set of services to their applications (eg, audio, video, input devices, sensor driven services).

Systems designers have attempted to address operating system internal complexity issues using various modularization approaches. Micro-kernels, exo-kernels and other similar approaches have been studied and tested to try to fragment and reduce the overall complexity of systems with the hope that managing smaller and less complex pieces would lead to an overall simpler design.

In the late 80's, the Mach micro-kernel [13] developed at CMU (Carnegie Mellon University) had a similar approach to reduce operating system complexity although primarily towards data center operating systems. A derivative is still used by Apple as the underpinning of MacOS X.

Between 1980 and 2001, the Chorus micro-kernel [13][14] has gone thru five major iterations, starting as a research project at INRIA (Institut National de Recherche en Informatique et Automatique in France) then evolving as a software product by Chorus Systems then Sun Microsystems. It has been used to re-architect the UNIX operating system as a set of cooperating system services on top of a minimum micro-kernel, as well as to combine real-time tasks with general purpose UNIX applications in consumer devices such as Java terminals and network infrastructure equipment. The experience gained during the years of applying micro-kernel technology to combine several operating environments on the same hardware led the Chorus Systems engineering team to fund VirtualLogix in 2002. In doing so, they took a different approach and designed native hardware virtualization software that allows re-use of existing operating systems with minimum changes instead of attempting to forcefully merge monolithic operating system kernel functions along microkernel services.

QNX [15] evolved from an embedded real-time kernel to a micro-kernel based operating system supporting combinations of real-time tasks and POSIX processes. It is now used primarily in the automotive industry and some industrial systems.

More recently several academic research communities started to develop a new generation of micro-kernels based along the L4 concept and APIs originally expressed in the mid 90's [16]. Several variants are still being developed in academic research [17] and in industry [18].

[1] Today, 10 000 lines of code is the difference between the 2.6.23 and 2.6.24 releases of the Linux kernel.

B. Micro-kernel goals and services

Micro-kernels share the common goal of keeping things as simple as possible in the micro-kernel, implementing low level mechanisms only, and isolating operating system servers in user mode. Inter-process communication (IPC) is usually provided at the lowest level possible and is widely used as a general communication paradigm between operating system servers and between servers and the micro-kernel, and as a substitute to system traps.

C. Drawbacks

Due to its basic and low-level nature, the micro-kernel API itself is not suitable to run meaningful applications. Therefore it must be extended with higher level APIs necessary to run real-time tasks or UNIX-like applications.

For real-time tasks, micro-kernels frequently introduce new proprietary interfaces and services, which add to the already plethoric set of embedded and real-time operating systems in the market.

For general purpose applications, POSIX is one of the commonly adopted standards. However, POSIX interfaces need to be implemented by means of operating system servers running on top of the micro-kernel and using its low-level APIs and services. This approach requires significant re-architecting of operating system kernel components so they run in different independent system servers communicating by means of the underlying micro-kernel IPC. The result diverges significantly from already existing and widely adopted operating system implementations such as Linux and is therefore difficult to maintain and evolve. This probably explains why all projects to re-architect UNIX along a micro-kernel approach failed so far, both in the industry (Novell's UNIX System V/MK based on Chorus, or OSF/1 based on Mach) and thus far in the open source community, (GNU/Hurd).

These failures are due to the fact that the multi-layer operating system architecture proposed by micro-kernels, although modular and elegant, adds significant complexity to system designers who need to provide standard APIs to their application developers, compared to already existing and widely adopted monolithic implementations, such as existing real-time operating systems or open systems such as Linux, Window or Symbian. Performance can be impacted as well due to multiple interactions between kernel components running at different levels and communicating by means of the underlying micro-kernel IPC.

IV. HARDWARE VIRTUALIZATION REQUIREMENTS

The goal of hardware virtualization is to run several existing operating environments on a single underlying physical hardware platform. Virtualization requirements that should be satisfied when applied to embedded system products may be summarized as follows:

- Run an existing operating system and its supported applications in a virtualized environment, such that modifications required to the operating system are minimized (ideally none), and performance

overhead is as low as possible;
- It should be straightforward to move from one version of an operating system to another one; this is especially important to keep up with frequent Linux evolutions;
- Reuse native device drivers from their existing execution environments with no modifications;
- Support existing legacy often real-time operating systems and their applications while guaranteeing their deterministic real-time behavior.

While consumer electronic devices leverage performance critical software on an RTOS and open operating systems such as Linux, there is usually little need for a tight integration between applications running on each environment. Traditionally, these functions are distributed between different processors and associated hardware. In today's devices where additional hardware is replaced with software virtualization, the existence of virtualization is not expected to impact the system interactions even though they run on the same piece of physical hardware. In some cases however, sharing of peripheral devices between execution environments can be required. Typical examples in mobile devices include audio peripherals, power management and modem access.

V. VIRTUALIZATION APPROACHES

A. Transparent and para-virtualization

There are basically two ways to provide transparent virtualization, ie run multiple operating systems simultaneously on the same processor with no modifications:

- Dynamic binary translation where a hypervisor dynamically rewrites some part of the guest OS,
- Hardware assisted virtualization as supported by the latest x86 processors from Intel and AMD.

The first approach, as used by VMware requires a large hypervisor with high overhead. Although suited for a wide number of uses in the data center, this architecture introduces non deterministic caching mechanisms which make it unsuitable for embedded systems products.

The second approach, hardware assisted virtualization, is used today for deployment within network infrastructure equipment, when combined with a real-time hypervisor. However, even though most embedded processor manufacturers or licensors plan to provide hardware assisted virtualization features as part of their technology roadmap, these new processor technologies are not yet widely deployed in the embedded space.

In order to minimize the performance impact on operating system performance, "para-virtualization" techniques have been introduced [19] that require modifications in the operating system kernel so that it calls the underlying virtualization software instead of relying on complete emulation of the hardware. In order to preserve as much transparency as possible to the guest operating system, such modifications can be limited to its Hardware Abstraction Layer (HAL) or Board Support Package (BSP) – hence these para-virtualization changes are no more than what is required to adapt an OS to a new hardware board.

B. Hosted versus native virtualization

Although one can think of an operating system as a virtualization of the hardware, its primary purpose is to support a higher level of abstraction to facilitate the operation of applications. Basic hardware features such as processor(s), physical memory, disks, networks and other peripheral devices are abstracted by the operating system into processes or tasks, virtual memory, files, inter-process communication and the like.

These abstractions could be used in turn to provide an emulation of the physical hardware on which on can run a complete operating system kernel. This technique of providing hardware virtualization is referred to as "hosted" virtualization software (or Type II Virtual Machines). This is a convenient approach but it comes at the cost of high memory footprint and relatively low performance because the system overhead compared to native operating systems is quite high.

Traditionally a host is a full operating system such as UNIX, Linux, Windows. With a micro-kernel based operating system, on might think of using the micro-kernel services to provide the hardware abstractions necessary to support virtual machines. However, as we will detail later in this paper, the level of services provided by the micro-kernel are at the same time too high-level (tasks or threads, inter-process communication) and insufficient (memory management, peripheral devices) requiring additional system services. The result of implementing virtualization on a micro-kernel host is a level of complexity similar to fully hosted VMs with high performance impact.

Hypervisors on the other hand providing "native" hardware virtualization (or Type I Virtual Machines) have been designed to emulate the low level features of the hardware that are expected by an operating system kernel, so they can be run into virtual machines with no changes.

VI. MICRO-KERNELS VERSUS HYPERVISORS

Para-virtualization techniques have been applied to both "hosted" and "native" approaches to virtualization. However, the level of modification required in guest operating systems depends on how closely the virtualization software abstractions model the native hardware. Higher level abstractions require more intrusive modifications. In that sense, micro-kernels and hypervisor approaches have a quite different impact on the amount and difficulty of modifications required to run a guest operating system.

Hypervisors typically know only about virtual machines and guest operating systems but tend to ignore all of the internal details of the workload being run in a virtual machine. Hence, an hypervisor knows only the properties of virtual machines it manages (memory, devices, scheduling). Such properties are quite close to those seen by a native operating system running on bare hardware.

On the other hand, micro-kernels provide higher level abstractions, such as tasks, threads, memory contexts and IPC mechanisms which are similar to those already implemented

by a complete operating system kernel, thus involving a more complex mapping.

TABLE I
OS PARA-VIRTUALIZATION ON MICRO-KERNEL VERSUS HYPERVISOR

Services	Micro-kernel	Hypervisor
Threads	Knows all threads of any OS and imposes global scheduling	Knows and schedule only guest OS
Processes	Knows about tasks and OS processes	Knows only guest OS partitions. Hypervisor can easily start, stop, resume complete OS
Memory	Knows memory context of tasks and is Involved in memory context switch, page faults, etc.	Knows only memory partitions. Memory management is done by guest OS
Communication	IPC imposes API and semantics	IPC services left to guest OS
Interrupts & Exceptions	Catch and forward as IPC	Virtual Interrupts forwarded to guest OS. Exceptions managed by guestOS
Peripherals	Device drivers as microkernel processes	Core drivers in hypervisor, peripheral drivers in OS partitions

Hypervisor approach is less intrusive and closer to what a native operating system is expecting from physical hardware.

Table I below outlines some of the differences in para-virtualization of an operating systems kernel to a micro-kernel versus a hypervisor.

A. Tasks and Threads

Micro-kernels manage tasks and threads: they offer services to create, delete and list them. They schedule threads they have created. As a result, the typical way to run an OS kernel on top of a micro-kernel is to insure that all processes and threads managed by the guest OS are known to, and managed by the underlying micro-kernel. This requires heavy guest OS modifications to allow this. Creation and deletion of such objects in the guest OS need to rely on a corresponding invocation of the micro-kernel. For mostly static embedded systems, such overhead could be acceptable as it would be incurred only at initialization time. However, a thread (task) is now described by two data structures, one in the guest OS and one in the micro-kernel, which introduces memory footprint overhead.

Hypervisors on the other hand, do not know about tasks and threads managed by their guest OSes. This independence avoids overhead in duplicate task creation and deletion operations or for managing such objects.

B. Scheduling

Micro-kernels provide services to schedule threads according to micro-kernel scheduling policies that replace the existing guest OS processor scheduling policies. The guest OS must be heavily modified to allow this. When more than one OS is supported, threads from different OSes are scheduled by the micro-kernel independently to the OS they belong to thus creating complex situations to analyze and debug.

Since hypervisors schedule only guest OS, each one keeps its own scheduling policy, which allow running an OpenOS with a time share policy and a real-time OS with a fixed priority scheduling policy with no pain.

C. Memory

Micro-kernels (or dedicated servers) manage physical and virtual memory for the tasks they support – whether these are native micro-kernel tasks or guest OS tasks. Each task context is known by the micro-kernel. The micro-kernel is involved in every memory context switch and the guest OS must be heavily modified to allow this. Memory and page faults are also captured by the micro-kernel and typically forwarded to the OS by means of the micro-kernel IPC.

Hypervisors know physical memory partition granted to each virtual machine (guest OS) and check the validity of memory mappings used by guest OS. Memory and page faults handler provided by guest OSes are triggered directly by that guest OS without involving an additional generic IPC mechanism.

D. Communications

Within a micro-kernel, there is no other way to communicate between threads running in different tasks of a guest OS than to use the microkernel IPC.

Hypervisors on the other hand enable communication between guest OSes by means of very low level mechanisms such as shared memory used to implement ring buffers to exchange data and virtual cross-interrupt to signal that some data is available for consumption. Such mechanisms are not intended to be used by applications but by OS kernels and device drivers that can map their existing higher level communication services and protocols with minimum impact.

E. Device Drivers

Micro-kernels typically catch peripheral interrupts and forward them using the micro-kernel IPC. Sharing of devices between multiple guest OS also relies on the micro-kernel IPC.

Hypervisors for embedded systems enable guest OSes to run with their native device drivers. Interrupts are transparently virtualized and delivered as they are expected and used by a native OS. In addition, some hypervisors provide a standard framework to share peripheral devices between several guest OS [20].

F. Virtual Machine Management

Micro-kernels abstractions do not have concepts of a unique "guest OS", "virtual machine" or "memory partition". Hence, there is no straight-forward way to perform management operations such as "stop" or "restart" on guest OSes or virtual machines. Such operations could be implemented as a service on top of the micro-kernel, but this would add extra performance and memory overhead. Similarly, accounting for or limiting how much memory or CPU time is used by a given guest OS may be quite complex to implement.

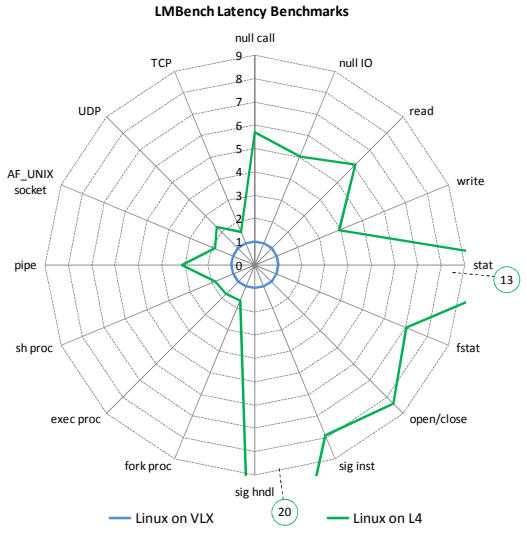
VII. MEASUREMENTS

One of the measures of the impact of a micro-kernel versus hypervisor approach to operating system kernel par-

virtualization is to look at the number of files that need to be modified in order to support a guest operating system kernel. Here, due to the availability of Linux on both the VLX hypervisor and the L4 microkernel, we provide some comparative source code modifications information.

On an ARM926 based platform, the Linux 2.6.13 kernel includes 1200 files and 30 Board Support Packages (BSP) in the `arch/arm` and `include/asm-arm` directory trees. In order to make it run on the VLX MH 3.1 hypervisor, 51 files are modified and 40 files added in the ARM architecture tree. By contrast, the corresponding L4 release 1.6.4 implementation requires the introduction of a new architecture tree (`arch/L4`) which includes 202 new files, 108 of which are ARM specific files. This new `arch/L4` source tree requires that changes to the ARM Linux tree be migrated to the `arch/L4` tree in order to leverage BSPs, ARM processor enhanced features, and device driver support.

On the performance side, the following diagram illustrates the overhead of typical LMbench performance tests run on Linux 2.6.23 with L4RootFS compiled with gcc 3.4.4 on a Freescale iMX31 ADS board (ARM1136 at 532 MHz with 128MB of RAM) on the VLX MH 3.2 hypervisor and on the OKL4 release 2.1 "pistachio" (April 15, 2008) micro-kernel respectively. The diagram shows the overhead factor of LMBench latency benchmark results obtained on Linux on L4 compared to the same tests results on Linux on VLX. The L4 overhead compared to VLX ranges from a 2x to 20x factor depending on the Linux system call benchmark.



VIII. Conclusion

This paper has outlined different approaches to introducing virtualization technology into consumer electronics systems and mobile devices, with a key benefit being to support several existing execution environments on the same hardware platform, with no or minimum changes to the existing software.

Although hardware virtualization software as provided by hypervisors or virtual machine monitors (VMMs) have been around for a number of years in general purpose computing environments, its introduction into embedded systems is quite new. Light weight hypervisors such as provided by VirtualLogix VLX have been designed to provide "real-time virtualization" that fits the needs of such market.

At the same time, virtualization has become a "buzz word" that is being used to "sell" micro-kernel technology, which has been designed from the outset as a better way to architect modern operating systems, and not as a way to share native hardware between existing operating systems as transparently as possible.

We have shown that, although the hypervisor and micro-kernel approaches may seem to address similar concerns, their design centers are quite opposite. Micro-kernels introduce virtualization as an extension to operating systems, whereby hypervisors consider virtualization as an extension to the hardware, taking advantage of hardware assistance when available on modern processor architectures.

Performance benchmarks also show that the impact of micro-kernel based virtualization on native operating system performance is bigger than with a hypervisor in addition to being more intrusive to operating system kernels when para-virtualization techniques are being used.

In conclusion, micro-kernel operating systems should be used when one can start from a white sheet of paper and introduce a new operating system into their devices that is able to run applications as different as real-time tasks and general purpose Linux processes. On the other hand, a hypervisor that is providing native hardware abstractions that allows a close to native existing operating system to run should be used when one wants to re-use legacy software or extend the capabilities of devices based on established open operating systems such as Linux, Windows or Symbian OS.

Acknowledgment

The authors would like to thank the whole VirtualLogix team and in particular the systems architects who put into question their years of experience developing micro-kernel based operating systems and helped shape the design of VirtualLogix VLX to better address embedded systems needs.

References

[1] NXP, "Nexperia Cellular System Solutions for Linux." Available: http://www.nxp.com/acrobat_download/literature/9397/75016263.pdf

[2] Purple Labs, "Purple Magic first sub-$100 3G Linux Reference Phone for Emerging Markets." Available: http://www.purplelabs.com/product-purple-magic.php

[3] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, D. Magenheimer, "Are Virtual Machine Monitors Microkernels Done Right?". In *Proc. 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, Sante Fe, NM, USA, June 2005.

[4] G. Heiser, V. Uhlig, J. LeVasseur, "Are Virtual Machine Monitors Microkernels Done Right?" NICTA Tech. Rep. PA005103, Sep. 2005.

[5] CP 67, http://en.wikipedia.org/wiki/IBM_System/360_Model_67

[6] VMware. Available: http://www.vmware.com

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, pp. 164–177, Oct. 2003.

[8] Microsoft, "Virtualization and Consolidation with Hyper-V." Available: http://www.microsoft.com/windowsserver2008/en/us/virtualization-consolidation.aspx

[9] T. Lindholm, F; Yellin, "The Java Virtual Machine Specification," Sun Microsystem, 1999. Available: http://java.sun.com/docs/books/jvms/
[10] VirtualLogix VLX. Available: http://www.virtuallogix.com
[11] OMTP, "OMTP Advanced Trusted Environment: OMTP TR1," May 2008.Available:http://www.omtp.org/Publications/Display.aspx?Id=dbe5b79c-216c-4ce6-a0f7-c482ac58f44c#
[12] G. Kroah-Hartman, J. Corbet, A. McPherson, "Linux Kernel Development.Available:http://www.linuxfoundation.org/publications/linuxkerneldevelopment.php
[13] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for UNIX development." In *Proc. Summer USENIX Conference*, June 1986.
[14] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser, "Chorus distributed operating systems", *USENIX Computing Systems*, 1(4), pp. 305–367, 1988.
[15] D. Hildebrand, "An architectural overview of QNX." In *Proc. of the USENIX Workshop on Microkernels and other Kernel Architectures*, Seattle, WA, USA, pp. 113–126, Apr. 1992.
[16] J. Liedtke, "Towards real microkernels," *Communications of the ACM*, 39(9), pp. 70–77, Sep. 1996.
[17] G. Heiser, K. Elphinstone, I. Kuz, G., Klein, S. M. Petters, "Towards Trustworthy Computing Systems: Taking Microkernels to the Next Level," *SIGOPS Operating Systems Review*, July 2007.
[18] R. Kaiser, S. Wagner, "The PikeOS Concept: History and Design," SysGO AG White Paper. Available: http://www.sysgo.com
[19] A. Whitaker, M. Shaw, S. D. Gribble, "Denali: Lightweight Virtual Machines for Distributed and Networked Applications," In *Proc. of the USENIX Annual Technical Conference*, 2002.
[20] F. Armand, M. Gien, G. Maigné, G. Mardinian, "Shared Device Driver Model for Virtualized Mobile Handsets," *In Proc. ACM MobiVirt Workshop held in conjunction with MobiSys 2008,* Brekenridge, CO, USA, June 17, 2008.