PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
COMPUTER SCIENCE GRADUATE PROGRAM

A LIGHTWEIGHT VIRTUALIZATION LAYER WITH HARDWARE-ASSISTANCE FOR EMBEDDED SYSTEMS

CARLOS ROBERTO MORATELLI

Dissertation submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fullfillment of the requirements for the degree of Ph. D. in Computer Science.

Advisor: Prof. Fabiano Hessel

Porto Alegre
2016

Ficha Catalográfica

M831L Moratelli, Carlos Roberto

A lightweight virtualization layer with hardware-assistance for embedded systems / Carlos Roberto Moratelli . – 2016.
155 f.
Tese (Doutorado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Fabiano Hessel.

1. Virtualization. 2. Embedded Systems. 3. Hypervisor. 4. Real-time. I. Hessel, Fabiano. II. Título.

TERMO DE APRESENTAÇÃO DE TESE DE DOUTORADO

Tese intitulada "*A Lightweight Virtualization Layer with Hardware-Assistance for Embedded Systems*" apresentada por Carlos Roberto Moratelli como parte dos requisitos para obtenção do grau de Doutor em Ciência da Computação, aprovada em 22 de março de 2016 pela Comissão Examinadora:

[illegible]

Prof. Dr. Fabiano Passuelo Hessel– Orientador — PPGCC/PUCRS

[illegible]

Prof. Dr. Tiago Coelho Ferreto – PPGCC/PUCRS

[illegible]

Prof. Dr. Rômulo Silva de Oliveira - UFSC

[illegible]

Prof. Dr. Rodolfo Jardim de Azevedo - UNICAMP

Homologada em 06/10/2016, conforme Ata No. 020 pela Comissão Coordenadora.

[illegible]

Prof. Dr. Luiz Gustavo Leão Fernandes
Coordenador.

To my family and friends.

"I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones."
(Linus Torvalds)

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to those who helped me throughout all my Ph.D. years and made this dissertation possible. First of all, I would like to thank my advisor, Prof. Fabiano Passuelo Hessel, who has given me the opportunity to undertake a Ph.D. and provided me invaluable guidance and support in my Ph.D. and in my academic life in general. Thank you to all the Ph.D. committee members – Prof. Carlos Eduardo Pereira (dissertation proposal), Prof. Rodolfo Jardim de Azevedo, Prof. Rômulo Silva de Oliveira and Prof. Tiago Ferreto - for the time invested and for the valuable feedback provided.Thank you Dr. Luca Carloni and the other SLD team members at the Columbia University in the City of New York for receiving me and giving me the opportunity to work with them during my ‘sandwich’ research internship.

Eu gostaria de agraceder minha esposa, Ana Claudia, por ter estado ao meu lado durante todo o meu período de doutorado. O seu apoio e compreensão foram e continuam sendo muito importantes para mim. Obrigado pela paciência, amor e carinho durante todos esses anos. Também gostaria de agradecer aos meus pais e irmão pelo apoio durante toda a minha vida. Agradeço aos colegas e amigos do laboratório GSE/PUCRS, especialmente ao amigo Sergio Johann Filho pelas importantes discusões durante a execução do trabalho.

UMA CAMADA LEVE DE VIRTUALIZAÇÃO ASSISTIDA POR HARDWARE PARA SISTEMAS EMBARCADOS

RESUMO

O poder de processamento presente nos sistemas embarcados modernos permite a adoção de técnicas de virtualização. Juntamente com os ganhos em redução de custo e melhor utilização dos recursos, como por exemplo uma melhor utilização do processador, a virtualização possibilita a co-execução de diferentes sistemas operacionais em um processador, sejam eles sistemas operacionais de tempo real (RTOS) e/ou de propósito geral (GPOS). A implementação da técnica de virtualização esta baseada em um módulo de software denominado hypervisor.

Devido a complexidade de se desenvolver uma nova camada de virtualização especialmente projetada para sistemas embarcados, muitos autores propuseram modificações em sistemas de virtualização que são largamente empregados em servidores na nuvem para melhor adapta-los às necessidades dos sistemas embarcados. Contudo, a utilização de memória e os requisitos temporais de alguns dispositivos embarcados requerem abordagens diferentes daquelas utilizadas em servidores. Além disso, a atual tendência de utilização de virtualização nos dispositivos projetados para a internet das coisas (do inglês Internet of Things - IoT) aumentou o desafio por hypervisors mais eficientes, em termos de memória e processamento. Estes fatores motivaram o surgimento de diversos hypervisors especialmente projetados para atender os requisitos dos atuais sistemas embarcados.

Nesta tese, investigou-se como a virtualização embarcada pode ser melhorada a partir de seu estado atual de desenvolvimento para atender as necessidades dos sistemas embarcados atuais. Como resultado, propõe-se um modelo de virtualização capaz de agregar os diferentes aspectos exigidos pelos sistemas embarcados. O modelo combina virtualização completa e para-virtualização em uma camada de virtualização híbrida, além da utilização de virtualização assistida por hardware. Uma implementação baseada neste modelo é apresentada e avaliada. Os resultados mostram que o hypervisor resultante possui requisitos de memória compatíveis com os dipositivos projetados para IoT. Ainda, GPOSs and RTOS podem ser executados mantendo-se o isolamento temporal entre eles e com o baixo impacto no desempenho.

Palavras-Chave: Virtualização, Sistemas Embarcados, Hypervisor, Tempo-real.

A LIGHTWEIGHT VIRTUALIZATION LAYER WITH HARDWARE-ASSISTANCE FOR EMBEDDED SYSTEMS

ABSTRACT

The current processing power of modern embedded systems enable the adoption of virtualization techniques. In addition to the direct relationship with cost reduction and better resource utilization, virtualization permits the integration of real-time operating systems (RTOS) and general-purpose operating systems (GPOS) on the same hardware system. The resulting system may inherit deterministic time response from the RTOS and a large software base from the GPOS. However, the hypervisor must be carefully designed.

Due to the complexity of developing a virtualization layer designed specially for embedded systems from scratch, many authors have proposed modifications of the widely used server virtualization software to better adapt it to the particular needs of embedded system. However, footprint and temporal requisites of some embedded devices require different approaches than those used in server farms. Also, currently virtualization is being adapted for the field of the Internet of Things (IoT), which has increased the challenge for more efficient hypervisors. Thus, a generation of hypervisors focused on the needs of embedded systems have emerged .

This dissertation investigated how embedded virtualization can be improved, starting from the current stage of its development. As a result, it is proposed a virtualization model to aggregate different aspects required by embedded systems. The model combines full and para-virtualization in a hybrid virtualization layer. In addition, it explores the newer features of embedded processors that have recently adopted hardware-assisted virtualization. A hypervisor implementation based on this model is presented and evaluated. The results show that the implemented hypervisor has memory requirements compatible with devices designed for IoT. Moreover, general-purpose operating systems and real-time tasks can be combined while keeping them temporally isolated. Finally, the overall virtualization overhead is for most part lower than in other embedded hypervisors.

Keywords: Virtualization, Embedded Systems, Hypervisor, Real-time.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ABI – Application Binary Interface

ADPCM – Adaptive Differential Pulse-code Modulation

ADU – Application Domain Unit

API – Application Programming Interface

BEC – Best Execution Case

BE-VCPU – Best-effort Virtual Central Processing Unit

BVT – Borrowed Virtual Time

CBS – Constant Bandwidth Server

CE – Costumer Electronic

CPU – Central Processing Unit

CP0 – Coprocessor 0

CSA – Compositional Scheduling Architecture

DMA – Dynamic Memory Access

DPR – Dynamic Partial Reconfiguration

DRAM – Dynamic Random-Access Memory

EDF – Earliest Deadline First

ES – Embedded System

FP – Fixed Priority

FV – Full-Virtualization

GCP0 – Guest Cooprocessor 0

GPOS – General Purpose Operating System

GSE – Embedded Systems Group

GSM – Global System for Mobile Communications

GuestCtl0 – Guest Control Register 0

HAL – Hardware Abstraction Layer

HLL – High Level Languages

HVM – Hardware-assisted Virtualization

IASim	MIPS Instruction Accurate Simulator
IoT	Internet of Things
IPA	Intermediate Physical Address
ISA	Instruction Set Architecture
I/O	Input/Output
IOMMU	Input–output Memory Management Unit
JVM	Java Virtual Machine
NVRAM	Non-Volatile Random Access Memory
OS	Operating System
OVP	Open Virtual Platforms
PA	Physical Address
PID	Processor Identification
PV	Para-Virtualization
RTM	Real-time Manager
RTOS	Real-time Operating System
RT-VCPU	Real-time Virtual Central Processing Unit
SEDF	Simple Earliest Deadline First
USB	Universal Serial Bus
VA	Virtual Address
VCPU	Virtual Central Processing Unit
VHH	Virtual Hellfire Hypervisor
VM	Virtual Machine
VMM	Virtual Machine Manager
WEC	Worst Execution Case

CONTENTS

1. INTRODUCTION

Embedded systems are being widely adopted in all areas of human activity. Their increased performance during recent years has enabled a wide range of new applications. However, more processing power implies more functionalities and increased software complexity, which directly impacts the design constraints and goals of embedded systems [85]. For example, the possibility for the final user to develop and download new applications, before exclusively found in general-purpose systems, can now be found in many modern embedded devices [31]. This scenario has forced designers and companies to adopt new strategies, such as increased use of software layers allowing for more flexible platforms capable of meeting timing, energy consumption, and time-to-market constraints. Although embedded systems are assuming many of the features of general-purpose systems, some differences remain. They present critical real-time constraints and frequently have resource constraints such as, limited battery, memory and processing power.

The appearance of server farms in the 1990s and the increasing performance of the computer systems stimulated the adoption of virtualization. Virtualization allows for the possibility of decoupling the one-to-one correspondence between hardware and software. Thus, keeping several different operating systems on the same computer system, can drastically reduce the maintenance and energy costs [54]. Different operating systems reside in a shared memory and are combined on a single or multi-core processor through a software stack called a virtual machine monitor or hypervisor. In recent years, virtualization technology has quickly moved towards embedded systems motivated by the increasing processing power of the embedded processors and the increasing challenges to comply with their requirements. Although server virtualization is a well-known and mature technology widely applied for commercial use, embedded system virtualization is still being studied and developed. The main restriction to broader use of virtualization for embedded systems is that their requirements differ from server and enterprise systems [65]. Most notably, timing constraints and limited resources, like CPU and memory, are the main concerns around embedded systems virtualization. Thus, virtualization as deployed in the enterprise environment cannot be directly used in embedded systems.

There are many studies proposing different techniques for embedded system virtualization. A common approach is to adapt hypervisors widely used in server virtualization to embedded systems. The main concern about general purpose hypervisors is the absence of suitable support for real-time and memory requirements incompatible with some embedded devices. Despite these limitations, researchers are continually working to improve the embedded support of open-source hypervisors for server virtualization, like XEN and KVM, as seen in [22], [92], [89] and [8]. On the other hand, the distinguished characteristics of the embedded systems have motivated the appearance of hypervisors specially designed for embedded virtualization. Among the goals for embedded hypervisors development, two of them are frequently addressed: to keep low memory requirements and some level of support for real-time applications. Additionally, the diversity of embedded systems and their applications encourages the development of many different embedded hypervisors, like SPUMONE [39], Xtratum [80], M-Hypervisor [93], Xvisor [59] and the L4 hypervisors family [86], [32], [38], [88].

Intel and AMD manufactures provided hardware support for virtualization on the x86 architecture to address the continuous adoption of server virtualization. This was aimed to simplify the virtualization software layer and to improve performance. Before hardware-assisted virtualization, hypervisors needed to implement a technique called para-virtualization that consists of modifying the operating system to be virtualized. These operating system modifications break the software compatibility and require additional engineering work along with licence and source-code access for proprietary software. With dedicated hardware for virtualization, the hypervisors could implement a technique called full-virtualization where no modifications are required on the target operating system. Thus, making it easier to support new operating systems, especially proprietary ones. Similar to the x86 architecture, the late adoption of hardware-assisted virtualization for embedded processors made para-virtualization rule the development of embedded virtualization. Nowadays, the main embedded processor manufacturers had already designed virtualization extensions for their processor families, like PowerPC [26], ARM [6] and MIPS [36]. These gave hypervisor developers more design choices. Modern hardware-assisted virtualization now makes it possible to virtualize an operating system without any modification and keeps performance close to the level of non-virtualized systems. This is important to support legacy software. However, advanced hypervisor features like communication among virtual machines or shared devices may still require para-virtualization technique.

This dissertation starts by addressing the different embedded virtualization approaches and discussing their drawbacks and advantages. It points to several features required for embedded virtualization that the current hypervisor approaches fail to support. Thus, it is proposed the hybrid virtualization model concept that combines full and para-virtualization in the same hypervisor. To support full-virtualization, the model considers the recent hardware-assisted virtualization adopted in some embedded processors. Para-virtualization is used for extended services, such as communication among virtual machines, and real-time services. A hypervisor implementation based on the proposed model was developed for the MIPS M5150 processor and evaluated in the SEAD-3 development board. The contribution of this Ph.D. research is to show how design choices based on hardware-assisted virtualization associated with embedded system characteristics result in a simpler hypervisor, while still improving overall performance of the whole system.

1.1 Hypothesis and Research Questions

The aim of this work is to investigate the hypothesis that is possible to have an embedded lightweight hypervisor capable of supporting the major embedded virtualization requisites. Moreover, if considered that this hypothesis can be true, how important is the new hardware-assisted virtualization to accomplish it? Fundamental research questions associated with the hypothesis that guided this research are defined as follows:

1. What is the state-of-the-art in embedded virtualization and how does it accomplish the needs of embedded systems? This research question's main objective is to study embedded virtualization in detail and identify where the current virtualization models fail to support virtualization for embedded systems. Answering this research question will make it possible to understand the approaches that have already been tested, identify their limitations, and point out opportunities for improvements.

2. Can a virtualization model embrace the major embedded systems' needs and what features does it need to support to accomplish this goal? The objective of this research question is to determine the required embedded virtualization features and to propose a virtualization model capable of supporting these features. It is important to understand how different features, some of them contradictory, can be combined in the same model. The resulting virtualization model will guide the development of a new embedded hypervisor.

3. Can an implementation from a virtualization model be trustworthy and how can hardware-assisted virtualization be used to achieve this goal? This research question addresses whether a practical implementation can be constructed from the proposed virtualization model. Theoretical models can be hard to implement or an implementation may not achieve the expected results.

1.2 Organization of This Dissertation

The remainder of this dissertation is organized as follows:

- **Chapter 2** explains the basic concepts for the best understanding of the remainder of the text. It starts by defining embedded systems and their applications. Virtualization is defined along with a taxonomy of virtual machines. The different types of hypervisors are shown and the requirements for virtualizing a processor are explained. The technologies that make virtualization possible are also discussed. Moreover, two critical problems resulting from virtualization usage are presented. Then, the relationship between virtualization and embedded systems is explained. Finally, it discusses the past, present and future possibilities for virtualization in embedded systems.
- **Chapter 3** describes the state-of-the-art in embedded virtualization. The chapter starts by showing hypervisors designed for server virtualization. Then it presents several different authors' proposals to adapt these hypervisors for embedded systems. Subsequently, different hypervisors developed specially for embedded systems are described. A comparative study about the hypervisors is then presented. Finally, a Linux approach to improve real-time responsiveness is explained and compared to virtualization. Chapter 3 addresses research question (1).
- **Chapter 4** presents the virtualization model proposed in this dissertation. First, it discusses the required features for a virtualization model for embedded systems. Then, the proposed embedded virtualization model is presented, and its different aspects are discussed. Finally, the model is compared to the Virtual Hellfire Hypervisor's model (VHH is a hypervisor previously developed in the Embedded Systems Group at PUC-RS). Chapter 4 addresses research question (2).

- **Chapter 5** describes the hypervisor implementation based on the virtualization model proposed. First, the processor supported by the hypervisor is briefly presented. The software architecture is discussed and the CPU virtualization strategy is explained. Then the memory virtualization approach is exposed and the strategy for interrupt handling is described. The mechanism to allow communication among virtual machines is explained. Finally, the engineering effort to support a new guest and the implementation restrictions are discussed. Chapter 5 partially addresses research question (3).

- **Chapter 6** shows the hypervisor evaluation and practical results. The hypervisor memory requirements are determined and compared to other hypervisors. Next, the experience to port Linux and minor modifications to the Linux kernel to improve performance is described. The overall hypervisor overhead impact is then analyzed. The interrupt delivery strategy and the communication mechanism among virtual machines are also evaluated. Moreover, the real-time capabilities are measured. Lastly the overall results are discussed. Chapter 6 concludes research question (3).

- **Chapter 7** summarizes the dissertation and presents the concluding remarks. It restates the answers to the research questions, the main contributions, and presents possible directions for future work.

2. BACKGROUND IN EMBEDDED SYSTEMS AND VIRTUALIZATION

This chapter explains basic concepts to better understand the remainder of the text. Virtualization is a complex subject that involves several different topics and a wide range of solutions. This dissertation focuses on virtualization for the embedded systems. Thus, Section 2.1 starts by explaining the concept of embedded systems. If the reader is familiar with the definition of embedded systems, it is suggested that you read this chapter from Section 2.3. Section 2.2 introduces virtualization and its taxonomy, the different types of hypervisors and the requirements for virtualization. Section 2.3 explains different technologies that make virtualization possible. Section 2.4 describes two different problems caused by the use of virtualization. Section 2.5 shows the advantages of virtualization for embedded systems. Finally, Section 2.6 discusses the past, present and future of virtualization for embedded systems. If you are familiar with virtualization and embedded virtualization concepts, it is suggested to read this work from Chapter 3.

2.1 Embedded Systems

The use of microprocessors in equipment and consumer applications rather than laptop or personal computers (PCs) is wide spread. Actually, PCs are only one application of microprocessors. Embedded microprocessors are deeply integrated into everyday life, i.e., cars, microwaves oven, TVs, cell phones, and many other electronics are powered by microprocessors. Additionally, the increasing adoption of Internet of Things (IoT) [28] is accelerating the use of embedded applications. It is expected that there will be 24 billion interconnected devices by the year 2020 [28].

An embedded system (ES) is a computer system designed for a specific purpose, which distinguishes it from PCs or supercomputers. However, it is difficult to find a unique definition for ESs as they constantly evolve with advances in technology and as costs decrease. The following definition from [29] says: *An embedded system is a microprocessor-based system that is built to control a function or range of functions and is not designed to be programmed by the end user in the same way that a PC is.* Most embedded devices are designed for a specific function. However, this definition sounds outdated as modern ESs may allow the final user to develop and download new applications, e.g., smartphones and smartTVs. Additionally, another definition highlights their reliability [55]: *An ES is a computer system with higher quality and reliability requirements than other types of computer systems.* This definition covers systems such as avionics or medical equipment where a malfunction is life-threatening. Due to the wide range of ES purposes, there is not a single definition. Table 2.1 shows examples of the embedded system markets and devices.

Table 2.1: Example of embedded systems market and devices. Adapted from [55].

Market	**Embedded Device**
Automotive	Ignition system Air bag sensors Engine control
Consumer electronics	Smart phones Smart TVs Games Toys Laptops
Industrial control	Robotics
Medical	Cardiac monitors Dialysis machines
Networking	Routers Switches
Office automation	Printers Scanners

A recent trend in ESs is about their software complexity. Noergaard [55] defined ES software and application layers as optional. Thus, an ES could be composed of only a hardware layer. However, the performance improvement in functionalities, processing power and storage capacity over the last years has enabled a wide range of new complex applications and functionalities for ESs. Thus, the designers tend to adopt complex software layers to accomplish their goals. As mentioned early in this section, the possibility for the final user to develop and download new applications, before exclusively found in general-purpose systems, can now be found in many embedded devices. This has forced designers and companies to adopt new strategies, resulting in the increased use of software layers.

Although ESs are incorporating general-purpose features, many constraints remain. Because they are designed for specific functions, engineers can reduce the size and cost of the final product resulting in relatively simple and cheap devices. Thus, ES may suffer from hardware constraints, like reduced processing power, memory, weight and battery life. This added to the increasing complexity of embedded software is a challenge for designers that still need to meet timing, energy consumption, and time-to-market constraints for their products. In this scenario, virtualization can be a useful tool to deal with software complexity while increasing reuse, security and software quality.

2.2 What Means Virtualization?

Virtualization means the creation of an environment or, the creation of a virtual machine (VM), that acts like the real target hardware from the software or user point of view. The VM is implemented as a combination of real hardware and software aiming to execute applications [70]. Figure 2.1 depicts the taxonomy of VMs suggested by [70]. There are two kinds of VMs: process VMs and system VMs. Typically, a process VM is implemented on top of the operating system's (OS) application binary interface (ABI) and is able to emulate both user-level instructions and OS system calls for an individual process. A system VM provides a complete system environment capable of supporting an entire OS. Thus, a single computer can support multiple *guest OSs* environments simultaneously. A guest OS is an OS executing inside of a VM, i.e., a virtualized OS. In this dissertation, the term guest OS will be used to distinguish between virtualized and non-virtualized OSs. The second taxonomy level is based on whether the guest and host execute the same instruction set architecture (ISA). Multiprogrammed OSs are a typical example of process VMs that execute the same ISA. Process VMs for different ISA needs to emulate program binaries compiled to a different ISA. Two techniques can be applied to perform emulation:

- Interpretation: Fetches, decodes and emulates the execution of individual source instructions. This technique results in a huge performance impact because each instruction requires many native instructions to be emulated.
- Dynamic Binary Translation: Blocks of source instructions are translated into native instructions. This technique increases the performance because the translated blocks can be cached and repeatedly executed without requiring new translation.

Interpretation and binary translation have different performance profiles [70]. Interpretation has lower startup overhead but greater overhead for overall execution. On the other hand, binary translation has high initial overhead in order to translate the instructions

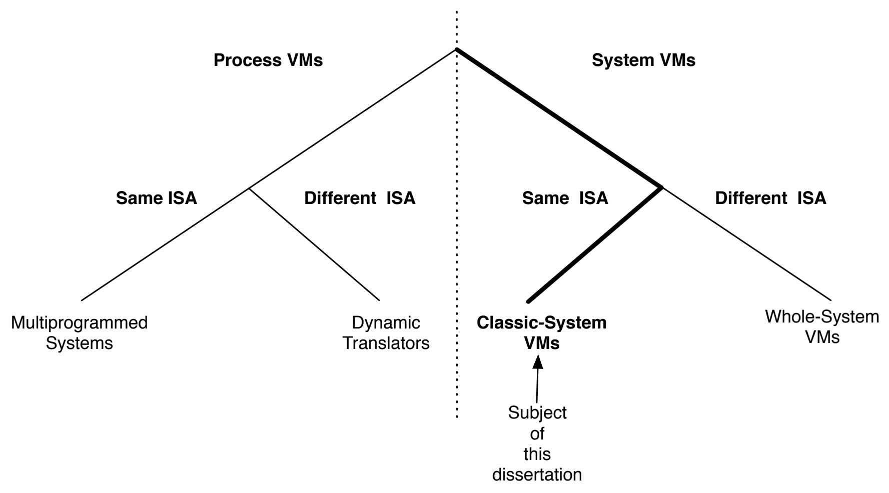


Figure 2.1: A taxonomy of Virtual Machines. Classic-system VMs are the subject of study of this dissertation. Adapted from [70].

for the first time. Thus, some VMs use both techniques combined with a heuristic to determine which source instruction blocks will be interpreted or translated. The best example of a process virtual machine is the Java Virtual Machine [46]. Java is a high-level language (HLL) designed for portability. The Java compiler generates a portable intermediate ISA that can be emulated by the Java Virtual Machine (JVM) for different host ISAs.

In system VMs, the virtualization software is often called a *virtual machine monitor* (VMM) or simply *hypervisor*. System VMs designed to support different ISAs need to emulate all software (OSs and applications). Thus, they are also called whole-system VMs. There are important applications for this kind of VM. For example, an ARM platform including the processor, memory and peripherals can be fully emulated on an Intel IA-32 processor. Thus, the designers can test their software without the target hardware. Additionally, a legacy software can be supported by newer platforms without requiring engineering work to port. An example of a system VM for different ISAs is the QEMU [11]. QEMU is an open source application that can be used either as an emulator or virtualizer. As an emulator, it can run OSs and programs compiled for different ISAs. When used as a virtualizer, it can execute the guest OS code directly on the host CPU. Another example is Open Virtual Platforms (OVP)[1] designed to accelerate embedded software development allowing hardware platforms to be described in C language. Thus, the designers can describe their custom platform, testing and debug their software without the real target.

[1] http://www.ovpworld.org/

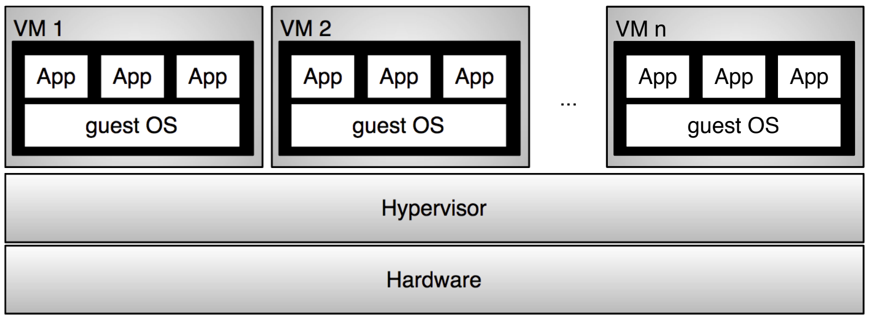
System VMs to the same ISA, also called classic-system VMs, were developed during the 1960s and they were the first application of virtual machines. At that time, mainframe computer systems were large and expensive. Thus, computers needed to be shared among users. Different users sometimes required different OSs creating the first practical application for virtualization. With the popularization of PCs, interest in classic-system VMs was lost. However, during the 1990s with the appearance of server farms, the classic-system virtualization became popular once again. In this case servers are shared among several users and VMs are a secure way of partitioning the user's software. Classic-system VMs have a lower performance impact when compared to whole-system VMs because the hardware directly executes the VM's instructions.

Recently, the interest in classic-system VMs for embedded systems has increased. However, classic-system VM as applied to server virtualization does not fit directly with ESs. Their intrinsic characteristics as discussed in Section 2.1 motivated the study of new approaches for virtualization on ESs. Thus, the subject of this dissertation is classic-system VMs for embedded systems. Hereafter, the term virtual machine refers to classic-system VMs, unless otherwise noted.

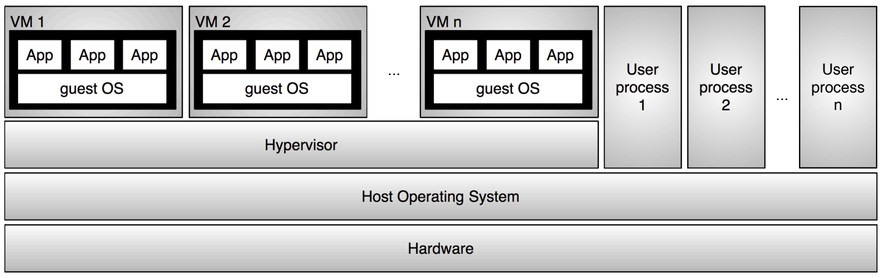
2.2.1 Types of Hypervisors

Virtualizing software (VMM or hypervisor) can be divided into two different kinds called *type 1* and *type 2*. Figure 2.2 depicts both. Type 1 hypervisors are also called *native* or *bare-metal* hypervisors. The virtualization layer is implemented between the hardware and the guest OSs, i.e., there is no native OS executing in a system that adopts type 1 virtualization as shown in Figure 2.2(a). A representative type 1 hypervisor is Xen (see Section 3.1). A type 2 hypervisor is implemented as a distinct software layer on top of an existing OS (see Figure 2.2(b)). From the point of view of the underlying OS, it is treated as a normal process and is subject to the same scheduler policies as other user and system processes. In this kind of virtualization, a guest OS can coexist along with the non-virtualized user processes. A representative type 2 hypervisor is KVM (see Section 3.2).

When a home/office user desires to utilize virtualization for some purpose, type 2 hypervisors are most popular. The type 2 hypervisor is installed as a user's application on the existing OS. This technique allows the user to execute a guest OS along with other user applications. Thus, the guest OS is competing with user applications for system resources, and it is submitted to the same host OS policies as any other process. In terms of memory requirements, type 1 hypervisors are more efficient than type 2 since there is not the overhead of the host OS. Additionally, the hypervisor has the freedom to control all system resources. The main application for type 1 hypervisors is to execute multiple OSs on the same computer, which makes it the perfect choice for server consolidation.



(a) Type 1 hypervisor.



(b) Type 2 hypervisor.

Figure 2.2: Types of hypervisors.

Type 1 and type 2 hypervisors are available for embedded virtualization. However, most type 2 hypervisors used in embedded virtualization were designed for desktop virtualization and adapted for embedded systems. As presented in the Chapter 3, hypervisors designed specifically for embedded systems tend to implement the type 1 approach. Due to restrictions such as memory footprint and real-time constraints, the type 1 model fits embedded systems better. The virtualization model for ESs proposed in this dissertation depicts a type 1 hypervisor.

2.2.2 Requirements for Processor's Virtualization

The classic paper written by Popek and Goldberg [60] formalizes the conditions for a processor architecture to be efficiently virtualized. Efficient virtualization means that a VM can be constructed over the processor's architecture without the necessity of modifying the guest OS. In order to understand their theorem, it is necessary to define the difference between control-sensitive, behavior-sensitive and privileged instructions. Control-sensitive instructions can change the configuration of the system's resources, while, behavior-sensitive instructions depend on the configuration of the resources. Instructions that are neither control-sensitive nor behavior-sensitive are termed innocuous [70]. Moreover, instructions that trap the OS when executed in user mode are called privileged instructions. In a hypervisor, any instruction that attempts to change the configuration of the system's resources must be intercepted and emulated accordingly. This is possible when two requirements are accomplished:

- The instruction was executed in user-mode and;
- The control-sensitive and behavior-sensitive instructions are privileged instructions too.

The first requirement is easily accomplished executing the hypervisor in kernel-mode and the guest OS in user-mode. However, the second requirement relies on the processor's architecture. Thus, Popek and Goldberg stated their theorem:

Theorem 1. *"For any conventional processor a virtual hypervisor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

Figure 2.3 depicts the theorem. In Figure 2.3(a) the control-sensitive instructions are not a subset of the privileged instructions, i.e., some of them can be executed in user-mode without trapping the hypervisor. This case violates Popek-Goldberg's theorem and cannot be efficiently virtualized. On the other hand, Figure 2.3(b) shows the control-sensitive instructions as a subset of the privileged instructions resulting in a processor that can be efficiently virtualized.

Para-virtualization (see Subsection 2.3.1) can be applied when the processor's architecture is not compatible with the Popek-Goldberg theorem. Thus, allowing a hypervisor implementation for that architecture. Alternatively, processors that support Popek-

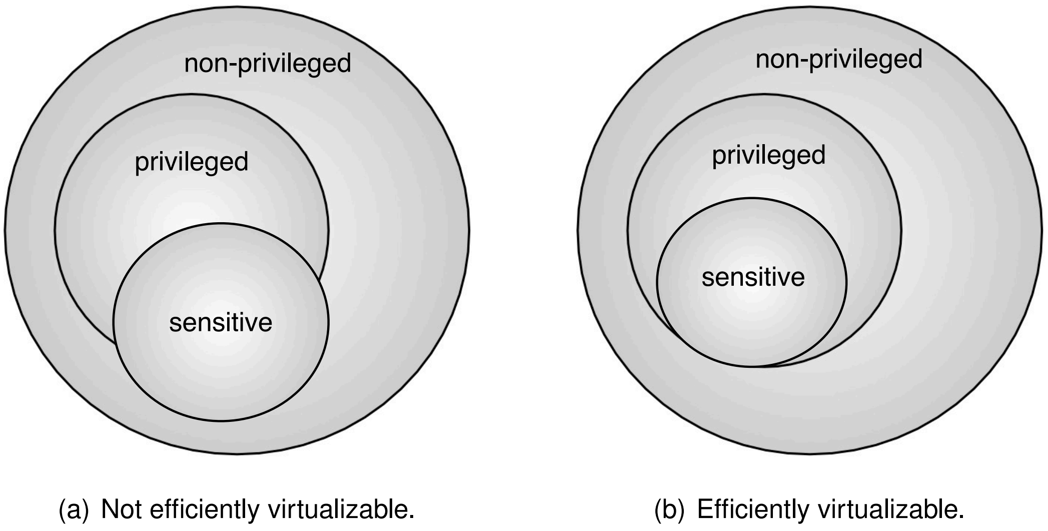


Figure 2.3: Illustrating Popek and Goldbergs's Theorem. Adapted from [70].

Goldberg's theorem can be fully-virtualized (see Subsection 2.3.2). Section 2.3 explains the difference between para-virtualization, full-virtualization and hardware-assisted virtualization.

2.3 Enabling Techniques and Technologies for Virtualization

The evolution of virtualization technology has resulted in several different approaches. The variety of processor architectures and the appearance of new applications with different needs, has driven techniques that can deliver virtualization on almost all devices. This section describes the three main techniques adopted for hypervisor implementation: para-virtualization (Subsection: 2.3.1), full-virtualization (Subsection 2.3.2) and hardware-assisted virtualization (Subsection 2.3.3). These techniques can be adopted separately or through a hybrid approach, combining two or more of them. The virtualization model and the resulting hypervisor implementation, which is the subject of this work, are a combination of these three techniques.

2.3.1 Para-virtualization

Hypervisors for processor architectures that do not fit Popek and Goldberg's theorem, as explained in Subsection 2.2.2, require different approaches. These architectures have control-sensitive instructions that do not trap the hypervisor. Therefore, the hypervisor cannot avoid an application changing the processor's behavior. There are two ways to avoid this problem: dynamic binary translation or para-virtualization (PV). Hypervisors that perform dynamic binary translation need to read the VM's instruction prior to their execution in order to emulate the control-sensitive instructions that do not trap the hypervisor. Thus, causing a significant performance penalty when compared to PV. Para-virtualization is a technique that requires modifying and recompiling the guest OS prior to its installation in a VM. These modifications consist of substituting control-sensitive instructions or other OS parts, like the I/O subsystem, by hypercalls. However, for proprietary software or other situations where the source code is not available, the only alternative is dynamic binary translation.

Hypercalls are services provided by the hypervisor and invoked from the guest OS. They are equivalent to system calls (syscalls) in an OS and Figure 2.4 describes their relationship. In operating systems, functions that can only be performed by the OS, like memory allocation or access to the file-system, result in calls to the OS. Similarly, functions that can only be performed by the hypervisor, such as the execution of control-sensitive instructions, can be converted to hypercalls.

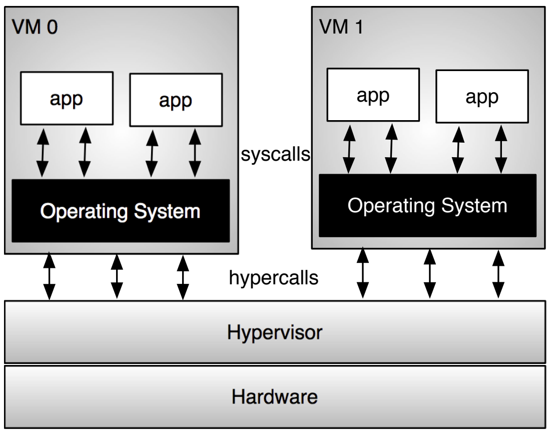


Figure 2.4: Relation between hypercalls on a virtualized system and syscalls on a typical operating system.

PV is not limited to processor architectures that cannot support Popek and Goldberg's theorem. In fact, it can extend hypervisor capabilities. For example, PV can be used to implement communication between the VM and the hypervisor or inter-VM communication. Another application for PV is to improve performance, since fully-virtualized hypervisors have a performance issue. If the VM executes control-sensitive instructions constantly, it will result in an excessive number of traps to the hypervisor, impacting performance. Thus, the OS can be modified to substitute these instructions with hypercalls.

The main limitation of para-virtualization is the engineering work needed to modify the OS to execute in a specific hypervisor. When utilized as a method to extend the capabilities of a virtualized platform, e.g., to support inter-VM communication, PV can be applied as a loadable module. Usually modern operating systems support kernel modules [17], i.e., executable code that can be dynamically loaded into an OS's kernel to add new functionalities. For example, an inter-VM communication module can be added without modifying the OS's source code. VirtIO [64] is a effort to standardize the I/O interfaces for Linux hypervisors consisting of a set of Linux modules. However, para-virtualization used as a method to provide CPU virtualization, i.e., to avoid control-sensitive instructions, requires modifications to the OS's kernel. These modifications must be compatible with the hypervisor's requirements making them unique.

2.3.2 Full-virtualization

The full-virtualization technique requires that the processor architecture fits with the Popek-Goldberg's theorem to implement a virtual machine able to execute a guest OS without any modification. All privileged instructions must be emulated by the hypervisor. Full-virtualization allows the guest OS to be completely decoupled from the virtualization layer. Hence, the guest OS does not need to be modified to be virtualized. Unmodified guests are advantageous because they avoid additional engineering work. However, if the number of emulated privileged instructions are excessive, a fully-virtualized system can suffer from performance penalties. Due to this, para-virtualization was widely adopted in the first generation of embedded hypervisors.

Para-virtualization and full-virtualization can coexist in a combined approach. This approach consists of fully-virtualizing the CPU and uses para-virtualization for certain peripherals or hypercalls for special services. For example, a network device may be para-virtualized to be shared among guest OSs or the hypervisor may implement a hypercall for a guest OS to change its execution priority. Additionally, the designers can implement hypercalls to substitute only instructions that generate an excessive number of traps and to use emulation for the remaining instructions. However, this requires prior study to determine what instructions must be substituted.

2.3.3 Hardware-assisted Virtualization

This section offers a brief description of typical virtualization extensions. Dedicated hardware for virtualization can simplify hypervisor design and to improve performance and security. However, these advantages require additional hardware support increasing the processor complexity, cost and power consumption. These factors are critical for the ES market. Usually, the specification of virtualization extensions describes a full set of hardware features. However, this set of features must be flexible enough to allow for partial implementation. Thus, the processor's manufacturer can determine the level of hardware-assisted support based on the necessity of the market. For example, limited hardware support may still allow for full-virtualization, but offers poor performance speed-up. In this scenario, the hypervisor must deal with the absence of features to support different processor models. Following is describe three important hardware features typically available in virtualization extensions sets.

Additional Privilege Levels

The typical para-virtualization approach on processors without proper virtualization support suffers from the inability to isolate the guest OSs' kernel and user-space. The hypervisor is typically executed in kernel-mode and the entire guest OS is executed in user-mode. As a result, there is no memory isolation between the guest kernel and applications. Thus, a misbehavior or malicious application can disrupt the entire guest OS. A secondary effect is the excessive number of traps generated by the guest OS executing in user-space. This effect can be reduced with PV.

Adding new privilege levels to the processor allows the guest OS to be executed in both kernel and user-modes, since the hypervisor is located in a higher privileged level. Figure 2.5 shows the model of a processor implementing a third privilege level called supervisor. Beyond allowing memory isolation inside the guest OS, the additional privilege level allows one to distinguish between user, kernel and supervisor exceptions. User exceptions, like system calls, can be handled directly by the guest OS kernel without the necessity to trap the hypervisor. Also, the hypervisor may control the accessibility of some hardware features, like configuration register or instructions, to the guest OS. Thus, the hypervisor may choose a policy of less intervention or a more controlled guest OS.

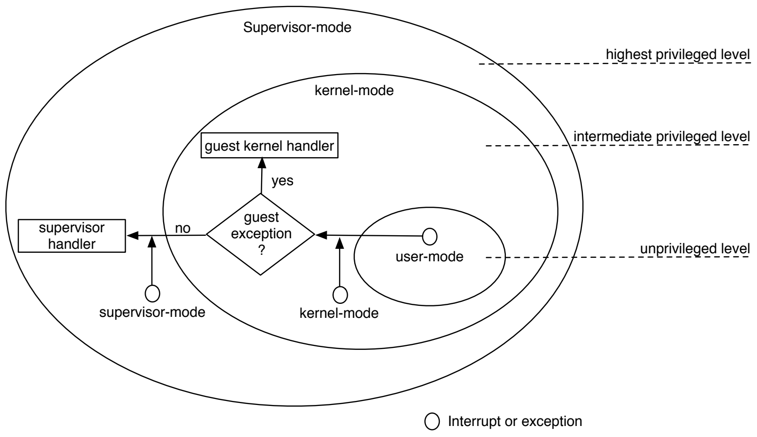
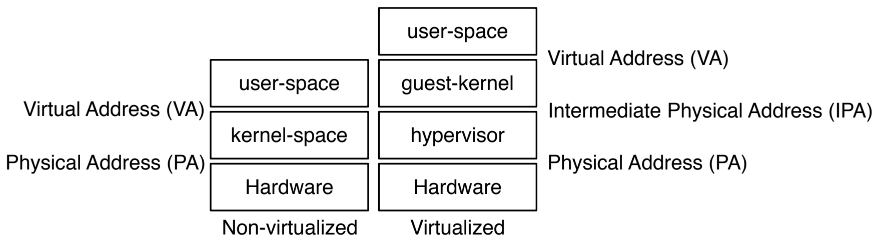


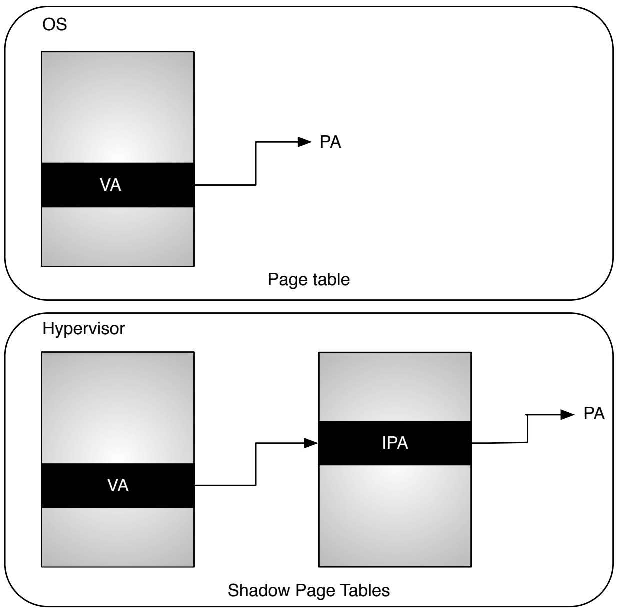
Figure 2.5: Supervisor privileged level for hypervisor execution.

2-Stage TLB Translation

Processors targeting GPOSs rich in features must support a memory management unit (MMU) to provide a virtual memory mechanism. Thus, the OS can implement memory isolation between processes increasing software reliability and security. Translation look-aside buffer (TLB) is a common hardware construction to assist MMU, which works like a page table cache. High performance processors can automatically performs page walks in the page table when a TLB miss occurs, i.e., the target address is not found in TLB. However, simpler processors will trap the OS in this situation. Thus, the OS must handle a TLB exception to reconfigure the TLB cache. In a virtualized system, the hypervisor must virtualize the MMU keeping the guest OSes isolated from each other, which requires a second stage of address translation. Figure 2.6(a) shows the difference between virtual memory translation in a non-virtualized versus a virtualized system. In a non-virtualized system, the OS translates virtual addresses (VAs) to physical addresses (PAs) using its page table to configure the TLB. In a virtualized system, VAs are translated to intermediate physical addresses (IPAs), which must be managed by the hypervisor. Processors without the proper virtualization support require the hypervisor to implement a technique called shadow page tables that keeps the correct translation from IPA to PA. Figure 2.6(b) represents the difference between an OS page table and a hypervisor shadow page tables. When using this technique, the guest OS still manages its page tables, but it cannot configure the processor's TLB directly. The hypervisor handles the page faults reading the address mapping from the guest OS and creating its shadow page table. This second-stage address translation performed by software generates an excessive number of hypervisor exceptions, since, the TLB configuration involves privileged instructions. Again, the number of exceptions can be reduced using hypercalls.



(a) Non-virtualized versus virtualized virtual memory address translation.



(b) OS page table versus hypervisor shadow page tables.

Figure 2.6: Memory virtualization approaches in non-virtualized versus virtualized systems.

Current virtualization extensions for embedded processor families, like MIPS and ARM, implement a second-stage TLB translation in hardware. Essentially, the hardware performs the translation from IPA to PA without software intervention. The hypervisor still manages its page table mapping IPA to PA. However, the guest OS is allowed to configure directly the TLB. This is possible because the two TLB stages are configured from different processor modes. For example, the hypervisor (supervisor-mode) configures the second-stage TLB while the guest OS (kernel-mode) configures the first-stage TLB. The resulting PA is generated by the hardware combining both TLBs. This mechanism decreases the number of hypervisor exceptions drastically, since the guest OS directly manages its TLB while decreasing the hypervisor complexity.

Directly Mapped I/O

In some cases, it is desirable to map a hardware device directly to a guest OS, e.g., a serial port or a USB device could be directly handled by a guest OS. However, in processors without proper virtualization support, all interrupts are handled in kernel-mode forcing the hypervisor to handle all interrupt sources. This prevents an efficient implementation of a directly mapped I/O.

One desired feature of a virtualization extension module is the capability of a hypervisor to map an interrupt source to a designed guest OS. For example, an interrupt used as system timer for guest OS scheduling purposes could be directly mapped, avoiding hypervisor intervention during the guest context-switch. Directly mapped devices require the hypervisor to configure an entry in its page table mapping the physical device address to the intermediate physical address (see Subsection 2.3.3) expected by the guest OS. Still, it must map the associated interrupts to the guest OS. Thus, the guest OS will receive interrupts and read or write to the device without any hypervisor intervention.

2.3.4 I/O Virtualization

There are different ways to manage I/O on a virtualized platform. On a platform without hardware-assisted virtualization, the hypervisor must implement device drivers for all peripherals in use, dealing directly with the system's I/O. This technique is used in para-virtualized hypervisors. Thus, the device drivers at the operating system level are substituted by para-virtualized device drivers. Beyond to the disadvantage of modifying the OS's I/O subsystem to support para-virtualization, hypervisor intervention in I/O increases the virtualization overhead [83]. A directly mapped I/O can be used to improve the I/O performance on guest OSs, as explained in the Subsection 2.3.3. However, it lacks correct support for dynamic memory access (DMA) technique. DMA allows a peripheral to have direct access to the main memory. In a virtualized platform, the hypervisor re-maps the guest OS's memory addresses causing DMA devices to fail. Additionally, this is a security hole since a guest OS can use the DMA to transfer data to another guest OS's memory. Thus, the hypervisor must deny access to the guest OSs from the DMA controller, and to implement a para-virtualized mechanism to allow DMA to such systems.

Modern processors with proper hardware support for virtualization implement a input–output memory management unit (IOMMU), as supported by the x86 family. IOMMU works similar to the processor's MMU (see Subsection 2.3.3). However, it translates memory accesses performed by devices instead of by the processor. Figure 2.7 compares IOMMU to the MMU. The IOMMU can solve the problem of memory re-mapping on virtualized platforms since the hypervisor can map the DMA addresses to the correct physical address. Additionally, it can avoid the security hole on DMA since it provides memory protection from peripheral read/write in a similar way that MMU provides spatial isolation.

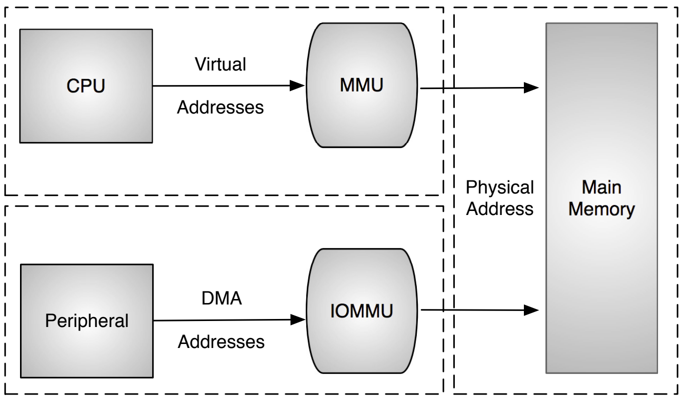


Figure 2.7: Comparison of the IOMMU to the MMU.

Unfortunately, IOMMU is not present on all architectures of the current generation of hardware-assisted virtualization for embedded processors. For more details, Section 2.6 discusses the evolution of virtualization on embedded systems.

2.4 Problems Caused by Virtualization

This section gives a brief introduction of two important problems faced in virtualized platforms: the hierarchical scheduling problem (Subsection 2.4.1) and the lock the holder preemption problem (Subsection 2.4.2). Chapter 4 explains how the virtualization model proposed by this dissertation avoids this problems.

2.4.1 The Hierarchical Scheduling Problem

As mentioned in the Subsection 2.1, real-time is a key feature in many ESs. Enabling real-time support on virtualized platforms causes the hierarchical scheduling problem, also known as two layers scheduling problem. A hypervisor schedules Virtual CPUs (VCPUs), and a guest RTOS executing in a VCPU schedules its processes or tasks. Even ensuring real-time characteristics at the VCPU level, it is difficult to ensure real-time execution to the tasks over the RTOS. However, if VM and hypervisor use proper real-time algorithms, the system can be modeled as a hierarchy of schedulers, and its real-time performance can be evaluated by using hierarchical scheduling analysis techniques [20].

One reason to use virtualization is to allow for the coexistence of different guest OSs designed for different purposes executing in the same system. For example, a *general-purpose OS* (GPOS) executing along with a legacy RTOS. However, different OSs have different characteristics, e.g., RTOSs privileges real-time over throughput performance. A hypervisor for ESs must be flexible enough to deal with these conflicting characteristics. Thus, more effective approaches should be taken to deal with the two layer scheduling problem.

Some authors, e.g., Kiszka et al. [41], propose the implementation of hypercalls to allow the guest OS to inform to the hypervisor of its time constraints. Other approach is the use of compositional scheduling as proposed by Lee et al. [42]. Additionally, an alternative adopted by this dissertation is to perform the scheduling of real-time tasks directly by the hypervisor, i.e., avoiding the two layer scheduling problem and performing a strong temporal separation between general-purpose guest OSs and real-time entities.

2.4.2 Lock Holder Preemption Problem

The Lock Holder Preemption (LHP) problem in a virtualized environment happens if a guest OS executing in a SMP configuration is preempted while its kernel is holding a spinlock. Spinlocks are a synchronization primitive widely used in OS's kernels. With spinlocks, a thread waiting to acquire a lock will wait actively monitoring the lock [52]. As this lock remains, any other VCPU from the same guest trying to acquire this lock will have to wait. Spinlocks imply active waiting, thus, the CPU time is wasted resulting in serious performance and real-time degradation. The LHP problem is possible even if two or more VCPUs execute on a single physical CPU concurrently.

Several approaches were proposed to deal with the LHP problem. The VMWare hypervisor[2] employs the co-scheduling technique [57], consisting of a scheduler that tries to execute processes communicating with each other in the same time slice, thus, reducing the chances of blocking. Uhlig et. al. [81] proposed a full-virtualized and a para-virtualized technique to deal with the problem. In the full-virtualized technique, the authors consider that the spinlocks are always acquired and released in kernel mode. Thus, the hypervisor never preempts the guest OS when executing in kernel mode, avoiding the LHP problem. Their para-virtualization technique consists of a hypercall to be invoked when entering in spinlock and another hypercall to be invoked when leaving the lock. Thus, the hypervisor knows when a VCPU is in a spinlock and should not preempt it.

In ES, the main consequence of the LHP problem is the impact in real-time responsiveness. Thus, the LHP should be carefully considered when designing a virtualized ES.

2.5 Virtualization for Embedded Systems

Virtualization is a widespread technology for enterprise and workstation applications, but, it is a relatively new concept in the ES area. The importance of virtualization for ESs is due to its ability to address some of the growing challenges in this area, e.g., the increasing software complexity and time-to-market. One advantage of virtualization is its ability to support heterogeneous operating-system environments to address conflicting requirements such as high level APIs, real-time support or legacy software. The architectural abstraction offered by virtualization allows for migration of multicore guest OSs to virtualized single core systems essentially without modification. Security is another important motivation, since, virtualization can enable a strong spatial separation between the guest OSs [31]. Virtualization as deployed in the enterprise environment cannot be directly used in ESs. Although ESs are starting to implement some features of general-purpose systems, some differences remain. ESs still present critical real-time constraints and frequently have re-

[2]http://www.vmware.com/

source constraints, such as reduced battery capacity, memory and processing power. Thus, virtualization in the ES world is a different challenge than that faced in the enterprise environment. To distinguish server virtualization from embedded system virtualization, the term embedded virtualization is used. Additionally, in this dissertation the term virtualization refers to embedded visualization unless stated otherwise.

This dissertation focuses on embedded systems that require large software stacks to implement their functionalities. These systems can use virtualization to meet these requirements. In addition, the proposed virtualization techniques presented in this dissertation can be used to improve real-time responsiveness and increase security on ESs. Some examples of ES candidates are customer electronics (CE) like smart phones, tablets, set-up box among others. Moreover, enterprise and industrial electronic devices such as network devices (routers and switches) may apply embedded virtualization. Recently, virtualization for IoT devices emerged as a trend. The main advantage of virtualization on IoT is security. Hypervisors can provide memory isolation between applications and implement a Root of Trust software base, which means an environment where the hypervisor guarantees the authenticity of the virtual machines.

2.6 Past, Present and Future of Embedded Virtualization

Roughly ten years ago, virtualization started to move to the ESs field. At that moment, embedded virtualization was ruled by hypervisors that adopted para-virtualization to overcome the performance issues of full-virtualization. Though some popular hypervisors used for server computing were adapted for embedded systems, other hypervisors were developed from scratch. Some of these hypervisors are still in use, and they will be discussed in Chapter 3. These hypervisors were especially designed to deliver virtualization in embedded systems, but in a lightweight way, and implement techniques to improve the real-time responsiveness along with memory separation.

With the advance of the semiconductors and their decreasing cost, it is possible to implement hardware support for virtualization on embedded processors affordably. The first embedded processors with hardware-assisted virtualization emerged around five years ago. The possibility of implementing full-virtualization without the performance penalties of the past motivated the appearance of newer hypervisors. The current stage of embedded virtualization allows full-virtualization of the CPU combined with para-virtualized I/O. Thus, hybrid hypervisors combine full and para-virtualization. In a few years, proper support for IOMMU on embedded processors will allow for high performance communication between guests using the DMA hardware.

In the next five years, virtualization will increasingly be used in embedded systems [61]. Virtualization will not be restricted to powerful embedded devices, but will also be used in IoT devices, primarily motivated by security concerns. Thus, embedded hypervisors will need to execute in devices with extreme hardware restrictions. Small footprint and lightweight execution will still be important.

3. STATE-OF-THE-ART IN EMBEDDED VIRTUALIZATION

This section describes the current status of embedded virtualization. As shown in Chapter 2, hypervisors designed for general purpose virtualization cannot be directly adopted for ES. However, this chapter starts by showing well-known hypervisors widely used for cloud computing. They present larger footprints and overheads than hypervisors that are specially designed for embedded systems. Several authors have dedicated their work to adapting them for ES. The chapter is divided into the following sections: Subsection 3.1 presents the architecture and proposals based on the Xen hypervisor. Xen was designed for enterprise appliances. Therefore, there are ports for embedded devices and special support for real-time. Subsection 3.2.3 presents proposals based on KVM (Kernel-Based Virtual Machine), a type 2 hypervisor, which was ported for the ARM architecture. The remaining hypervisors that will be presented were developed for ES. Section 3.3 describes the SPUMONE hypervisor. Section 3.4 presents XtratuM, a hypervisor for safety critical systems. Section 3.5 presents three different embedded hypervisors based on the L4 microkernel family: OKL4, L4/Fiasco, PikeOS and Mini-Nova. Section 3.6 presents the M-Hypevisor. Section 3.7 shows the XVisor embedded hypervisor. A comparative study between the hypervisors is presented in Section 3.8. Section 3.9 explains how the RT-Linux patch modifies the Linux kernel and how it is similar to virtualization. Finally, Section 3.10 concludes this chapter.

3.1 The Xen Hypervisor

Xen is an open-source type 1 hypervisor that has been developed over the last 13 years [9]. Today, Xen is used for different commercial and open-source projects, such as: server virtualization, desktop virtualization, security and embedded applications. Additionally, Xen supports the largest clouds in production today[1]. Initially developed for desktop or server application, it was modified to be embedded, especially for ARM processors.

[1] http://www.xenproject.org/

3.1.1 Xen's Architecture

Figure 3.1 depicts Xen's architecture. Xen is responsible for handling CPU, memory, interrupts and scheduling the VMs. Xen is a type 1 hypervisor, thus, it interacts directly with the hardware and the VMs executes over it. A running instance on a VM in Xen is called domain or guest. There is a special domain, called *Domain 0* (Dom0), that is responsible for I/O and also contains a stack to manage the VM's creation, destruction and configuration, called *Toolstack*.

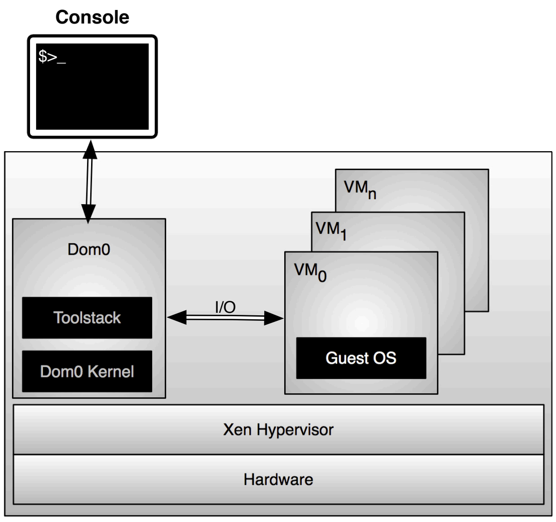


Figure 3.1: Xen's Architecture. Adapted from [9].

The Xen hypervisor has less than 150,000 lines of code and a footprint around 600KB. However, the footprint can variate with different compilers and compilation flags. Xen does not implement I/O functions such as networking or storage itself. Instead, I/O functionality is accomplished by the Dom0. Dom0 has special privileges to access the hardware directly and it is responsible for handling the I/O for all VMs. Thus, if a VM needs an I/O function it must interact with Dom0, which will execute the transaction with the hardware. Additionally, the Dom0 implements the Toolstack, which exposes an interface composed of the command line and graphical interface, as well as interacts with cloud APIs like OpenStack [67] for configuration purposes.

The Xen architecture requires that the guests are modified (patched), since all I/O are redirected to Dom0. Essentially, Xen was designed to be a para-virtualized hypervisor. Therefore, this approach prevents unmodified guest OSs be virtualized on top of Xen. Thus, proprietary OSs like Microsoft Windows can not be virtualized. Aiming to overcome this limitation, Xen implements the hardware-assisted virtualization (HVM) that uses virtualization extensions from the CPU, like Intel VT or AMD-V. Xen does not deal with I/O directly, thus, it needs to use the Qemu [11] to emulate BIOS, IDE disk controller, VGA graphics adapter among others devices. This allows for Xen to perform full-virtualization. However, the device emulation causes a performance issue and fully-virtualized guests over Xen are usually slower than the para-virtualized ones. Nonetheless, Xen provides a technique to avoid the performance issue when using HVM, called PV on HVM. This technique consists in to combine para-virtualization and full-virtualization. Thus, para-virtual device drivers are installed in the guest OS to bypass the emulation for storage and network I/O. CPU and memory virtualization continues to be performed by hardware support. This allows for performance improvement for guest proprietary or even open source OSs. However, it requires the implementation of para-virtual device drivers to the guest OSs. Both PV and HVM guests can coexist on a single Xen system.

3.1.2 The Xen Scheduling Algorithm

An important component of a hypervisor is the scheduler. The default Xen's scheduler is called Credit Scheduler [23]. Xen implements two other scheduling algorithms: the Simple Earliest Deadline First (SEDF) [10] and the Borrowed Virtual Time (BVT) [24]. Although at the present time both algorithms are available, they are marked as deprecated and probably they will be removed from the Xen's source code. Nevertheless, several works still rely on SEDF to improve real-time on XEN. Thus, just the Credit Scheduler and the SEDF will be discussed.

In the Credit Scheduler, each domain has two special attributes: weight and cap. The weight determines the proportional fair share of CPU resource. For example, a domain with weight of 1024 will get the CPU two times more than a domain with weight of 512. The weight values range from 1 to 65535 and the default is 256. The cap is an optional parameter used to determine the amount of CPU that a domain will be able to consume, even if the host system has idle CPU cycles. The cap is expressed as a percentage of one physical CPU. For example, a cap of 0.5 means that the domain can take up to 50% of CPU time, 1 means the domain can take upto 100% of the CPU time and 2 means the domain can take upto 100% of two CPUs' time. When cap is 0 means there is no limit for CPU usage. Credit scheduler keeps a local queue of runnable VCPUs by physical CPU ordered by priority. When a VCPU is scheduled, the algorithm computes how much time (credits) was consumed and subtracts from the total time reserved for it. If the VPCU exceeds its available credits it is in "over" status, otherwise, it is in "under" status. Once in over status, the VCPU needs to wait for the next period to execute again. By default, the scheduler quantum is 10ms and the VCPU credits are recomputed each 30ms. In a SMP platform, when a CPU does not find a VCPU in under status in its local queue, it looks on other CPU queues for one. This is called SMP load balancing and it guarantees that there is not an idle CPU when there are runnable VPCUs in the system.

The SEDF scheduler allows to allocate processing resources for each domain according to period, deadline (the time when the period ends) and capacity (the amount of processing time reserved to the domain in a period). In fact, the algorithm guarantees that a domain will be executed for the amount of time given by its capacity on each time interval given by its period [56]. The algorithm keeps the deadline for each domain and the amount of processing remaining to be done before the deadline. The domains are sorted in the execution queue according to their deadlines. An interesting effect of this algorithm is that when a domain consumes processing time, its deadline will be moved forward. Thus, I/O-intensive domains that consume little processing time will typically have earlier deadlines, which means higher priority over CPU-intensive domains. This is advantageous to environments that combine CPU and I/O intensive domains. The main disadvantage of the SEDF algorithm is to keep local queues organized by CPU on multicore processors preventing VCPU migration across multiple cores and avoiding dynamic workload balance. Nevertheless, it was made obsolete because with Credit scheduler is possible to improve scheduling performance on multiprocessors.

3.1.3 Xen-Based Proposals

The proposals to improve real-time responsiveness on Xen hypervisor are typically based on two different approaches: a) improving the Xen's scheduler real-time responsiveness, or; b) dealing with the hierarchical scheduling problem. The first approach typically tries to improve the real-time scheduler responsiveness by applying different scheduling algorithms or non-intrusive techniques to monitor the guest's behavior. Nevertheless, such approaches generally do not consider the hierarchical scheduling problem. The second approach uses theoretical research on hierarchical real-time scheduling or intrusive techniques that require modifications inside of the guest OSs.

Improving The Scheduler's Real-time Responsiveness

Ongaro et al. [56] studied the performance impact on the Xen hypervisor when multiple guests with different workloads (CPU-intensive, I/O-intensive and latency-sensitive) run concurrently. The results led to some optimization proposals, including disabling preemption for Dom0 and sorting the scheduler run queue based on the remaining credits to give priority to I/O-intensive domains. These optimizations were focused on improving the I/O performance, nevertheless, the authors claim that it impacts latency positively. Finally, their study showed that latency-sensitive applications will perform better when not combined in the same domain with CPU-intensive applications. Instead, latency-sensitive applications should be placed within their own domain.

Lee et al. [43] introduced the laxity concept enhancing the soft real-time performance of the Xen Credit scheduler. Each domain has an attribute named laxity used to define the position where its VCPUs will be inserted in the scheduler's run queue. In other words, it is possible to use laxity to determine the maximum desired latency for a real-time domain, because when a VCPU is inserted into the run queue, it is inserted where its deadline is expected to be met.

Masrur et al. [48] implemented a new scheduler for Xen based on the SEDF scheduler aiming to improve its worst-case response time, named *Priority-based scheduling plus SEDF* (PSEDF). The major difference is that PSEDF implements the concept of real-time domains (domRT), where, a domRT has scheduling priority over non real-time ones (domU). When several domRTs coexist at the same Xen instance, priorities must be defined between them. The domUs continue to be scheduled with the default SEDF. The resulting scheduler allows safety-critical applications (e.g. airbag control and brake system) to be run on the same hardware with general-purpose applications (e.g. navigation and multimedia). The main drawback of the authors' technique is that a domRT can just contain a single real-time application to avoid the hierarchical scheduling problem.

Hu et al. [34] proposed a technique to improve the I/O performance of Xen on multicore processors. Their technique take advantage of the fact that traditional hypervisor schedulers, including Xen, focus on sharing the processor's resources fairly among domains, while, the scheduling of I/O resources is treated as a secondary problem. This seriously impacts on the performance of applications that execute I/O-intensive tasks, like network or storage. The authors' technique consists of implementing a scheduler for multicore dynamic partitioning, where, the processor cores are divided into three specialized subsets with different scheduling strategies. The subset called driver core host only the dom0, i.e, the dom0 is bounded to a physical core and never is scheduled. The fast-tick cores subset handles I/O events and its scheduler employs small time slices to ensure fast response to I/O events. Finally, the general cores subset is used for computation-intensive workloads. Their approach increases I/O performance, although, it degrades the performance for CPU-intensive applications.

Huacai et al. [15] proposed improvements to Xen's Credit scheduler from the real-time perspective to guarantee audio performance (avoiding human-sensible glitches) in virtualized environments. Their strategy consist of three approaches: flexible time slices, real-time priority adjustment and adaptive audio-aware. The Credit scheduler was modified to allow flexible time slices, i.e, the scheduler restriction of fixed time slices or quantum of 10ms (see Subsection 3.1.2) was removed allowing different time slice lengths. For example, a real-time domain may receive 1ms time slice and a non real-time may receive the default time slice (10ms). The real-time priority adjustment promotes a domain from non real-time to real-time when the user plays audio. The adaptive audio-aware algorithm is used to detect when the user starts/stops the audio player given to it a higher priority.

Credit scheduler was designed to be efficient with computation intensive workloads. However, I/O-intensive tasks suffer from latency issues. To minimize this problem, Zeng et al. [91] enhanced performance to the I/O latency-sensitive applications minimizing the system response time by balancing the VCPUs with BOOST priority in multi-core systems. BOOST is an additional priority level of the Credit scheduler, which allows a VCPU to preempt a running VCPU in under state. The results showed that their proposal can improve the response time for latency-sensitive applications without having adverse impact on compute-intensive applications.

Cheng et al. [16] optimized the Xen's scheduler for soft real-time applications. They introduced a new priority level marking high priority VMs as real-time domains. Additionally, their strategy includes management of the physical CPU's queue, limiting the number of real-time domains to guarantee to meet the scheduling requirements, and pinning VCPUs to physical CPUs to reduce the performance impact caused by VCPU migration. The authors evaluated their solution using IP telephony workloads for soft real-time applications. The results compared to Credit scheduler and RT-Xen showed that their purpose achieves best performance and maintains a fairer scheduling in the meantime.

Dealing With The Hierarchical Scheduling Problem

Masrur et al. [49] proposed the design of a scheduler to deal with the hierarchical scheduling problem targeting automotive applications. The authors consider that all domains schedule their applications using the deadline-monotonic scheduler algorithm. On the other hand, the hypervisor utilizes the rate-monotonic scheduler algorithm [14] [13] for VCPU scheduling. Thus, they proposed a method to determine optimum time slices and periods for each VM in order to maintain their time constraints. The proposed scheme was validated using the Xen hypervisor. Their solution solves the hierarchical scheduling problem without hypercalls between the VM and hypervisor, although the Xen's default scheduler algorithm must be modified to accommodate the rate-monotonic algorithm.

The RT-Xen project [84] implements a hierarchical scheduling architecture based on fixed-priority scheduling focused on soft real-time applications for single-core processors. It considered four fixed-priority scheduling policies for the RT-Xen scheduler: Deferrable Server [75], Sporadic Server [72], Periodic Server and Polling Server [68]. The four scheduling policies were implemented and compared to Xen's Credit Scheduler and SEDF schedulers. The experiments was conducted using a modified kernel Linux implementing the rate-monotonic scheduling algorithm. The results show that RT-Xen presents an acceptable soft real-time performance. The Deferrable Server presented the best results, while the Periodic Server suffered from a high number of missed deadlines in overloaded situations. Recently, it was released the RT-Xen 2.0 [85] supporting multi-core processors and up to eight combinations of real-time VM scheduling policies. Moreover, it supports a range of interfaces for compositional scheduling, which enables designers to calculate and specify the resource demands of VMs to the underlying RT-Xen 2.0 scheduler.

Lee et al. [42] proposed a Compositional Scheduling Architecture (CSA) that enables timing isolation among VMs and supports timing guarantees for real-time tasks running on each VM. The CSA supports a broad range of real-time scheduling algorithm at the hypervisor level, and it was implemented extending the original RT-Xen interfaces. The authors presented an extensive evaluation to demonstrate the utility and effectiveness of CSA in optimizing real-time performance. The results using both synthetic and avionics workloads showed significant improvements regarding response time.

Yoo et al. [89] studied the schedulability in VMs considering the quantization overhead caused by the Xen's scheduler. Quantization overhead comes from tick-based scheduling of Xen-ARM, which requires integer presentation of scheduling period and execution slice. To minimize the quantization overhead they proposed a new algorithm to provide accurate and efficient parametrization of real-time VMs. Next, they presented an inter-VM schedulability test to the real-time VMs. Their proposal was evaluated on an ARM platform using a RTOS. Their experiments showed that is possible to guarantee the deadline for real-time tasks scheduled in a hierarchical fashion. However, the SEDF scheduler must be used as Xen's scheduler. Moreover, their work does not cover the I/O, since it is complex in Xen and requires a separate investigation.

Other Proposals

Avanzini et al. [8] proposed the integration of the ERIKA OS[2] and the Linux on a dual-core processor through Xen. ERIKA OS is an open-source, low-footprint and real-time operating system certificated for automotive applications. Their system configuration consisted in the Linux as dom0 and ERIKA OS as domU. The system was prototyped on a cubieboard2[3], an ARMv7 with virtualization extensions. In their proposed setup, each of the domains runs on a dedicated core, assigned statically by the hypervisor. Linux is the control domain, thus performing the Xen toolstack. Moreover, it must grant to ERIKA access to any I/O-memory range needed. Their approach provides an improved temporal isolation of concurrent operating systems. The main drawback is to keep the guests on dedicated cores not allowing load-balancing.

[2]http://erika.tuxfamily.org/drupal/
[3]http://cubieboard.org/

3.2 KVM Hypervisor

The Kernel-based Virtual Machine (KVM) is a type 2 hypervisor designed to be a kernel-resident virtualization infrastructure, meaning it is totally integrated with the Linux Kernel since the 2.6.20 release. KVM is implemented as a Linux kernel module providing full-virtualization when hardware-assisted virtualization is supported, otherwise it provides para-virtualization. It supports SMP hosts and guests and enterprise level features like live migration[4], which is the ability to move guest OSs between physical servers.

3.2.1 KVM's Architecture

KVM's architecture is shown in Figure 3.2. It is composed of two main components: the KVM-loadable module that manages the virtualization hardware and exposes a configuration interface through the /dev file system; and a modified version of Qemu, called qemu-kvm, used to provide platform peripheral emulation.

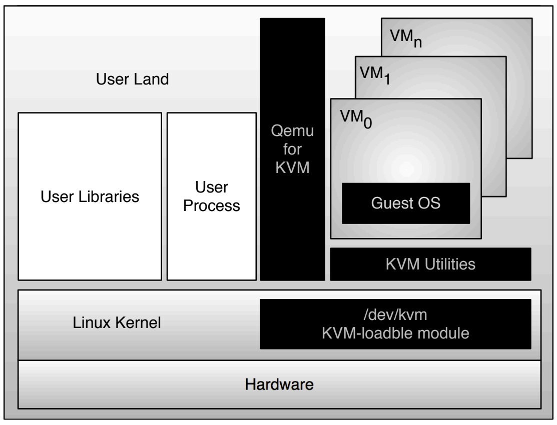


Figure 3.2: KVM's Architecture.

A utility called kvm is used in user land to boot a guest OS in KVM. Once booted, the guest becomes a process of the host OS being scheduled like any other process by the Linux scheduler. Each VM has its own virtual memory space mapped to the host's physical address. Nevertheless, I/O requests are mapped through the host kernel to the Qemu for emulation purposes. Although KVM is strongly dependent of the Linux OS, it supports a rich

[4]http://www.ibm.com/developerworks/library/l-hypervisor/

list of guest OSs which includes Linux distributions, Microsoft Windows, QNX, Solaris x86, FreeBSD among others.

KVM is mature enough to support advanced enterprise features such as enhanced security and live migration. Security in Linux is based on the Security-Enhanced Linux (SELinux) [62], which is a project developed by the NSA to add access control, multi-level and multi-category to the kernel. From the Linux kernel point-of-view, a guest OS is like any other process, it is possible to apply the standard Linux security model over it. Thus, SELinux can be used by the administrator to define permissions assuring that a VM resource can not be accessed by any other process or VM. Nonetheless, with live migration, a VM can be moved between physical servers transparently to the end user.

Similar to Xen (see Subsection 3.1.1), KVM implements a mix of para-virtualization and full-virtualization in order to speed up I/O operations. Para-virtualized device drivers are installed in the guest OS in order to avoid the necessity of Qemu emulation for certain devices, specially networking and block devices. Nevertheless, KVM implements a standard called VirtIO (developed by IBM and Red Hat in conjunction) [64], which is an independent interface for building device drivers to offer better interoperability between hypervisors. When a hypervisor uses a proprietary interface for para-virtualized device drivers the guest is not portable between different hypervisors. VirtIO improves the portability of guests between hypervisors.

3.2.2 The Linux Scheduling Algorithm

KVM does not have its own scheduler, instead, it takes advantage of the Linux scheduler. Therefore, a minimal background in Linux scheduling is necessary to understand how real-time is addressed by KVM. The Linux OS is a general purpose OS. Thus, the scheduling algorithm has a set of requirements with conflicting objectives: good throughput for background jobs, avoidance of starvation, dealing with high and low priority processes, among others. The Linux scheduler implements a set of rules, called scheduling policy, that determine how a process will be scheduled. In order to apply a scheduling policy onto a process the scheduler needs to classify the process according three different process classes:

- *Interactive*: Processes that interact constantly with the user. These processes spend a lot of time waiting for user inputs. Once, the input is received, it must return a quickly response or the user will consider the system unresponsive.
- *Batch*: Processes running in the background and that do not need user interaction.
- *Real-time*: Processes with very strict scheduling requirements.

Linux 2.6 and higher implement a sophisticated heuristic algorithm based on the past behavior of the processes to classify they as interactive or batch. The scheduler gives priority to interactive processes over batch ones [12], which results in a better response time even when the system is running under a heavy load. Real-time processes need special attention and there is a set of kernel features that improves the real-time responsiveness. In fact, improvements in the KVM real-time response implies on improvements in the Linux real-time response.

Conventional processes (non real-time) have an attribute value called static priority which determines the time quantum of the process following the Equation 3.1. The legal range for static priority is from 100 (highest priority) to 139 (lowest priority). As a consequence of Equation 3.1, higher priority processes get longer slices of CPU time.

$$\textit{quantum} = \begin{cases} \text{(140 - static priority)} \times 20 & \text{if static priority} < 120) \\ \text{(140 - static priority)} \times 5 & \text{if static priority} \geq 120) \end{cases} \tag{3.1}$$

In addition, a conventional process also has a dynamic priority which is the number actually used by the scheduler to choose the next process to run. The dynamic priority is determined by the Equation 3.2.

$$\text{dynamic priority = max (100, min(static priority - bonus + 5, 139))} \tag{3.2}$$

The bonus variable is determined by the process' average sleep time, that is, the average time (in nanoseconds) spent by the process while sleeping. The average sleep time is not allowed to be greater than 1 second. Each interval of 100 milliseconds represents a bonus value. For example, an average sleep time between 0 and 99 milliseconds results in a bonus equal to 0, intervals between 100 and 199 result in a bonus of 1. The greatest bonus value is 10. Nonetheless, the average sleep time is used by the scheduler to determine if a process is considered interactive or batch. If a process satisfies Equation 3.3, it is considered interactive.

$$\text{dynamic priority} \leq 3 \times \text{static priority}/4 + 28 \tag{3.3}$$

A real-time process has an attribute called real-time priority with a legal range from 1 (highest priority) to 99 (lowest priority). Their main difference from conventional processes is that a real-time process inhibits the execution of every lower-priority process while it remains runnable [12]. Nevertheless, if several real-time process with the same priority are ready to run at same time, the scheduler will choose the process that occurs first in the runnable queue. Hence, there is no time reservation and no guarantee that a process will be scheduled at a certain time. The user must explicitly set the process to the real-time state using the system calls sched_setparam() and sched_setscheduler().

3.2.3 KVM-based proposals

As previously explained, a VM in KVM is treated as a Linux process, hence, any real-time improvement to the Linux host scheduler will translate in real-time improvement to the VM. Yet, other approaches are possible such as adding hypercalls to allow a guest OS to dynamically boost the priority of its VCPUs or implement a hierarchical scheduling algorithm scheme in both the host and guest OSs.

Kiszka [41] analyzed the KVM real-time responsiveness in two situations:

- Qemu/KVM running onto a Linux kernel with CONFIG_PREEMPT, which is a compilation flag that allows the Linux kernel to be preemptive resulting in better responsiveness;
- The same configuration plus the PREEMPT-RT kernel patch. PREEMPT-RT is a kernel patch that improves the Linux kernel real-time responsiveness.

Based on the results, the authors proposed real-time improvements to KVM given higher priority to the qemu-kvm thread and to the desired VCPUs at the expense of giving lower priorities to other Linux processes. Additionally, a para-virtualized scheduling approach was implemented consisting of two hypercalls: set scheduling parameters (used to change a VCPU priority) and, interrupt done (used to indicate that its interrupt handling is done and no reschedule is required). These hypercalls allow the guest to adjust its priorities according to the task currently running. Nevertheless, this implies that KVM cannot be executed as a strict full-virtualization system. The results showed that when using PREEMPT-RT kernel patch, the guest can achieve better determinism and sub-millisecond scheduling latencies. Yet„ the hypercall scheme introduced up to 25% higher average latencies.The authors claim that this impact could be reduced by optimizing their approach.

Cucinotta et al. [20] proposed the use of mechanisms designed according to the theory of hierarchical scheduling on real-time systems aiming to enhance predictability of the temporal behavior of VMs. They offered two different approaches: fixed priority (FP) inter-VM scheduling and reservation-based inter-VM scheduling. The first approach considers that the hierarchy is FP/FP, i.e, fixed-priority scheduling in both the hypervisor and guest OS scheduler. In the second approach, they implemented a variant of the Constant Bandwidth Server (CBS) scheduler algorithm [1]. Introduced by [58], this variant consists of a modification to ensure hard time reservation behavior at the hypervisor level and keeping the fixed-priority scheduling in the guest OS. The authors used KVM to validate their approach without considering the influence of the virtualized I/O. The results show that the hierarchical real-time scheduling theory may be effectively applied to virtualized systems. In [79], Cucinotta et al. addressed the problem of providing network response guarantees to VMs on multicore hosts. Their proposal is focused in single-core VMs scheduled according to a partitioned EDF policy, meaning that each VM is pinned to a certain CPU. The EDF scheduler algorithm provides time isolation between VMs resulting in a significant delay time when asynchronous events occur, e.g., when a network packet is received and the VM is in the non-responsive time frame. In order to avoid delays imposed by the EDF scheduler and improve the responsiveness for network traffic, the authors modified the original EDF algorithm adding spare time to the VMs. The spare time is used by a VM when it exhausted its time budget, thus, the VM is allowed to execute for a short time being able to receive and eventually respond to network traffic. The results show a significant response time increase for network traffic.

Zuo et al. [94] studied the behavior of a virtualized RTOS running on KVM. A way to minimize the average latency on the RTOS is to increase its priority over the other Linux processes. This approach affects negatively the whole CPU throughput. However, the authors claim that usually the execution of RT tasks is short or periodic, thus, the RTOS do not always need to have higher priority. Based on this affirmation, hypercalls are introduced allowing the guest to inform its priority to the host. Thus, if a guest is going to enter a time-critical execution, it informs the host about the desired scheduling policy and priority. When no more time-critical execution necessary, the guest informs to the host to lower its priority.

RESCH (Real-time Scheduler) framework [40] is a loadable kernel module (RESCH core) and a user library (RESCH library) that allows for the implementation of new scheduling algorithms within Linux without modifying the kernel. Asberg et al. [7] implemented a hierarchical scheduler using RESCH and applied it to both host and Linux guest. As result, it is possible to have a two layer scheduler approach that does not require kernel modifications. Instead, only RESCH kernel and libraries must be installed on both the host and guest. The advantage of this approach is that it is not restricted to KVM, i.e, it can be applied at any other virtualization layer, like VirtualBox [5]. The authors claim that their approach can improve performance of multimedia intensive applications, because the CPU availability for video and audio can be better controlled.

Yunfang et al. [76] proposes the KVM-Loongson, a virtualization solution for MIPS based on KVM to the Loongson-3A [47] processor. This processor do not implement the recently released MIPS virtualization extensions. Nonetheless, MIPS without the VZ module does not allow full-virtualization because it cannot support complete virtualization of kernel virtual address space. A possible solution targeting full-virtualization is to modify the processor core as presented at [2]. However, the authors modified the KVM (primarily designed for full-virtualization) to support para-virtualization. The authors performed an intensive study to determine all Linux kernel privileged instructions that could be substituted by hypercalls. As a result, they showed that about 98.6% of the privileged instructions can be substituted. Still, their memory virtualization overhead is about 4% on average. The main disadvantage of their technique is the effort to para-virtualize the Linux kernel and the impossibility to support proprietary software.

Dall and Nieh [22] proposed a KVM port to the ARM architecture with virtualization extensions, called KVM/ARM. This architecture takes the advantage of the wide Linux hardware support to the ARM family to simplify the hypervisor development and maintenance. It became the standard ARM hypervisor for Linux platforms. Experimental results showed an

[5]https://www.virtualbox.org/

average of 10% overhead when compared to native execution and significantly lower overhead when compared to KVM x86 virtualization. KVM was designed for general-purpose computing. Therefore, it does not comply with ES constraints like small footprint and real-time. For example, KVM/ARM requires a native Linux as host increasing the minimal system memory footprint. Additionally, KVM/ARM relies on the Linux OS scheduler which means it is difficult to improve the real-time responsiveness on virtualized OSes.

Zhang et al. [92] used the KVM/ARM to explore virtualization with a promising kind of main memory: non-volatile random access memory (NVRAM). This technology has attractive features, such as high density and low standby power. However, it has a long write latency that slow down the performance of write-intensive applications. Thus, time-critical tasks may result in unacceptable performance. The authors focused on improving the embedded virtualization with NVRAM/DRAM hybrid main memory. They proposed the NV-CFS, an optimized scheduling method that includes two main elements: a task allocator and a VM scheduler. The task allocator place write-intensive tasks in DRAM and read-intensive tasks in the NVRAM. The VM scheduler improves the system performance by giving a higher priority to foreground VMs. Thus, focusing on the user experience. Their approach can improve the performance by at least 30% when compared to the same system without NVRAM. In addition, they claim a reduction on task deadline misses ratio compared to the original scheduling algorithm.

3.3 SPUMONE

Kanda et al. [39] proposed a lightweight virtualization layer for embedded systems called SPUMONE (acronym for Software Processing Unit, Multiplexing ONE into two). SPUMONE builds a hybrid operating system environment composed by a RTOS running in parallel with a GPOS, i.e., multiplexing the CPU between the OSs. SPUMONE's architecture is shown in Figure 3.3. It was designed to address three main goals:

- minimal code modification on the guest OS, since it uses the para-virtualization technique;
- the hypervisor should be as light as possible;
- reboot the guest OSs independently from each other.

SPUMONE supports just the SH-4A [63] architecture and virtualizes only the CPU, i.e., it does not implement peripheral virtualization (a peripheral must be mapped directly to a guest OS). In order to minimize the engineering cost of modifying the guest OS and improve performance, both the hypervisor and guest OSs run in privileged mode. Thus, most of the privileged instructions are executed directly and just minor instructions are replaced by hypercalls. The GPOS can provide isolation between user applications, since they run in the unprivileged mode. Nevertheless, the hypervisor, RTOS and GPOS run in the privileged mode without memory isolation. A fault in one of these three software components may compromise the entire system.

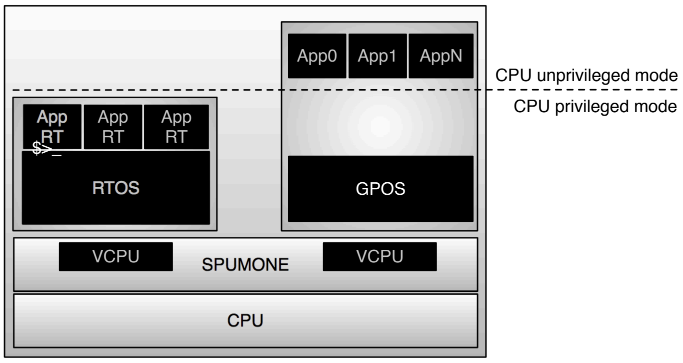


Figure 3.3: SPUMONE's single-core Architecture. Adapted from [39].

Mitake et al. [51] proposed a distributed multicore architecture for SPUMONE as depicted in the Figure 3.4. Each CPU runs a SPUMONE instance and all guest kernels run at the privileged level. Each instance keeps a local memory area for data accessible only from local CPU. The authors claim that this design can reduce the intrusion risks at the virtualization layer, because a hypervisor instance is limited to one CPU. Additionally, this reduces the need of shared structures and synchronization among instances increasing the scalability of the system.

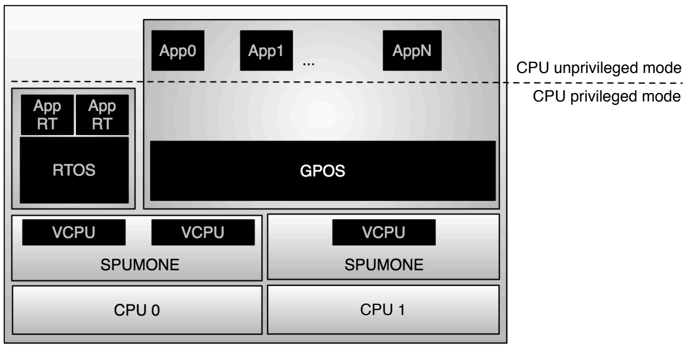


Figure 3.4: SPUMONE's multicore Architecture. Adapted from [51].

In both SPUMONE versions (single and multicore), the authors chose better performance over reliability. Both RTOS and GPOS are complex software and can fail. For example, a faulty AppRT on the RTOS may compromise the whole system, because the hypervisor is at the same privilege level as the guest OS kernels. On the other hand, this approach decreases the number of hypercalls implemented in the guest OS, improving performance and maintainability.

In addition to the previously mentioned works, Mitake et al. [52] proposed two new techniques to avoid the LHP problem (see Subsection 2.4.2) in real-time virtualized systems using SPUMONE. These techniques rely on the VCPU migration among physical cores. The first technique called trap based migration consists in does not allow Linux to execute the kernel code on core 0 (see Figure 3.4). As the LHP is caused by the kernel's spinlocks, performing Linux kernel only on core 1 guarantee that the RTOS is performing concurrently with the Linux (see Figure 3.4). The second technique called on-demand migration consists in to migrate Linux from core 0 to core 1 when the RTOS executes on core 0 avoiding to execute the Linux and the RTOS on the same physical core. If the RTOS is addle, the Linux can perform on both cores.

3.4 XtratuM

Crespo et al. [19] presented the XtratuM, a type 1 hypervisor specially designed for real-time embedded systems to meet temporal and spatial requirements of safety critical systems. In order to better fit real-time constraints, XtratuM was developed with the following a set of requisites:

- data structures are static to allow for better control over the resources being used;
- XtratuM's code is non-preemptive to make the code simpler and faster;
- all hypercalls are deterministic;
- peripherals are managed by the VMs (directly mapped devices);
- interrupt occurrence isolation (when a VM is in execution only the interrupts related to it are enabled).

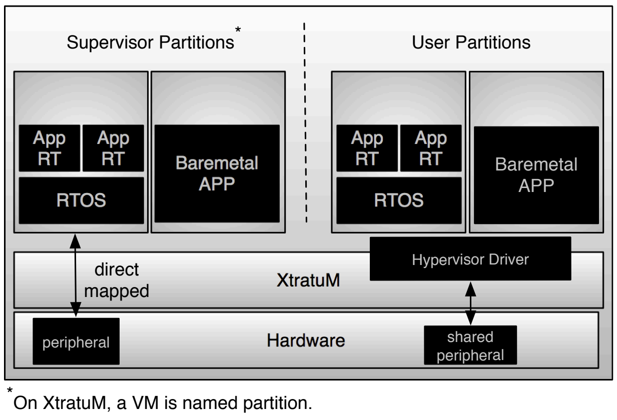


Figure 3.5: XtratuM's architecture. It is a type 1 hypervisor designed for safety critical systems. Adapted from [19].

Initially developed over x86, XtratuM now supports LEON2, LEON3, LEON4 (SPARC-V8 RISC [71]) and ARM processor architectures. The hypervisor's overall architecture is shown in Figure 3.5. In XtratuM, each VM is called a partition and each partition can support a RTOS or a bare metal application (GPOSs are not supported). There are two types of partitions: normal and system. Normal partitions have restricted functionality while system partitions can manage and monitor the state of the system and other partitions. Unlike SPUMONE, XtratuM virtualizes not just the CPU and interrupts, but also some specific peripherals using para-virtualization. If there is no need to share a peripheral between two or more partitions, it can be directly mapped to the specific partition. When a peripheral needs to be shared between partitions, XtratuM implements a device driver to serialize the access and the guest OS must be modified in order to implement hypercalls to this device driver.

In addition to previous work, Trujillo et al. [80] proposed the MultiPARTES: a project to support mixed critically integration for embedded systems based on virtualization techniques. It uses the XtratuM for virtualization purposes. MultiPARTES project resulted in a tool that allows for the application description using UML and additional annotations, like CPU time, memory, or bandwidth. The tool uses these descriptions to define a system partitioning and generate three outcome: source code, XtratuM configuration files, and system generation files. The authors claim that their approach accelerates the certification process for mixed-critical systems.

3.5 L4 Microkernel Family based hypervisors

The microkernel concept reduces the OS kernel code to fundamental mechanisms and implements the remainder of the system services at the user level [45]. The L4 microkernel was based on the L3 OS [44] developed by Liedtke to the x86 processor architecture. During his work at GDM and IBM, Liedtke modified the L3 to implement and test new ideas resulting in the L4. This initial development triggered an entire family of L4 based microkernels. Figure 3.6 summarizes the L4 microkernels family tree that resulted in embedded hypervisors.

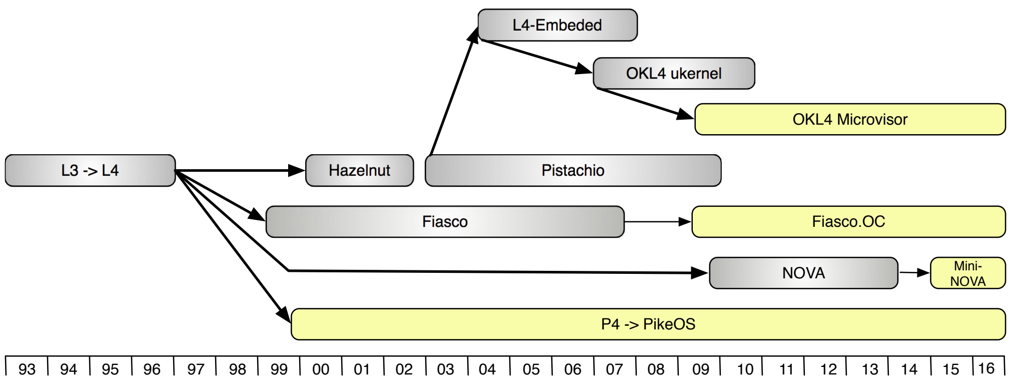


Figure 3.6: Summarized L4 microkernel family tree that resulted in embedded hypervisors. Adapted from [25].

GDM and IBM imposed a restrictive intellectual propriety for other researchers, forcing the University of Dresden to implement an x86 version of the L4 from scratch, called Fiasco. It was later renamed to Fiasco.OC. Liedtke moved to the University of Karlsruhe where he developed a version of the L4 supporting Pentium and ARM architectures, called Hazelnut. In 2001, it was implemented a new open-source kernel, called Pistachio, based on the previous version of L4. This kernel was written to x86 and PowerPC. The NICTA Laboratory and the University of South Wales ported it to MIPS, Alpha, 64-bit PowerPC and ARM. NICTA created a fork of Pistachio, called L4-embedded, that was adopted by Qualcomm as an RTOS on wireless modems. The NICTA work resulted in a spin-off company called Open Kernel Labs for further support and development of the kernel. This company renamed L4-embedded OKL4 and developed the OKL4 Microvisor for virtualization purposes. PikeOS is another commercial clone of the original L4 that is certified for use in safety-critical avionics. NOVA is a open-source hypervisor based on L4 principles and designed for hardware-assisted virtualization on x86 platforms. Mini-NOVA is a port of the NOVA to the ARM Cortex-A9 using para-virtualization due to the absence of hardware support for virtualization on this processor.

3.5.1 OKL4 Microvisor

The OKL4 Microvisor [32] was developed targeting mobile devices. Currently, it supports Linux, Android, Symbian and Windows Mobile OSs. The OKL4 powered the first commercial virtualized phone in 2009, named Motorola Evoke QA4. It executed two VMs on top of the OKL4 Microvisor: a Linux guest to handle the user interface and another guest for the BREW (Binary Runtime Environment for Wireless).

Figure 3.7 shows OKL4's overall architecture. The microvisor is the only software layer that executes in the privileged mode while GPOSs, bare metal applications and even device drivers execute in the unprivileged mode. It has the ability to execute several GPOSs and bare metal applications concurrently.

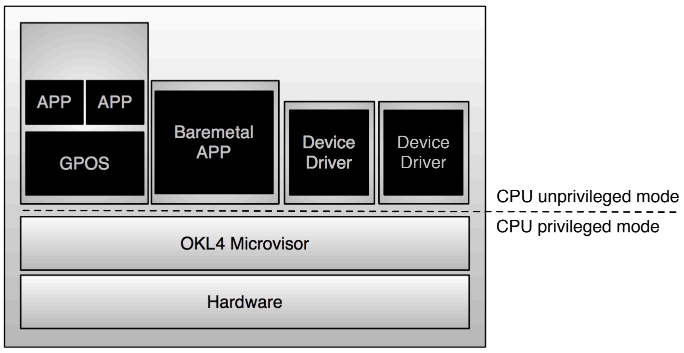


Figure 3.7: OKL4's architecture. It was developed targeting mobile devices supporting several mobile OSs, and it powered the first commercial virtualized phone in 2009.

With OKL4 it is possible to execute both GPOS and a real-time environment on a single ARM processor, while providing high performance communication between them. This eliminates the need for a second ARM processor, which reduces the final cost and design complexity.

3.5.2 L4/Fiasco

The L4/Fiasco was developed as a clone of the original L4 kernel and adapted to provide virtualization on single-core x86 processors. Figure 3.8 shows L4/Fiasco's architecture. L4Linux is a modified Linux kernel to support the L4/Fiasco's para-virtualization mechanism. It is mainly different from other hypervisors due to its ability to map Linux kernel threads directly onto VCPUs. The L4Linux invokes hypercalls during kernel thread creation to allow to the L4/Fiasco to create VCPUs. However, this mechanism does not avoid the two-level scheduling problem because the Linux scheduler keeps mapping processes to Linux kernel threads. As is the case with other microkernels, device drivers are executed in user land. Additionally, L4/Fiasco supports RTOSs.

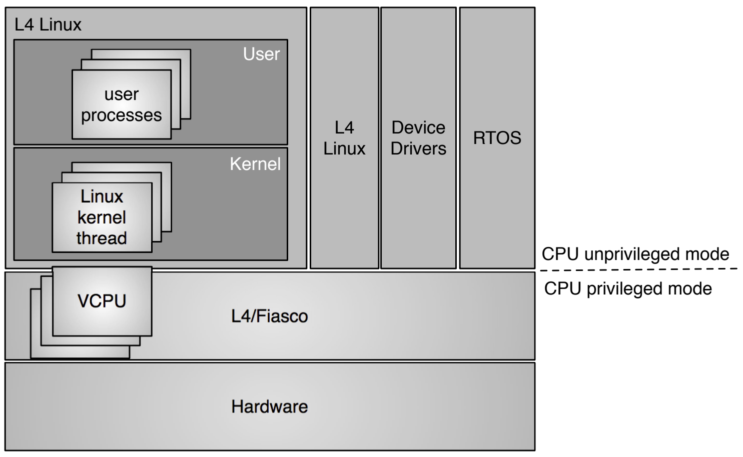


Figure 3.8: Fiasco's architecture. The L4/Fiasco maps the L4Linux kernel threads directly to VCPUs.

Jungwoo et al. [88] proposed a compositional scheduling framework to deal with the two-level scheduling problem and extended L4/Fiasco to support real-time constraints. L4Linux was modified to obtain the timing requirements from its internal processes and send them to the hypervisor through hypercalls. L4/Fiasco's scheduler implements a periodic task model and it uses the timing information from the guest OSs to meet the time constraints. This approach drastically reduces the number of deadline misses when compared to the original implementation. Additionally, it adds a small overhead that increases with the number of threads.

3.5.3 PikeOS

PikeOS [38] is a proprietary microkernel developed by SYSGO AG[6] with support for virtualization. It was designed for safety-critical systems by implementing the concept of partitioning as defined by the ARINC 653 standard [4]. Additionally, the standard determines the CPU time allocation across the partitions. The approach consists of a fixed cycle time in a specified order for a guaranteed duration. This allows for real-time guarantees, but it leads to a poor CPU utilization. The microkernel has approximately 6,000 lines of code. It traps all interrupts and privileged instructions. Thus, it implements para-virtualization. Currently, it supports x86, PowerPCs and MIPS processors.

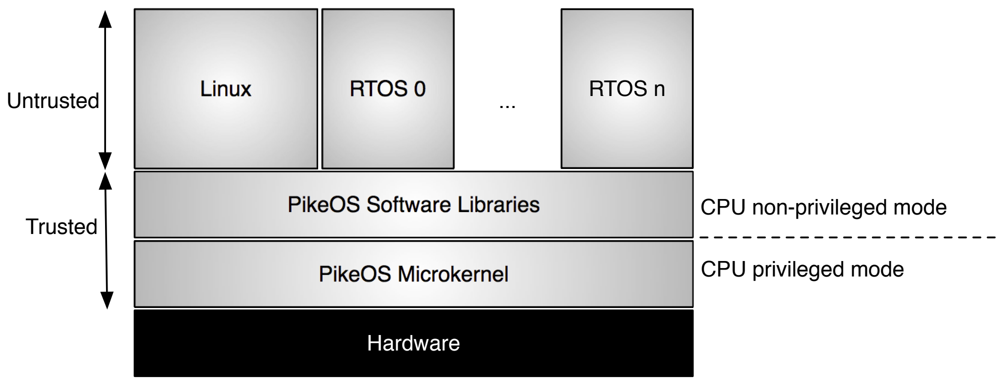


Figure 3.9: PikeOS's architecture. It is a proprietary microkernel with support for virtualization designed to attend safety-critical systems.

Figure 3.9 depicts the architecture of the PikeOS. The microkernel is the only software executed in CPU's kernel mode. The PikeOS software layers establish a set of partitions or VMs. Each partition can host a para-virtualized OS. The guest OSs are considered untrusted software and are kept isolated from each other. Finally, PikeOS supports inter-VM communication between guest OSs based on a mechanism defined by the ARINC standard. In this mechanism, the microkernel acts as a communication arbiter to manage message exchanges.

[6]https://www.sysgo.com/

3.5.4 Mini-Nova

In 2010, Steinberg et al. [73] proposed a hypervisor supporting the VT-x extensions on x86 architecture, called NOVA. Based on the NOVA hypervisor, Xia et al. [86] suggested the use of the NOVA microkernel on an ARM-FPGA platform capable of managing reconfigurable hardware parts dynamically, but without support for virtualization. In the following work, Xia et al. [87] presented the Mini-NOVA, a type 1 embedded hypervisor for the ARM architecture using para-virtualization due to the absence of hardware support.

The main difference in their approach is to manage the coexistence of software and hardware computing resources. Thus, Mini-NOVA provides support for dynamic partial reconfiguration (DPR) to hardware tasks. According to the authors, the major challenge of this approach is to coordinate the hardware resources efficiently among the separate VMs. Security is another major concern, because the hardware tasks are shared among the guests. The solution adopts a centralized technique where a hardware task manager performs as a hypervisor service that manages the hardware. If a guest needs to execute a hardware task, it must require it using hypercalls. The evaluation demonstrated that the hardware can be efficiently managed with a short response latency and with a small overall latency.

3.6 M-Hypervisor

The M-Hypervisor [93] is a type 1 hypervisor supporting the Loongson 2f [74] MIPS based processor. It is implemented by referencing the XtratuM's principles (see Section 3.4). Para-virtualization is adopted because Loongson 2f does not support the newer MIPS-VZ module. Hardware virtualization support is expected to be seen in the next generation of Loongson. Similar to XtratuM, the VMs are called partitions and support different OSs, like Linux and RTOSs. Real-time is supported through bare metal applications. The M-Hypervisor provides virtualization of the timer, interrupt, memory management, process switching and scheduling to ensure the real-time performance of high-level applications in the partitions and also temporal and spatial isolation. Based on the benchmarks, the authors claim that the M-Hypervisor can guarantee accurate real-time performance.

3.7 Xvisor

Xvisor, shown in Figure 3.10, is an embedded hypervisor that supports both full-virtualization and para-virtualization[59]. It supports ARM virtualization extensions to provide full-virtualization and para-virtualization through optional VirtIO compatible device drivers. It can map interrupts directly to guests, allowing guest interrupts to be handled without the intervention of the hypervisor. Additionally, it provides memory isolation between hypervisor, guests and guest applications using the third privileged level from ARM's virtualization support. However, Xvisor has a poor support for MIPS processors since it only supports para-virtualization on the MIPS 24k processor model under the Qemu emulator.

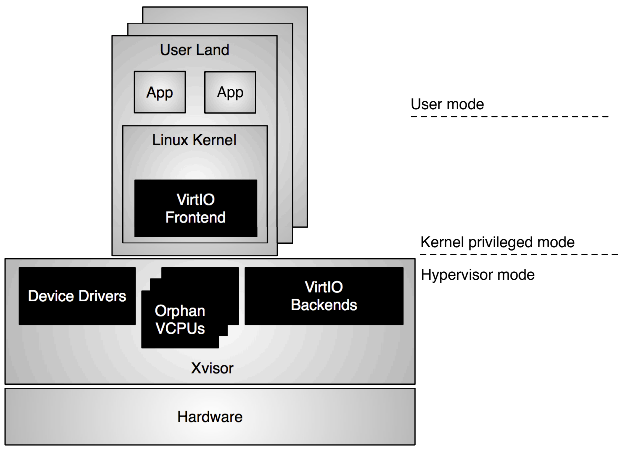


Figure 3.10: Xvisor's architecture. The Xvisor hypervisor uses the third privileged level from ARM's architecture.

Xvisor implements the concept of Orphan VCPUs that executes background processing for device drivers and management purposes with the highest privilege. When compared to Xen or KVM, Xvisor has the advantage of having a single software layer where all virtualization related services are provided. For example, KVM depends on Qemu for device emulation and Xen depends on dom0 for I/O. Thus, the Xvisor's context switches are lightweight, resulting in instruction trap handling, host interrupts and guest I/O events. Also, the Xvisor's scheduler is per-CPU and does not do load balancing for multiprocessor systems. Instead, load balancing is performed by a distinct entity in Xvisor.

3.8 Embedded Hypervisors Comparison

Table 3.1 is a comparison of the previously described embedded hypervisors and their key features.

Table 3.1: Embedded hypervisors' comparative table.

Hypervisor	Supported OSs	Architecture	Virtualization technique	Native Real-time support[7]
Xen	Linux, Windows	ARM	PV, FV	No
KVM	Linux, Windows	x86, ARM	PV, FV	No
SPUMONE	Linux	SuperH-4A	PV	Yes
XtratuM	RTOSs	LEONx, ARM,	PV	Yes
OKL4	RTOSs, Linux, Android	ARM, MIPS, x86	PV	Yes
L4/Fiasco	Linux	x86(single-core)	PV	No
PikeOS	RTOSs	x86, PowerPC, MIPS	PV	Yes
Mini-NOVA	Linux, RTOSs	ARM	PV	Yes
M-Hypervisor	Linux, RTOSs	MIPS	PV	Yes
Xvisor	Linux, RTOSs	ARM	PV/FV	Yes

The Xen hypervisor was developed for enterprise and desktop applications, as explained in Section 3.1. However, recently it was ported to ARM processors. It does not support real-time natively. Instead, it requires modifications to improves the real-time responsiveness. Similar to Xen, KVM was recently ported to ARM processors and requires additional improvements in the Linux kernel to support real-time. All the other hypervisors presented were developed for embedded virtualization. However, with the exception of Xvisor, they were the first generation of embedded hypervisors. Thus, they share two important characteristics: lack of support for full-virtualization and lack of memory isolation between the guest's kernel and applications. Yet it is important to note that in SPUMONE the guest's kernel is executed in privileged mode along the hypervisor. Thus, it can keep the user applications isolated from each other, but the guest's kernel can access the hypervisor's memory space. Xvisor was designed to take advantage of the ARM virtualization support. Thus, it supports both full and para-virtualization and provides memory isolation between the hypervisor, guest kernel and guest applications.

[7]Xen, KVM and L4/Fiasco can support real-time through modifications.

Tables 3.2 and 3.3 compare the footprint of the hypervisors studied. Xen and KVM require the installation of additional elements to perform virtualization. The installations column in Table 3.2 shows the different elements that must be installed with these hypervisors and their footprint. For example, Xen requires the Xen kernel, Dom0 and the Toolstack while KVM requires the Linux kernel and QEMU. Table 3.3 compares the footprint of the remaining hypervisors.

Table 3.2: Footprint requirements for Xen and KVM.

	Installations				
Xen	**Xen Kernel** 600KB >	**Dom0** 3-4MB	**Toolstack** 20MB <	**RAM** 180MB>	**footprint** 204MB<
KVM	**Host Linux zImage** 3-4MB	**QEMU** 20MB <		20MB >	**footprint** 44MB<

Xen has a total footprint of roughly 204MB. KVM has a smaller footprint, but it requires a host Linux to execute. When compared to the configuration of modern computer servers, these requirements are modest. However, though they are used in embedded systems, their footprint may be unacceptable for many devices.

Table 3.3: Footprint requirements of the hypervisors when available.

	kernel size	**RAM Memory**	**footprint**
SPUMONE	30KB	-	>30KB[8]
Xtratum	8KB	16KB	24KB
OKL4	48KB	78KB	126KB
Xvisor	1-2MB	4-16MB	5-18MB
Mini-NOVA	40KB	20MB	20.04MB

Hypervisors developed for embedded systems typically present a smaller footprint than hypervisors for server virtualization. Unfortunately, some authors do not mention the memory requirement of their hypervisors. This is the case for L4/Fiasco, PikeOs and M-Hypervisor. The remaining of the hypervisors are shown in Table 3.3. Note that the footprint and RAM requirements can be quite different. For example, Xvisor has a footprint between 5 and 18MB. OKL4 has a footprint of only 126KB. SPUMONE's kernel requires 30KB each

[8]The total memory needed by SPUMONE during its execution is unknown.

core in its multicore version. For example, in a dual-core processor SPUMONE will require 60KB. However, the amount of memory needed during its execution is unknown. Xtratum has the smallest footprint, requiring only 24KB during its execution.

3.9 RT-Linux

RT-Linux [18] provides the capability of executing bare-metal tasks along a Linux OS. It is a kernel patch that adds a software layer with a real-time scheduler. From the point of view of the RT-Linux layer, the Linux kernel is a process that shares the processor with real-time tasks. Figure 3.11 shows RT-Linux's architecture. The I/O and interrupts subsystems of the Linux kernel are modified to communicate with the RT-Linux layer. This is similar to a para-virtualized hypervisor. The bare-metal tasks have priority over the Linux kernel and can access peripherals directly. Additionally, the Linux kernel will always be pre-empted to give priority to the real-time tasks.

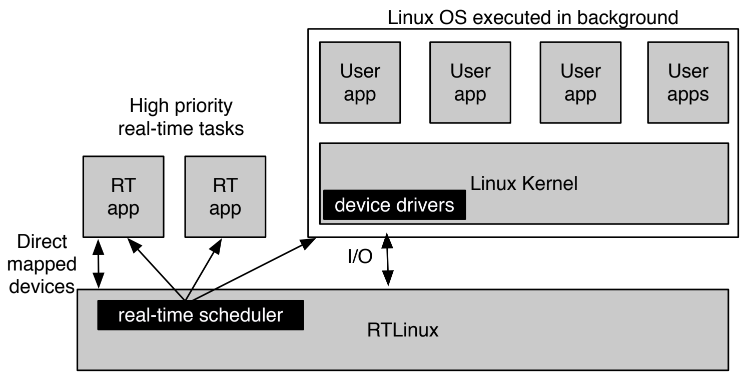


Figure 3.11: RT-Linux is a kernel patch that adds a software layer with a real-time scheduler and it makes the Linux kernel to share the processor with real-time tasks.

RT-Linux's architecture is similar to a lightweight hypervisor, but without the capability of supporting multiple guest OSs. In fact, its objective is to improve the responsiveness of real-time tasks while maintaining the Linux software stack. The real-time tasks must be written using the RT-Linux API. Finally, shared memory can be used for communication between real-time tasks the Linux threads.

3.10 Final Considerations

This chapter presented the main hypervisors available for embedded virtualization. This study exposed the complexity of embedded virtualization and the wide range of possible solutions. It was identified that real-time is an issue addressed frequently, since all mentioned hypervisors have some real-time support. In respect to Xen and KVM, the primary focus of the studied works is the issue of real-time improvements. These hypervisors have a deep dependency on the Linux OS, which makes it difficult to reduce their memory requirements. Thus, a recurrent challenge for hypervisors specially designed for embedded systems is to achieve a small footprint. To accomplish this goal, they avoid dedicating domains for I/O (in contrast to Xen) and they implement the type 1 hypervisor model (unlike KVM). Also, the presented hypervisors regularly do not meet common embedded system requirements. For example, XtratuM does not support GPOSs while SPUMONE cannot provide spatial isolation between guests. L4 based hypervisors support GPOSs and provide spatial separation but do not support full-virtualization. Para-virtualization is extensively adopted on embedded hypervisors. As seen in Table 3.1, KVM and Xvisor are the only embedded hypervisors that support full-virtualization, but their support is limited to the ARM virtualization extensions. Chapter 2 showed that hardware-assisted virtualization is a trend in embedded processors. Thus, it is inevitable for the further development on hypervisors to adopt full-virtualization even when combined with para-virtualization. This study makes it clear that an novel embedded virtualization model is required to take advantage of newer processors and to better fit embedded system needs.

4. A VIRTUALIZATION MODEL FOR EMBEDDED SYSTEMS

This chapter presents the virtualization model proposed by this dissertation. As discussed in Subsection 3.10, the techniques currently applied for virtualization on embedded environments do not meet all requirements. Additionally, the evolution of the embedded processors is growing towards the usage of hardware support for virtualization. However, this is ongoing, and the next generations of embedded processors will support new hardware features for virtualization. Thus, the existing virtualization techniques for ES require continuous improvement and new approaches must be proposed. Based on the state-of-the-art for embedded virtualization, a set of important features were chosen as a guideline for the development of embedded virtualization models. Section 4.1 discusses these features. Section 4.2 presents the proposed virtualization model. The Embedded Systems Group (GSE) started to research embedded virtualization in 2010. Since then, another virtualization model was proposed by Aguiar [3]. Thus, Section 4.3 compares the virtualization model presented in this dissertation to the model proposed by Aguiar [3]. Finally, Section 4.4 is the final considerations of this chapter.

4.1 Desirable Features for an Embedded Virtualization Model

This section states the important features for an embedded virtualization model. As explained before, real-time support on virtualized ESs has become an major concern, since all works presented in Chapter 3 have some support for real-time. However, these works still have different drawbacks. Widely adopted hypervisors like Xen and KVM (see Subsections 3.1 and 3.2) were designed for enterprise usage and do not scale well on some ESs because of their memory footprint. Solutions designed to target ESs, like SPUMONE, XtratuM and L4/Fiasco (see Sections 3.3, 3.4 and 3.5.2) have different constraints. For example, SPUMONE does not provide spatial isolation, XtratuM was designed for critical systems, therefore it does not support GPOSs, and L4/Fiasco was designed for only single-core processors and it does not have proper real-time support. Moreover, the most of the hypervisors do not make use of the current hardware-assisted support present in the modern embedded processors. The only exception is the Xvisor (Section 3.7), but its support is restrict to ARM processors. The research shows an increasing interest in emergent topics like hardware-assisted virtualization and security. Thus, it is essential to address these topics on a virtualization model.

Based on the related works, there is not a consolidated approach for embedded virtualization. Additionally, due to the wide diversity of ES architectures and applications, there is not a single approach that can overcome all challenges at once. Nevertheless, more accurate techniques can be applied to build a hypervisor able to better address the virtualization needs on the most of ESs. Thus, this dissertation suggests a virtualization model that supports the following major features:

- **Hardware-assisted virtualization**: Considering that virtualization will be widely adopted for ESs shortly, the manufacturers will provide hardware-assisted virtualization for their embedded CPUs. The decreased fabrication cost of the embedded processors allowed for the adoption of hardware features for virtualization support. In fact, ARM and MIPS have already specified their virtualization extensions as seen in [6] and [36]. Additionally, several manufacturers have released their virtualization platforms. Once the hardware supports a flexible set of features, hypervisor designers must decide what policy usage to adopt. As explained in Subsection 2.3.3, hardware-assisted virtualization allows more efficient and simpler hypervisors. The main advantages of the hardware-assisted virtualization are to speed-up hypervisor performance, save engineering costs and improve time to market.

- **Multicore support**: Multicore processors are already widespread in ES platforms. When available, the hypervisor should ensure the proper support to obtain maximum performance. Additionally, migration of VCPUs between physical cores should be considered to allow load balancing.

- **Real-Time support**: Real-time is an intrinsic characteristic of embedded systems. The hypervisor should be predictable and ensure temporal isolation between GPOS and real-time applications. For multimedia purposes, soft real-time support is enough. Nevertheless, some ESs require accurate time constraints to run with GPOSs. For example, a smartphone system that must support a rich user graphical interface still needs to deal with the GSM stack [5] time constraints. In this case, real-time support is desirable because it avoids the need for a second processor dedicated to the GSM stack.

- **Coexistence of multiple GPOSs and Real-time instances**: A GPOS is highly desirable for certain ESs, such as smartphones and set-up boxes, due to its wide diversity of software. However, usually GPOSs have poor real-time responsiveness. In fact,the hypervisor must guarantee real-time instances that are executed with GPOSs. Additionally, some virtualized platforms can require more than one GPOS at the same time. For example, in a virtualized network router that, for reliability reasons, the data plan must be isolated from the control plan.

- **Direct mapped and shared devices**: The current generation of embedded platforms designed for virtualization do not support I/O virtualization. Thus, when it is necessary to share a physical device, e.g. an Ethernet device, the hypervisor may use the para-virtualization technique. In this case, a device driver at hypervisor level serializes the access to the Ethernet device, and the guest OS implements a para-virtual device driver. However, aiming better performance, when device sharing is not desired, the hypervisor must map the device directly to the desired guest OS. Efficient implementation of directly mapped devices depends on the support of the processor's hardware (see Subsection 2.3.3).

- **Security**: The hypervisor must provide robust spatial isolation between VMs, i.e., a misbehavior in the guest OS should not affect the behavior of the other guest OSs or even the hypervisor. The goal of separation used for security purposes is to create and preserve a trusted operating environment for an embedded system. Separation is intended to prevent exploitable defects in one virtualized environment from propagating to adjacent virtual environments, or to the physical platform as a whole [61].

- **Inter-VM communication**: A virtualized system is composed by a set of VMs. Possibly, these VMs will require some level of interaction between them. Also, some applications can require secure communication channels for sensitive information. Thus, an efficient and secure inter-VM communication mechanism must be available in the hypervisor.

- **Lightweight virtualization layer**: Due to the resource constraints on ESs, the hypervisor layer must have a small memory footprint and induce the least amount of overhead possible. This requirement is intrinsically related to hardware-assisted virtualization. As explained in Section 2.3.3, greater hardware support simplifies hypervisor implementation and can improve performance.

The design of an embedded hypervisor should consider this features. However, some are difficult to measure. For example, if the hypervisor is lightweight, it cannot be measured directly, but it can only be compared regarding the lines of code and memory footprint of similar hypervisors. Security is another feature that is hard to evaluate. The simplest way to evaluate security is to consider if the hypervisor ensures spatial isolation or separation between VMs. However, critical systems may require formal proof. Other system characteristics are directly measured based on responsiveness, memory and time overhead. Section 4.2 presents the proposed virtualization model that is flexible enough to accommodate these features.

4.2 Model Overview

An innovative virtualization model was proposed based on the desired features described in the Section 4.1. As a result, an implementation of this model is presented in the Chapter 5. Thus, this section describes the proposed virtualization model. However, there are two considerations about this model:

1. The model does not try to impose or improve the real-time capabilities of non real-time entities, like a GPOS. Instead, it provides mechanisms for GPOSs and real-time entities coexist and;
2. The model was not designed to be a pure full-virtualization model. Instead, it was designed to support full-virtualization of the CPU and use the para-virtualization concept for other services, such as communication and shared devices.

Many hypervisor solutions for ESs try to improve the real-time responsiveness of GPOSs in virtualized environments. For example, different authors propose to apply techniques to improve the Linux real-time responsiveness when virtualized in KVM or Xen (see sections 3.2.3 and 3.1.3). However, the proposed model avoids the utilization of GPOSs as real-time instances. Instead, it allows real-time services to be directly scheduled by the hypervisor, see Subsection 4.2.3. Additionally, the model provides full-virtualization based on the hardware-assistance of the CPU. Other peripherals may require para-virtualization techniques, see Subsection 4.2.2.

Figure 4.1 depicts the proposed model. At the hardware level, it uses a bus-based homogeneous MPSoC along with shared memory. The first software layer is the the hypervisor, which is responsible for the creation and management of each VM. It also provides a logic arrangement for associating the VM with its VCPUs. Moreover, the hypervisor controls the memory space of each VM for memory isolation. Thus, this model describes a hypervisor type 1, as explained in Subsection 2.2.1. Type 1 hypervisors usually requires lower footprint then type 2 ones, since they does not require an underline OS.

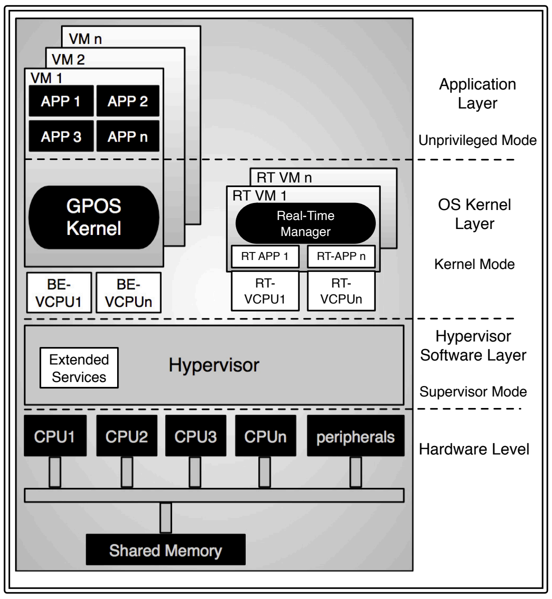


Figure 4.1: Overall view of the proposed virtualization model.

The GPOSs are mapped onto best-effort VCPUs while real-time instances are mapped onto real-time VCPUs, as further explained in Subsection 4.2.1. The model takes advantage of the processor's additional privilege levels (see Subsection 2.3.3) to provide spatial isolation between hypervisor and guests. Additionally, unlike virtualization models that do not utilize the additional privilege levels scheme, the proposed model can provide spatial isolation between the guest kernel and guest applications. Therefore, the hypervisor is protected from the actions of malicious or misbehaving guest OSs and the guest OSs are protected from their applications.

The model provides additional features, called extended services, that expand the virtualization platform functionalities. See Subsection 4.2.2 for more details. Additionally, for better temporal isolation, the model proposes the adoption of the RT-VM, which maps real-time services onto RT-VCPUs, as explained in Subsection 4.2.3. Finally, direct communication among VMs is possible using shared memory areas or message passing mechanisms can be implemented using the hypervisor as a communication arbiter, which is explained in detail in Subsection 4.2.4.

4.2.1 The Flexible Scheduling Mapping

Figure 4.2 shows the possible flexible mapping and the partitioning model for virtualized architectures based in the proposed model. To perform temporal isolation between VMs, the hypervisor specifies two different kinds of VCPUs: best-effort VCPUs (BE-VCPUs) and real-time VCPUs (RT-VCPUs). RT-VCPUs have priority over BE-VCPUs and follow the policy of a real-time scheduling algorithm. BE-VCPUs are scheduled by a best-effort scheduler algorithm that is invoked when there are not RT-VCPUs ready to execute. Additionally, the BE-VCPUs can migrate among physical cores, i.e., when a CPU becomes idle the scheduler will choose the next BE-VCPU ready to run in a global queue. The migration of RT-VCPUs depends on the real-time scheduler algorithm adopted. Due to time constraints, a real-time scheduler algorithm for multicore processors must be carefully designed, as proposed by [66] and [82]. This model is flexible enough to accommodate different combinations of real-time and best-effort scheduler algorithms, which is an implementation decision.

A multiprocessing GPOS may have several BE-VCPUs available, and its scheduler is responsible for mapping the task among BE-VCPUs according to its policies. At the hypervisor level, the best-effort scheduler algorithm will map a BE-VCPU to the next idle CPU. This arrangement enables an N-to-N mapping of task over BE-VCPUs and BE-VCPUs over CPUs.

4.2.2 Extended Services

The model suggests the implementation of the para-virtualization concept to provide extended services to the guest OSs. Extend services are useful to expand the virtualization platform functionalities, i.e., to implement functions that does not exist in a pure fully-virtualized system. Hypercalls are widely used in para-virtualization based approaches,

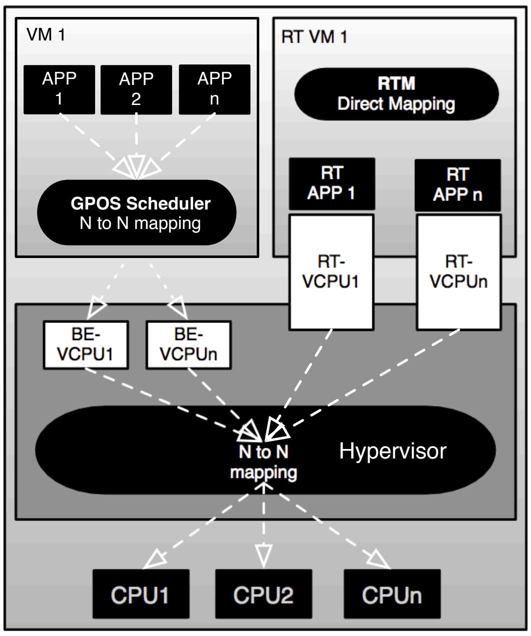


Figure 4.2: Flexible mapping for multiprocessor embedded systems with real-time support.

where the guest OS needs to be modified to invoke hypercalls instead of using privileged instructions. However, this work uses full-virtualization with a hardware-assisted technique, where the guest OS does not need to be modified for virtualization purposes. Yet it must use hypercalls to take advantage of the extended services, e.g., real-time services and communication among VMs.

The model suggests a minimal set of extended services. However, an implementation can add services for different purposes as necessary. Table 4.1 resumes the extended services available through hypercalls. Altogether, six different hypercalls are supported and may be implemented by a guest OS when needed. The services are divided into three different groups:

- VM identification composed by a hypercall that returns a unique VM identification number issued by the hypervisor.
- RT-VCPU management is composed by a group of three hypercalls responsible to manage the RT-VCPUs. Using this hypercalls the guest OS can create, launch or delete real-time services.
- Communication services are composed of two hypercalls designed for communication purposes.

Table 4.1: Hypercalls as extended services.

Extended Service	Hypercall	Description
VM identification	HCALL_INFO_GET_ID	Return the VM ID number
RT-VCPU Management	HCALL_RT_CREATE_APP	RT application Instantiation
	HCALL_RT_LAUNCH_APP	RT application launch
	HCALL_RT_DELETE_APP	RT application delete
Communication Services	HCALL_IPC_SEND_MSG	Send messages
	HCALL_IPC_RECV_MSG	Receive messages

The utilization of the hypercalls for management of the RT-VCPUs is better explained in Subsection 4.2.3. The extended services proposed in Table 4.1 are supported by the hypervisor implementation presented in Chapter 5.

4.2.3 Real-time Aspects

To deal with real-time constraints, a special VM called RT-VM was designed to provide a robust temporal isolation between real-time services and GPOSs. The RT-VM does not support RTOSs. Instead, it implements what is called a Real-Time Manager (RTM), which supports communication facilities and basic user libraries. The RTM can map its tasks directly to RT-VCPUs in the hypervisor scheduler, i.e., it does not implement a scheduler on its level avoiding the hierarchical scheduling problem and improving performance. The real-time services must be registered during the RT-VM initialization calling the HCALL_RT_CREATE_APP hypercall. After the registration, the real-time services are available in the system and they can be started/stopped by the guests using the hypercalls HCALL_RT_LAUNCH_APP and HCALL_RT_DELETE_APP.

RTM may or not provide spatial isolation between tasks. Providing spatial isolation means the implementation of virtual memory support on the RTM, which may impact in the real-time responsiveness. Nevertheless, if spatial isolation between different services is desired, more than one RT-VM can be instantiated. When a developer is interested in taking advantage of real-time in this model, he must implement its real-time service using the RTM API.

The RT-VCPUs are scheduled by a real-time algorithm, e.g., EDF, which has priority over the best-effort scheduling algorithm. To keep the RT-VCPUs in a global queue (allowing migration) or in a local queue per CPU is a implementation decision. As described in Subsection 4.2.1, an implementation for multi-core processor using a global queue and allowing CPU migration must be carefully designed to guarantee temporal isolation. Additionally, critical systems require schedulability analysis to guarantee system resource availability during RT-VCPU admission. Finally, during the execution of a RT-VCPU, interrupts sent to BE-VCPUs may or not be postponed in order to increase temporal isolation, see Section 5.5 about interrupt delivery policies.

Schedulability Analysis for Real-time Services

As explained, the real-time services are available in the RT-VM, and they can be started/stopped by the guest OSs. An important concern about this scheme is how to deal with concurrent real-time services that can be started asynchronously. For example, a set of real-time services available in a RT-VM may not be schedulable altogether. Thus, some of the real-time services may not be accepted for execution when a guest OS requires it due to unavailable CPU time. This is especially critical for systems where the non-execution of a service may cause safety concerns.

The model does not specify mechanisms for schedulability analysis or any other approach to avoid the schedulability problem. In fact, to determine the schedulability of a task set, the real-time scheduling algorithm and the real-time scheduling parameters must be known. However, the model allows for different real-time algorithms making the schedulability analysis dependant of implementation. Finally, the simpler approach for a hypervisor based on this model to avoid schedulability problems is to guarantee the schedulability of the set of real-time services available in the system. For this, two approaches can be take:

- The development team performs a schedulability analysis;
- The hypervisor implements its own schedulability test, that is performed during its initialization.

Both approaches can be used together. If the development team fails in their schedulability analysis, the hypervisor can detect the problem during the system's initialization, since the RT-VM must register its services at boot time using the hypercall HCALL_RT_CREATE_APP. It is important to highlight that both approaches work only for real-time services implemented at design time. However, the model does not permit to install new real-time services in runtime. The hypervisor implementation presented in Chapter 5 implements the schedulability test to the EDF scheduler algorithm.

4.2.4 Communication Model

Inter-VM communication capabilities must be implemented as part of the extended services, i.e., using hypercalls. However, the model does not impose any specific communication mechanism. Instead, this decision is made by the developer. For example, for better performance, the designers may choose to implement direct communication between VMs using memory sharing. Or, if the major concern is security, the designers may choose to implement a message exchange mechanism using the hypervisor as a communication arbiter. The model's implementation presented in Section 5 provides communication between VMs using the hypervisor as a manager.

Inter-device communication, i.e., communication of the virtualized platform with external devices is desirable in modern ESs. The proposed model is flexible enough to support typical socket style [77] communication among VMs. Thus, the hypervisor may implement a network layer in order to route Internet Protocol (IP) datagrams [77] between VMs. Still, when communication with the external world is desired, the hypervisor may virtualize a network adapter providing network capabilities to the platform. Nevertheless, in the embedded system context, the socket communication scheme can be expensive in terms of system overhead, therefore message passing may be preferable.

4.3 Comparison with the Virtual Hellfire Hypervisor (VHH)

In 2010, the GSE group started to research virtualization for embedded systems. Aguiar [3], proposed an embedded virtualization model and a hypervisor implementation in 2013. The resulting hypervisor was called Virtual Hellfire Hypervisor (VHH). This section discusses the main differences between the virtualization model proposed by Aguiar and the model presented in this dissertation.

Figure 4.3 presents the VHH's virtualization model. Similar to the model proposed in this dissertation, the VHH model assumes a bus-based homogeneous MPSoC along with shared memory. On top of the CPUs, it executes the hypervisor, which is responsible for the creation and management of each Application Domain Unit (ADU). The ADU is a logical arrangement responsible for associating the guest OS with its VCPUs. Moreover, each ADU has its own memory space controlled by the hypervisor to provide memory isolation. Applications can be mapped on best-effort (non-real-time) and real-time VCPUs according to their needs in an ADU. The guest OS is responsible for instantiating RT-VCPUs for each real-time application it wishes to execute.

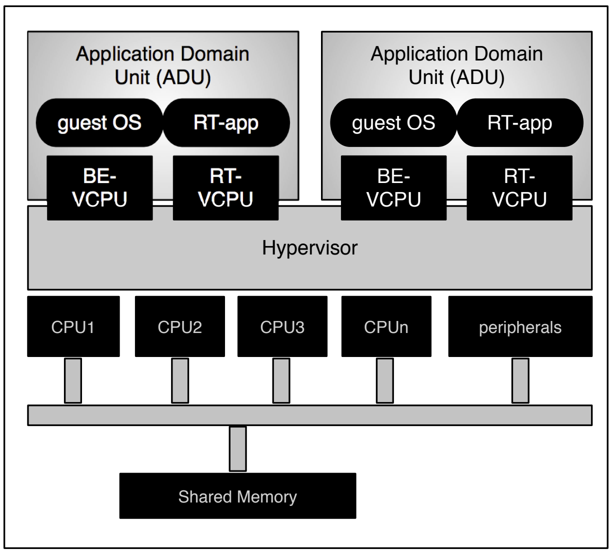


Figure 4.3: Virtual Hellfire Hypervisor's virtualization model.

The VHH's model was designed to be compatible with the existing embedded processors that were available at the time it was developed, see Section 2.6. Thus, the resulting model does not support essential hardware features to meet security and performance providing full-virtualization without hardware-assistance. A side effect is the absence of spatial isolation between the guest OS's kernel and user modes. Without additional processor privilege levels, the entire guest OS must execute in user-mode. In addition, all privileged instructions must be handled by the hypervisor. Thus, the model can only be used when the processor meets the Popek and Goldberg requirements (see Subsection 2.2.2).

Aguiar's model provides weak temporal isolation between real-time and non-real-time applications since it does not provide spatial isolation between BE-VCPUs and RT-VCPUs. Figure 4.3 shows that BE-VCPUs and RT-VCPUs coexist in ADUs. Sharing memory space is advantageous for software stack sharing and communication performance. This means that a GPOS could instantiate RT-VCPUs that share all its software stack and communicate using shared memory. However, this approach has one major drawback. A

RT-VCPU may be prevented from executing because a BE-VCPU is holding a shared lock causing a priority inversion. The model proposed in this work, implements spatial isolation between real-time and best-effort VCPUs to provide stronger temporal isolation between them.

The VHH's model was implemented in the 4Kc MIPS processor. This implementation required the modification of the target processor to allow virtualization of the MIPS's exception vector [53]. As the processor's core was modified the hypervisor could not be evaluated in a hardware platform, thus only OVSim [37] simulations were performed. Also, the resulting implementation has performance issues as shown in [3]. Despite the fact that full-virtualization avoids the need for guest OS modifications, the absence of hardware-assistance forces the hypervisor to intervene constantly in the execution of the guest OS. The hypervisor handles all system interrupts routing them to the destined guest OS. VHH uses the trap-and-emulate strategy to deal with privileged instructions. Additionally, memory mapped devices use a similar strategy. When the guest OS tries to access a protected memory address, the hypervisor intervenes checking if it is a valid address to the guest OS, if so, the hypervisor proceeds to emulate the operation. The model proposed in this dissertation uses hardware-assistance to avoid excessive hypervisor intervention. Finally, guest OSs are allowed to access memory mapped devices directly and interrupts targeted to guests can be directly routed.

4.4 Final Considerations

The proposed embedded virtualization model was designed to take advantage of the current technologies available in the embedded processor families. These technologies allow for full-virtualization of the CPU, but they do not support efficient I/O virtualization. Despite the advantage of avoiding guest modifications, experience with the VHH has shown that full-virtualization without proper hardware support results in excessive overhead. Thus, this model avoids full-virtualization for hardware subsets without proper hardware-assisted virtualization. As a result, the proposed model implements a hybrid approach, which provides full-virtualization for the CPU and para-virtualization for I/O devices. Additionally, a list of major features for embedded virtualization was determined and the model was designed to support them. For example, spatial isolation among VMs is guarantied based on hardware support while temporal isolation is provided by BE and RT-VCPUs. Moreover, the extended services concept uses para-virtualization to allow inter-VM communication and real-time service management.

The hierarchical scheduling problem is avoided by the introduction of the RT-VM that maps real-time services directly onto RT-VCPUs. The flexible scheduling mapping proposed may cause the LHP (Subsection 2.4.2) because a multiprocessing OS with concurrent VCPUs may be preempted while holding a spinlock. However, the LHP is not directly addressed in the model, since this problem requires different approaches depending of the scheduling algorithm implemented. But, it can be avoided with different approaches. For example, the real-time services proposed in this model does not implement spinlocks. Thus, they will not be subjected to the LHP. Additionally, single-core processors where the hypervisor maps each VM to a single VCPU does not suffer from the LHP.

The model proposed in this dissertation strengthens the hypothesis that is possible to have an embedded lightweight hypervisor capable of supporting the major embedded virtualization requisites Moreover, hardware-assisted virtualization is important to achieve low footprint and high performance.

5. HELLFIRE HYPERVISOR - AN IMPLEMENTATION USING MIPS HARDWARE-ASSISTANCE

This chapter presents a hypervisor implementation based on the virtualization model discussed in Section 4. This implementation was partially built by Zampiva [90] during his master degree, and finished during the development of this dissertation. Focused on full-virtualization, as proposed by the model, this implementation supports the MIPS virtualization extensions (MIPS VZ). The target processor is the M5150 that supports two-level TLBs, thus, it allows memory isolation at both the hypervisor and guest OS levels. Additionally, this chapter shows how software design decisions based on hardware-assisted virtualization associated with embedded systems characteristics result in a simpler hypervisor. Section 5.1 gives a brief description of the MIPS M5150 processor core. Section 5.2 describes the software architecture of the hypervisor implementation. Section 5.4 explains the memory virtualization strategy adopted. Finally, Section 5.5 describes the fast interrupt delivery mechanism supported by the hypervisor.

5.1 The M5150 processor

The M5150 processor was released by the end of 2013 and was one of the first cores based on the MIPS32 Release 5 (MIPS32r5) specification. It is a fully synthesizable MIPS processor designed for embedded applications. The main features of the processor are: single-core, 5-stage pipeline, 32-bit address and data paths, MIPS32 and microMIPS ISA [35], 16 or 32 dual-entry joint Translation Lookaside Buffer (TLB) with variable page sizes, Multiply/Divide Unit, Floating Point Unit (FPU), upto 16 general purpose register shadow sets (copies of the normal GPR to avoid the need to save and restore them on entry to high-priority interrupts or exceptions) and Virtualization Module Support (MIPS VZ). Additionally, as part of the virtualization module, the core has two COP0 instances: one for the guest OS and the other exclusive to the hypervisor. Moreover, in the M5150 terminology the COP0 state for the guest OS is called guest-context while the COP0 state for the hypervisor is called root-context of the processor.

5.1.1 MIPS32's Memory Model

Figure 5.1 depicts the memory model of the MIPS32 processor family. The memory map for processors that implement TLB is different from processors with fixed mapping. For example, the M5100 processor does not implement the TLB on the guest's level. Thus, only the hypervisor has control over the virtual memory. The M5100 is designed for guests without TLB support. The MIPS-VZ module always requires the root-context TLB. For the purpose of this dissertation, only the memory model of the M5150 processor with two-stage TLB is presented.

The MIPS32 core has 4 gigabytes (GB) of virtual memory space and supports 4GB of physical memory. The virtual memory space is divided into four segments for different purposes. The kuseg support 2GB of virtual address range and starts at 0x0000_0000. It is designed for user applications and can be mapped through the TLB anywhere in the physical memory. The kuseg can be accessed from kernel or user-modes. The segments kseg0 and kseg1 are designed for OS's code and data. It only can be accessed from kernel-mode. Access to these segments from user-mode will cause an address error exception. Both segments are directly mapped to the lower 512 megabytes (0.5GB) of the physical memory. For example, the addresses 0x8000_0000 and 0xA000_0000 are mapped to the physical address 0x0000_0000. However, the kseg1 does not support cache and is used during boot time and for memory-mapped I/O. The last two segments, kseg2 and kseg3, are accessible only in kernel-mode and can be mapped anywhere in the physical memory. These segments were intended to execute programs in supervisor mode. However, it took many years from the conception of MIPS32 before the virtualization in these processors could take place. Additionally, the Linux Kernel uses these segments as high kernel memory.

5.2 Software Architecture

The source code of our hypervisor was written mainly in the C programming language, but some hardware abstraction layer (HAL) parts were written in assembly. For implementation, debug and simulation purposes it was used the MIPS Instruction Accurate Simulator (IASim), which is a hardware simulator for MIPS processors able to simulate an

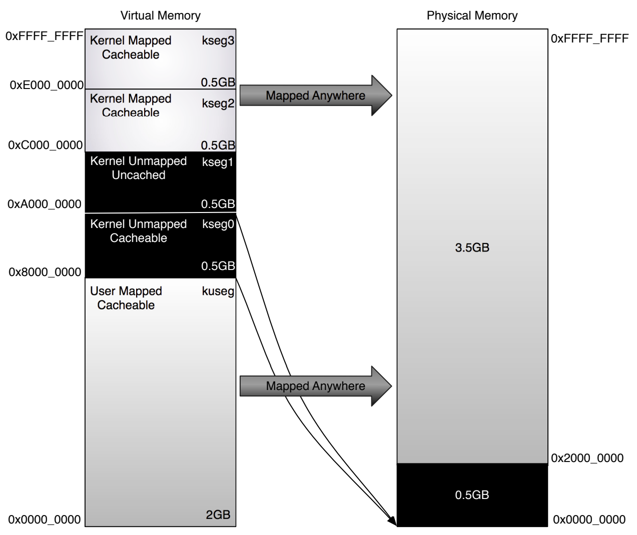


Figure 5.1: MIPS32 memory map.

entire platform, and based on the OVSim. However, during the development of this dissertation, only the M5150 processor, serial port and RAM memory models were available. IASim performs fast simulation aiming to deliver a virtual platform for embedded software development without the need of the real hardware platform. For better accuracy, all performance tests were performed on the SEAD-3 [50] development platform board, as detailed in the Chapter 6.

Figure 5.2 shows the block diagram of hypervisor software implementation. It is composed of a hardware abstraction layer (HAL), real-time and best-effort schedulers, dispatcher, VCPU manager, VM instantiation and toolkit.

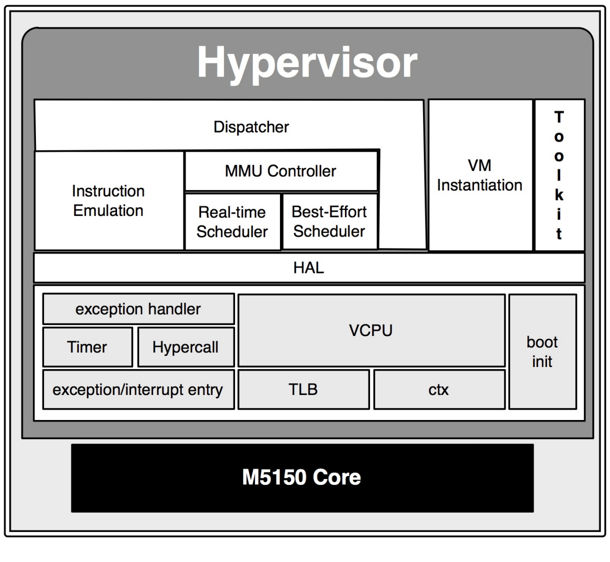


Figure 5.2: Hypervisor's software block diagram.

The HAL implements the low-level application programming interface (API) used to isolate higher layers from further hardware details. Some HAL parts were written in MIPS assembly language due to the necessity of use specific COP0 access, TLB or cache control instructions. After reset, the M5150 processor starts the instruction fetch at address 0xBFC0_0000 in the kseg1 memory segment (non-cacheable). For a stand-alone boot process, e.g., when a boot-loader software is not present, the boot init code is placed at this address to be the first code executed after reset. This code must configure the processor accordingly and copy the hypervisor code to a cacheable memory area, in this case, the kseg0 memory segment. This boot process is used with the IASim. In the SEAD-3, the native bootloader (Yammon) is used to configure the processor and copy the hypervisor directly to the kseg0 segment. Thus, avoiding the use of the hypervisor's boot-init code.

The real-time and best-effort schedulers are responsible for implementing the Earliest Deadline First (EDF) and best-effort round-robin scheduler algorithms for scheduling the RT-VCPUs and BE-VCPUs, respectively. The scheduler module can be easily substituted by any other algorithm. During timer interrupt handling, the hypervisor invokes the scheduler module that manages the VCPU's queues returning to the scheduled VCPU. Consequently, the dispatcher is responsible for dispatching the chosen VCPU to the physical CPU. Thus, the dispatcher uses the HAL interface to write onto the correct CPU's registers.

The instruction emulation module is needed to emulate instructions that cannot be directly executed by the guest OS, e.g., write to specific bits of the COP0 that change the overall processor behavior as the reduced power mode bit in the status register. As a result of the MIPS-VZ module, a minimal number of instructions need to be emulated.

The VM instantiation module is used during the hypervisor's initialization process to configure and start up the VMs. Finally, the toolkit module is a collection of software tools such as linked-list and UNIX compatible libraries.

5.2.1 Virtual Machine and Virtual CPU Software Abstraction

A virtual machine must create the abstraction of a different hardware set from the original hardware platform. This different view consists of a subset of the real platform, like a compatible subset of the processor or a small amount of the entire memory. Additionally, the virtual machine can emulate hardware subsets that do not exist in the underlying hardware.

KVM uses Qemu for this purposes. Pragmatically, a VM is a data structure that keeps all information necessary to control the guest execution. The VM data structure is showed in details in Appendix A.1. Typically, this data structure is kept for each guest OS. In this implementation, the VMs are allocated dynamically during the hypervisor initialization. Thus, the hypervisor's kernel uses the control information kept in these data structures to control the guest behaviors.

Virtual CPU is an abstraction of the real CPU used to keep the last state of the CPU's registers during a context-switching. Similar to VMs, VCPUs are represented by a data structure in memory and are dynamically created during the hypervisor's initialization. The VCPU data structure is showed in details in Appendix A.2. VCPUs are the execution unity of the VMs, each one represents a different execution thread. Thus, the level of concurrence in a VM depends on the number of VCPUs associated with it and the level of concurrence in the hypervisor is directly related to the total number of VCPUs in the system. There is no performance improvements if multiple VCPUs are given to a VM on a single-core processor, instead, the context-switching overhead will cause performance penalties. Thus, this implementation does not support multiple VCPUs for a VM, since the target processor is single-core.

5.3 CPU Virtualization Strategy

The virtualization model described in the Chapter 4 suggests full-virtualization of the CPU. VHH [3] implements full-virtualization of the CPU using a trap-and-emulate strategy (see Section 4.3) causing frequent hypervisor interventions, since all privileged instructions result in traps. This hypervisor implementation takes advantage of the hardware-assisted virtualization of the M5150 processor core. As a result, a hypervisor able to execute guest OSs with minimal intervention is expected. This section describes how the hypervisor uses the MIPS-VZ module to avoid traps.

The MIPS-VZ module allows the hypervisor to use a different hardware configuration for each guest. Thus, the virtualized system can support software from different hardware platforms by running guests with different configurations. Additionally, the guest OSs can have features and capabilities that are different from the host processor and other guests. These different views are possible because the hypervisor can determine what processor feature subset is accessible from the guest-context. In root-context, the hypervisor configures the GuestCtl0 (guest control register 0) register that controls what privileged features can be accessed from the guest mode. Using the GuestCtl0 register, the designers can choose fewer hypervisor interventions, consequently less control over the processor hardware. Alternatively, they can opt for more hypervisor interventions that give them better hardware control. Some virtualized systems require more accurate hardware control, e.g., when guest OSs wish to configure different cache algorithms. In this case, the hypervisor will keep the cache instructions privileged and control the cache operation. This hypervisor implementation is focused on avoiding hypervisor interventions during guest executions as much as possible. Thus, reducing overhead and improving real-time behavior. Table 5.1 summarizes the configurable processor features. First, the hypervisor gives general access to the coprocessor 0 (CP0) using the bit 28 of the GuestCtl0 register. However, some features in the CP0 remain privileged. Thus, the hypervisor allows access to the cache instructions, setting the CG field (bit 24), which allows the guest to configure the cache algorithms. Still, guest OSs are allowed to have access to the config 0 through config 7 registers. The config register set specifies various configuration and capability information. Most of the fields in the config registers are initialized by hardware during the exception reset process, or they have constant values [78]. When the hypervisor wishes to give a different view of the system configuration to a certain guest OS, it must keep the config registers privileged. Thus, reads to this registers in a guest-context will trap the hypervisor that performs a trap-and-emulate strategy. The config registers are unprivileged in guest-context when the bit 23 of the GuestCtl0 register is set. Finally, the guest OSs can directly access the timer registers. Timer virtualization is explained in Subsection 5.3.2.

Table 5.1: GuestClt0 register fields. These bits are used to control the hardware accessibility by the guest OS.

Fields		**Description**
Name	**bits**	
CP0	28	Guest access to coprocessor 0.
AT	26-27	Guest Address Translation control.
GT	25	Guest Timer register access.
CG	24	Cache Instruction Guest-mode enable.
CF	23	Config register access.

Despite the MIPS-VZ module support, some registers or specific bits of the registers always remain privileged. For example, the reduced power mode bit in the status register will always trap the hypervisor on guest write attempts. A VM should not have direct control of the processor performance or any other feature that could disrupt the execution of other VMs. Another reason to keep certain registers privileged is to give a different view of the system to a certain guest OS, as performed by the config registers. The same case applies to the processor identification (PID) register. When the guest OS is trying to identify the processor, the hypervisor can return a different processor identification. For example, in the M5150 processor the hypervisor can return the 4Kc processor identification limiting the guest to a compatible processor subset.

The hypervisor can keep different views of the hardware to different VMs modifying the desired fields of the GuestCtl0 register during context-switching. Thus, the only hypervisor requirement is to keep an individual copy of the GuestCtl0 register in the VM's software control structure for each guest OS. This hypervisor implementation keeps direct access to all features described in Table 5.1.

5.3.1 Best-effort and real-time scheduler algorithms

The hypervisor implements both Earliest Deadline First (EDF) [33] and best-effort (round-robin) policies to deal with RT-VCPUs and BE-VCPUs, respectively. The EDF has higher execution priority than the best-effort scheduler. The latter will not suffer from starvation since the implementation counts on time reservation for it. The EDF scheduler requires the deadline (D), period (P), and capacity (C) real-time parameters, which are received from the hypercall HCALL_RT_CREATE_APP, as explained in Section 4.2.2. For the sake of simplicity, BE-VCPUs and RT-VCPUs are kept in different software queues.

Figure 5.3 illustrates the scheduling strategy where both EDF and best-effort schedulers cooperate to increase the CPU utilization. Assuming the execution of two BE-VCPUs and two RT-VCPUs with the following real-time parameters:

- RT-VCPU 0: period = 5, capacity = 3 and deadline = 5;
- RT-VCPU 1: period = 4, capacity = 1 and deadline = 4.

The BE-VCPUs have the same priority. The scheduling scenario during 20 time slices can be viewed in Figure 5.3. Each RT-VCPU is scheduled according to the EDF

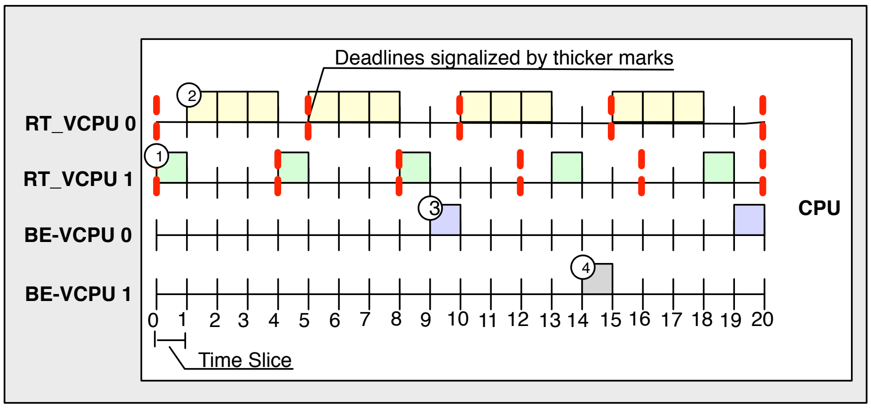


Figure 5.3: Example of the scheduling strategy to combine the EDF and best-effort schedulers.

policy. Thus, RT-VCPU 1 executes in the first quantum as shown in (1). Next the RT-VCPU 2 executes three quanta, according to (2). In quantum 9, BE-VCPU 0 is executed for the first time because there was not any RT-VCPU ready to execute in this time slice (see (3)). Finally, The BE-VCPU 1 is executed for the first time on quantum 14 according to (4). Thus, when executed concurrently, BE-VCPUs will only execute when the RT-VCPUs are not ready to execute. An amount of CPU time can be reserved for best-effort VMs to avoid starvation. The schedulability test for the EDF algorithm when the deadline is equal to the period is determined by the following equation:

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \tag{5.1}$$

U is the CPU utilization. Thus, the sum of the ratios between the capacity and the period of all real-time tasks must be less than 1. The equation 5.1 can be used to reserve an amount of CPU for best-effort tasks. For example, if the designers wants to reserve at least 40% of the CPU time for the best-effort, the result of the above equation must be less than to 0.6, according to equation 5.1. Note that the amount of CPU dedicated to real-time and best-effort tasks is determined at design time. However, during the creation of RT-VCPUs the hypervisor checks the amount of CPU used and if it exceeds the maximum time allowed it will not be created.

5.3.2 Timer Virtualization

Timer virtualization is an important topic for hypervisor implementation, since it implements the timing view from the VM's perspective. MIPS32 cores can be configured to generate timer interrupts periodically. Two registers are used to configure the timer: counter and compare registers. The counter is incremented for each other processor's clock. When the counter value is equal to the compare register, the processor's core generates a timer interrupt. Thus, the hypervisor or OS that wants to generate timer interrupts must read the counter value, add the amount of time to the next interrupt and write the resulting value to the compare register. This procedure must be performed on each timer interrupt to keep its periodicity.

The accessibility of the compare and timer registers from the guest-context is controlled by the GT field on the GuestCtl0 register. Allowing direct access to the counter and compare registers, the guest OS will be able to perform read and writes to the compare register and reads to the counter registers. Thus, the guest OS cannot perform writes to the counter, since it will disrupt the global view of the system timer. In fact, a typical OS implementation will write on the counter register only during its initialization. Therefore, the hypervisor needs to emulate a few writes to the counter register during the VM's initialization. This hypervisor implementation allows the guest OSs direct access to the timer registers avoiding traps during context-switching of the guests.

The hypervisor implements timekeeping within VMs, i.e., it can decide what to do with the time frame that a VM was not executing. The guest OS's time view is called virtual time. The MIPS-VZ module has a special register to configure the time within VMs called GTOffset. This register accepts a two's complement value that will add or subtract the current value of the counter in the guest-context. Thus, the hypervisor can hide the time frame that the VMs were not executing. However, this implementation gives an absolute time view to the guest OSs, i.e., they are aware of the lost time frames. Thus, the value written to the GTOffset is zero, resulting in a virtual time equal to the absolute time.

5.4 Virtual Memory Management

The MIPS VZ module implements a second-stage TLB translation in the hardware. Essentially, the hardware performs the translation from IPA to PA without software intervention, as explained in Subsection 2.3.3. The hypervisor still manages its page table mapping IPA to PA in a second-stage TLB. The guest OS is allowed to configure directly the first-stage TLB. The resulting PA is generated by the hardware combining both TLBs. This mechanism decreases the number of hypervisor exceptions drastically and also hypervisor complexity. The M5150's TLB supports a second-stage TLB translation and a range of page sizes from 1Kbyte to 256Mbytes. Our hypervisor takes advantage of large pages to avoid TLB misses during IPA to PA translations. VMs are loaded into a contiguous memory region, and the IPA translation is statically mapped to reserved TLB entries. For example, to allocate 32Mbytes of physical address space to a VM, the hypervisor uses a dual-TLB entry (MIPS' TLB allows to map two pages by TLB entry) to map two consecutive 16Mbyte pages in the second-stage TLB. Figure 5.4 depicts this scheme. The guest OS manages the first stage address translation in exactly the same way as on a non-virtualized system. In the second-stage translation, the hypervisor maps a virtual memory address range to a contiguous physical address range.

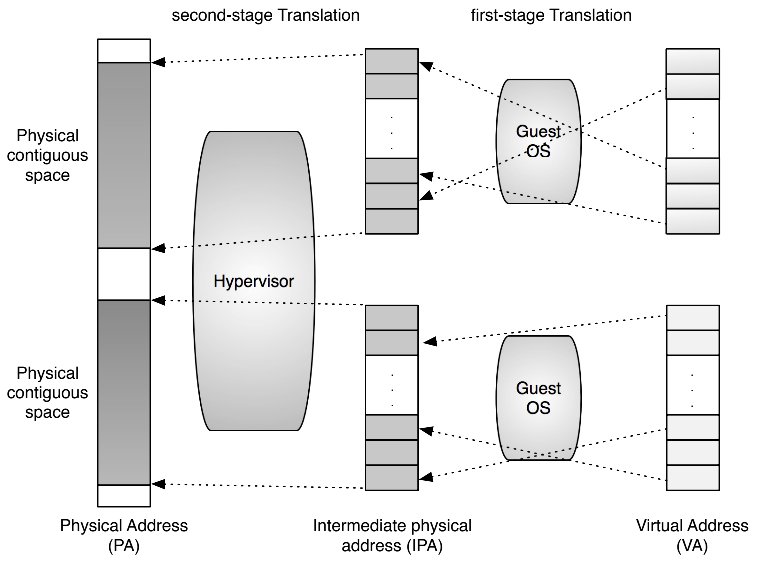


Figure 5.4: Virtual memory organization view of our hypervisor.

Typical hypervisors for cloud computing implement a complete paging mechanism. The hypervisor keeps a page table to map guest OSs to physical addresses. In these systems, the guest OS does not need to be entirely loaded into the main memory to be executed. In fact, the hypervisor can implement an on-demand paging mechanism (swapping). Such a scheme reduces the memory usage since pages that have not been used recently can be stored in the swapping system. Additionally, it avoids the memory external fragmentation problem because the VMs do not need to be allocated contiguously in the physical memory. However, this approach has critical drawbacks for ESs. First of all, swapping systems and on-demand paging mechanisms impact real-time responsiveness. Additionally, some ESs do not support swapping due to storage restrictions. Moreover, a complete virtual memory management mechanism implies a more complex hypervisor and, consequently, a larger footprint and more processing requirements.

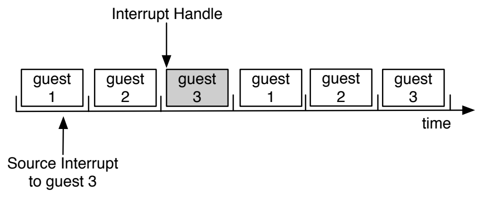
A simplified virtual memory management mechanism brings some advantages to ESs. First, it avoids second-stage TLB misses keeping the VM entirely mapped at the TLB during its execution. Thus, RTOSs that do not implement virtual memory support will not suffer additional delays and jitter due to hypervisor paging management. Secondly, some ESs execute a limited number of virtual machines and some keep a static configuration during their execution. For these systems, memory fragmentation due to contiguous guest OS allocation is not a major problem. This implementation supports mapped devices directly, i.e., it can map non-continuous memory regions to a VM. Usually, such devices are mapped to specific physical addresses requiring a special mapping for guest access. For example, a VM may have mapped 32Mbytes of RAM to be loaded into the physical memory at 0x1000_0000. If the same guest requires access to a memory mapped peripheral at physical address 0x1F00_0800 the hypervisor must configure a second-stage TLB entry to match this address. Finally, a VM requires, at least, one TLB-entry to be mapped at the second-stage TLB, and an additional TLB-entry for each directly mapped peripheral. The M5150 has 32 TLB-entries. If the number of TLB-entries needed to map all VMs and peripheral on a system exceeds 32 entries, the hypervisor must reconfigure the TLB on each context-switching.

5.5 Interrupt Virtualization

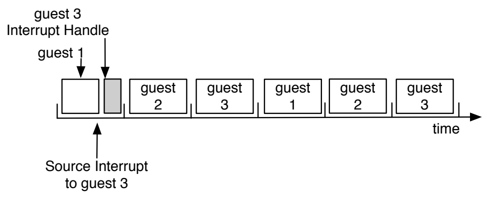
The MIPS VZ module allows to map an interrupt source directly to a guest OS avoiding hypervisor intervention. This mechanism is known as interrupt pass-through and it allows one to support directly mapped devices. Therefore, it does not share devices between guests, but allows a certain guest OS to have direct access. The main advantage of this technique is low overhead, which is close to non-virtualized systems in terms of throughput and latency. However, if an interrupt occurs during the execution of any other guest OS, the hypervisor must decide between:

- keep the interrupt masked and delay its delivery or;
- intercept the interrupt and reschedule the guest OSs.

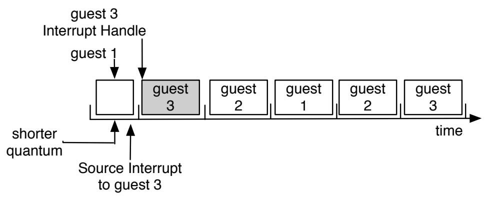
This allows the implementation of different policies for interrupt control. For example, for low priority devices such as a serial port for a user console, the hypervisor may choose to keep the interrupt masked while in the supervisor mode. Thus, the guest OS will handle the interrupt only on its next execution (hypervisor scheduling quantum), resulting in a relatively long delay. This requires a large buffer queue in hardware or hardware flow control, otherwise, data may be lost. This arrangement is acceptable for low bandwidth, non interactive devices. Devices with higher priority or higher bandwidth may require hypervisor intervention to avoid long delays in the interrupt handling. To illustrate a typical scenario, suppose that a Linux guest has a directly mapped Ethernet device. In this case, the hypervisor can keep the Ethernet interrupt unmasked to perform a context switch between guests when a network packet is received and the Linux guest is not executing.



(a) Interrupt delayed.



(b) Fast interrupt with recycled quantum.



(c) Fast interrupt with reset quantum.

Figure 5.5: Quantum scheduler scheme for interrupt delivery.

Figure 5.5 depicts three guest OSs being scheduled in a round-robin policy with the three different quantum schemes for interrupt handling. Figure 5.5(a) shows an interrupt targeted to the guest OS 3 being asserted during the execution of the guest OS 1. Without a proper hypervisor policy, the interrupt delivery will be delayed until the execution of the guest OS 3. In fact, the overhead is similar to a non-virtualized system since there is no hypervisor intervention. However, this causes a long delay to deliver the interrupt. This implementation was designed to use the interrupt pass-through mechanism, which allows for the coexistence of general-purpose OSs and RTOSs with minimal interrupt delivery delay. First, we studied two different schemes regarding the use of the scheduler quantum: recycle quantum and reset quantum. Figure 5.5(b) shows a fast interrupt delivery approach, causing the preemption of the current guest OS and the dispatch of the guest which will execute during the time remaining in the same quantum. In Figure 5.5(c), the current quantum is shortened, and the dispatched guest will execute during an entire new quantum. Chapter 6.6 presents the results and discussion about the implications of these different approaches.

Independently of the quantum scheme adopted, our hypervisor accepts a set of rules to describe the behavior to address different interrupt sources. This set of rules is defined at design time. For each guest OS, the designers define which interrupts are directly mapped and, if an interrupt can induce the preemption of the current guest OS. Additionally, a higher priority guest OS can be marked to never be preempted by the hypervisor during its scheduler quantum. Table 5.2 shows a possible configuration for three guest OSs. Guest 1 has the timer and serial 1 interrupts directly mapped. The column Root Int describes which interrupts will trigger the hypervisor when the guest is not executing. In this case, the hypervisor will intercept serial 1, dispatching Guest 1. Guest 2 has the timer and serial 2 interrupt sources directly mapped. The hypervisor will intercept serial 2 interrupts when the guest is not executing only if the current guest is marked to be preempted. In the example, the hypervisor will preempt guest Linux 1 to deliver network interrupts to guest 2, but the same interrupts will be delayed if the guest 1 is executing. Thus, a high priority guest OS cannot be preempted by events from other guests, but only by the hypervisor scheduler. Our hypervisor provides a strong temporal isolation between guests, which is essential in ES virtualization if predictability of the RTOSs is a major concern. In addition, the flexibility of this approach allows the system to be configured for specific applications. Finally, our hypervisor always keeps the guest timer interrupt directly mapped, since it avoids hypervisor intervention during guest scheduling.

Table 5.2: Example of a set of interrupt rules for a system configured with three guest OSs.

		Rules	
Guests	**Direct Mapped Interrupts**	**Root Int**	**Preempt**
Guest 1	Timer, Serial 1	Serial 1	No
Guest 2	Timer, Serial 2	Serial 2	Yes
Guest 3	Timer, Network	Network	Yes

5.5.1 Virtual Interrupt

The M5150 processor core supports an interrupt injection mechanism called virtual interrupt. Virtual interrupts are software generated and used when the hypervisor needs to inject interrupts in the guest OSs. Our hypervisor implements a secure inter-VM message passing mechanism described in Section 5.6 based on para-virtualization and virtual interrupt injection.

5.6 Inter-VM Communication

As explained in Subsection 4.2.4, the proposed virtualization model does not define a communication mechanism. However, it defines a hypercall interface for communication among VMs. Thus, this implementation adopts a message passing mechanism based on para-virtualization. The hypervisor uses the VM identification number to route the messages among the VMs. The hypervisor requires the address, size and ID destination to route a message, which are discovered during the hypercall. Thus, the hypervisor does not make any assumptions about the message formatting; this is entirely the responsibility of the communicating guest OSs. For example, if a multi-task guest OS needs to demultiplex [77] incoming messages among different tasks, it may add a header to the message indicating the origin and destination task id. In this case, different guest OSs must agree about the header format.

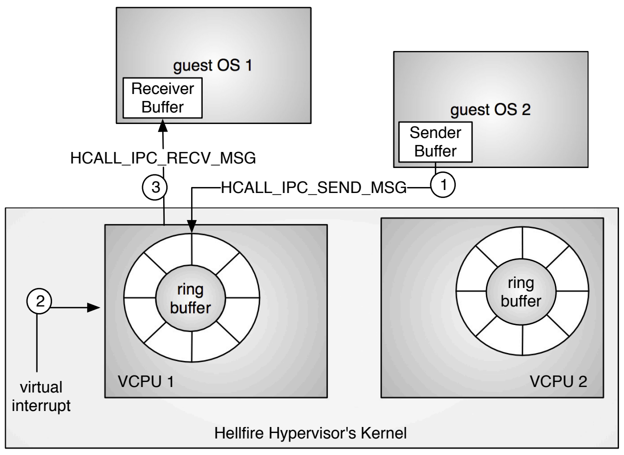


Figure 5.6: Example of inter-VM communication involving two guests.

Each VCPU implements its incoming message queue as a circular buffer, statically allocated for performance purposes. A message targeting a determined VCPU will be copied to its queue, and the hypervisor will insert a virtual interrupt to the VCPU. The next time that the VCPU is executed it will handle the virtual interrupt and call the hypercall to retrieve the message. Figure 5.6 describes the hypervisor behavior while redirecting messages between guests. The guest OS 2 invokes the HCALL_IPC_SEND_MSG hypercall (1) causing a message copy from the guest's buffer to the ring buffer of the VCPU 1. After, the hypervisor injects a virtual interrupt (2) in the VCPU 1. In the next execution, the guest OS 1 will answer to the interrupt and executes the HCALL_IPC_RECV_MSG hypercall. Thus, the hypervisor will copy the message from the ring buffer to the target buffer (3).

5.7 Engineering Effort to Support a Guest OS

Due to the full-virtualization feature, to port a new guest OS to the Hellfire Hypervisor does not require any modification of the guest OS's kernel. However, it is important to highlight that the OS must be ported to the target platform. Thus, it must be able to execute natively on the hardware platform. To port an OS to a new processor or platform is a different job, and it is not necessarily related to virtualization. Thus, the designers must know some details about the memory mapping and platform resources required by the target OS. A guest OS requires specific memory regions, and it may depend on platform resources such as timer counters or other sources of interrupts. The hypervisor allows one to describe the memory arrangement for each guest OS at design time. This description consists of mapping from IPA to PA (see Subsection 2.3.3) addresses that are written to the second-stage TLB by the hypervisor. The designers must choose the amount of main memory available to the guest OS based on the application and OS requirements. For example, 16MB of DRAM are enough to execute a minimal Linux distribution, while 64KB of DRAM are sufficient for a Hellfire OS guest with several tasks. Next, the designers must choose a contiguous area of physical memory to allocate to the guest OS. Additionally, the designers must determine which interrupts and peripherals will be directly mapped to the guest OS. Again, the arrangement between the source of interrupts and target guest OSs is described in a data structure at design time. In any case, the timer interrupt must at least be mapped to the guest OS, since this implementation does not provide timer emulation, i.e., each guest OS must receive its timer interrupts directly. Table 5.2 shows how the interrupt mapping can be described.

Table 5.3: Example of memory mapping for a Linux and a HellfireOS guests.

	Addresses		
	VA	**IPA**	**PA**
Linux	0x8000_0000	0x0000_0000	**0x1000_0000**
	0xBF00_0000	0x1F00_0000	0x1F00_0000
HellfireOS	0x8000_0000	0x0000_0000	**0x1200_0000**
	0xBF00_0000	0x1F00_0000	0x1F00_0000

Table 5.3 shows a valid memory mapping for a Linux and Hellfire OS guest. Both OSs are compiled to execute at the 0x8000_0000 VA. Additionally, each OS require access to peripherals. Thus, a 4KB page is mapped at 0xBF00_0000, where the peripherals mapping starts in the SEAD-3 development board (see Section 6.1). Note that each guest requires, at least two TLB entries. The addresses 0x8000_0000 and 0xA000_0000 (including 0xBF00_0000) correspond to the Kseg0 and Kseg1 memory segments, respectively, as shown in Figure 5.1. These segments are fixed mapped and correspond to the 0x0000_0000 physical address. However, when implementing the MIPS-VZ module, these addresses become IPA, and the hypervisor must map them to the PA. The PA column shows the 0x1000_0000 and 0x1200_0000 addresses highlighted since they correspond to the physical addresses where the guests are located in the memory. Additionally, for peripheral mapping the IPA and PA are the same because the hypervisor keeps the original addresses where the guest OSs expect to find the peripherals.

5.7.1 Linux and Hellfire OSs as Guests

Most of the work to port an OS to the Hellfire Hypervisor is described in Tables 5.2 and 5.3. However, some details must still be explained. Current Linux releases do not use periodic timer interrupts. In fact, when the OS is idle, it programs the timer interrupt to the next event, even if it will take several scheduler quanta, and executes the *wait* instruction. This instruction will put the processor in the power saving mode. The processor comes up once an interrupt arrives. However, wait is a privileged instruction that must be emulated by the hypervisor. Emulating this instruction means to check when the next OS timer interrupt is programmed to happen and to program an event in the hypervisor to insert a virtual interrupt in the guest. However, the current implementation does not support emulation of this instruction. Thus, the Linux kernel was configured to use the periodic timer interrupt technique. Note that this is not an OS modification, since it was not needed to write kernel patches. Instead, both ways to use the timer interrupts are kernel features and they can be selected during kernel initialization. Finally, modifications were made to the Linux kernel to improve its performance once executed as a guest. This is better explained in Section 6.3.

Hellfire OS has fewer restrictions than Linux when virtualized since it is a simpler operating system. However, an initial port to the M5150 processor had to be done before it was virtualized. Once virtualized, para-virtualized communication drivers were then implemented to allow inter-VM communication.

5.8 Current Hypervisor Restrictions

Similar to any other implementation based on a theoretical model, the proposed hypervisor faces some limitations. The first limitation is that the M5150 is a single-core processor. The model proposes to support homogeneous multicore processors. The decision for this processor core was based on availability. The M5150 was one of the first MIPS processors, released in early 2013, implementing the MIPS-VZ technology. Additionally, it was the first MIPS core supporting MIPS-VZ module available to the GSE group. As a single-core processor, mutex or spinlocks are not required in the hypervisor since there are no concurrent executions on its kernel. A consequence is a simpler hypervisor kernel. Finally, it is important to highlight that this is a limitation imposed by the hardware platform that can be overcome when multicore processors become available.

Despite the overhead imposed by share devices, they are important in different situations. Because many guest OSs may require communication with the external world, the only solution is to implement a switching layer at the hypervisor level and para-virtualized device drivers to the VMs. The current stage of this implementation does not support any shared device, e.g. Ethernet. However, the implementation of a shared device is a matter of engineering effort since the para-virtualized interface between guest OSs and the hypervisor is already supported for inter-VM communication. The optimization and techniques to support shared devices is a future work.

The inter-VM communication implemented as a message passing mechanism is a secure and effective communication channel. However, as shown in Section 6.7, it imposes an excessive overhead to the system. Significant improvements can be achieved using a shared memory communication mechanism that is not available at this moment.

The RT-VM concept provides strong temporal isolation among VMs. Additionally, it avoids the hierarchical scheduling problem. A restriction of the RT-VM is that it does not provide a compatible software environment for legacy software libraries. This means, any implementation must be build from scratch, impacting time-to-market and costs. However, this is a model restriction since it does not provide any tool or resource to avoid the problem. A practical implementation may avoid this problem porting a RTOS to execute as a guest in the RT-VM with its scheduler disabled, consequently, executing just one task. Thus, the application can use the RTOS's legacy libraries, but without concurrent tasks.

The Hellfire Hypervisor was built from scratch avoiding any proprietary or open-source external library which makes it the propriety of the GSE group. However, this results in an important drawback: all engineering efforts must be from the university group. Different from several other open-source hypervisors those count on contributions from different groups, the Hellfire Hypervisor is entirely the responsibility of the GSE group. As a result, the current implementation only supports Linux/MIPS and Hellfire RTOS guests. Additionally, there are limitations caused by the lack of engineering efforts because of the small number of engineers working on the project. However, the Hellfire Hypervisor is planned to be released as an open-source software in early 2016.

5.9 Final Considerations

An important question about theoretical models is how practical is it for them to be implemented on a hardware/software platform. This implementation demonstrated that the model can be implemented with relatively simple algorithms and data structures help to keep a small footprint. A factor that contributed to the simplification of the hypervisor was the M5150's hardware-assisted virtualization, especially the support for memory and MIPS CP0 virtualization. The footprint and performance results are analyzed in Chapter 6. The only feature not implemented was the multi-core support because the target processor is single-core. The absence of multi-core support simplified the hypervisor's kernel because it avoided the need for mutex or spinlocks. Other restrictions, like the lack of shared devices, concerns to the limited time and number of engineers working on the project, which can be overcome with more engineering effort. It is fair to conclude that the presented implementation is a suitable representation of the virtualization model proposed in the Chapter 4.

6. HYPERVISOR EVALUATION AND PRACTICAL RESULTS

This chapter is dedicated to the evaluation of the Hellfire Hypervisor. The chapter is composed of a sequence of experiments that show the practical results for different hypervisor aspects. When possible, the results are compared to other hypervisors. Section 6.1 describes the SEAD-3 development board used for the hypervisor deployment. Section 6.2 shows the hypervisor memory footprint. Section 6.3 describes the Linux/MIPS port to the hypervisor and a modification for better performance. Section 6.4 describes the overhead impact of the virtualization layer on the Linux/MIPS. Section 6.5 analyses the overhead impact of the hypervisor on the HellfireOS. Section 6.6 evaluates the proposed fast interrupt delivery mechanism. Section 6.7 shows the results of the inter-VM communication mechanism. Section 6.8 describes two experiments to show the effectiveness of the real-time services. Finally, Section 6.9 presents the final considerations of this chapter.

6.1 SEAD-3 Development Board

The SEAD-3 development platform board [50] was used to validate and conduct performance tests. It supports MIPS processors allowing the user to evaluate the cores in a FPGA environment. The board can be used for performance benchmarking and software development. It is configured with the M5150 processor core running at 50MHz and 512Mbytes of main memory. Additionally, it has several peripherals including a Ethernet network device, two serial ports and a USB.

6.2 Hypervisor Memory Footprint

The memory footprint is the amount of main memory that a software occupies while executing. Similar to OSs, the hypervisor footprint must be as conservative as possible. The memory required by a hypervisor to manage VMs results in overhead. This is especially critical when targeting embedded virtualization. Additionally, virtualization has been applied to the IoT field, where the target devices have severe memory limitations.

Table 6.1 shows the size of the hypervisor's segments. The executable code or text segment requires 35,008 bytes of memory. A read-only data segment needs 2,300 bytes. The data segment (global variables) is only 976 bytes. The amount of memory reserved for the stack is 2,048 bytes. Finally, the memory segment reserved for dynamic allocation (heap) is 20,480 bytes. Thus, the hypervisor footprint during its execution is 60,812 bytes. However, the heap size requirement may vary depending on the application. The VM and VCPU data structures allocation happen dynamically during the hypervisor initialization, as explained in the Subsection 5.2.1. The data structure to represent a VM requires 56 bytes of the heap. The VCPU data structure varies depending on the inter-VM communication requirements (see Subsection 5.2.1). If inter-VM communication is not required for a virtualized system, the VCPU data structure will only need 780 bytes. Otherwise, the space occupied by the message queue will be added to the total amount required by the data structure. For example, a VCPU compiled to support a queue for five messages with 128 bytes each will result in 1460 bytes. After the initialization, heap allocations are no longer required. The maximum number of VMs allowed by the processor's hardware is 7. Thus, a virtualized system executing 7 VMs with a VCPU attached to each one will require 10,612 bytes of the heap.

Table 6.1: Size of the hypervisor's segments. The sum of all segments is the footprint of the hypervisor during its execution.

Segment	**Size (bytes)**
text	35,008
read-only data	2,300
data	976
stack	2,048
heap	20,480
footprint	**60,812**

The Hellfire Hypervisor presents an intermediate footprint between Xtratum and OKL4 with 60,812 bytes, as can be seen in Table 3.3. Footprint comparison with SPUMONE is not possible because its total footprint amount is unknown. With a few tenths of kilobytes, this hypervisor implementation is acceptable to be used in IoT devices.

6.3 Linux Port Experience

Linux/MIPS is the port of Linux for the MIPS architecture. The kernel sources are maintained by the open source community with the support of a few companies, such as Imagination[1]. The community is doing great work to keep the Linux/MIPS updated with the newest mainstream kernel version. Thus, we worked with the Linux kernel release 4.0.0, which already had proper support for the SEAD-3 platform board.

Virtualization provides a subset of the all hardware resources to a guest OS. In this case, the network card, a serial port and a small amount of the main memory for the Linux guest was reserved while the processor was shared with one or more RTOSs. The current release of the Hellfire Hypervisor does not support the sharing of the network or serial port among guest OSs. To begin with, the implementation was focused on exploring assisted virtualization benefits. However, a para-virtualized device driver was implemented to the Linux guest for inter-VM communication purposes. It is important to highlight that the para-virtualized driver is intended for inter-VM communication only and is not required for Linux virtualization. Thus, the fully-virtualized hypervisor supports Linux without any kernel modification. Additionally, the development was focused on avoiding hypervisor interventions during Linux guest execution as much as possible. Thus, it took advantage of the configurable privilege access to the MIPS VZ guest coprocessor 0 (GCP0). The M5150 processor allows the designer to configure privileged access to different instructions and GCP0 subsets. The designers can choose fewer hypervisor interventions, consequently less control over the processor hardware. Or they can opt for more hypervisor interventions which gives them better hardware control. Yet, some virtualized systems require more accurate hardware control, e.g., when guest OSs wish to configure different cache algorithms. In this case, the hypervisor will keep the cache instructions privileged and control the cache operation. This hypervisor implementation allows for complete access to the GCP0 during Linux execution. However, some registers or specific bits of the registers are always privileged. For example, the reduced power mode bit in the status register will always trap the hypervisor in guest write attempts. Another example is the processor identification (PID) register. When the guest OS is trying to identify the processor, the hypervisor can return a different

[1] https://imgtec.com/

processor identification. For example, in the M5150 processor the hypervisor can return the 4Kc processor identification limiting the guest to a compatible processor subset.

Table 6.2: Number of guest exceptions in the original and the modified Linux/MIPS guest during the boot.

	Kernel	
	Original	**Modified**
# of exceptions	1476000	6

The number of guest exceptions generated during the Linux kernel boot process was determined. However, even with the maximum level of privilege, the kernel was still generating an excessive number of exceptions due to GCP0 access to privileged registers. In fact, the Linux/MIPS kernel performs the pooling of the PID register, which is always a privileged GCP0 register in the MIPS VZ specification. In the MIPS R4000/R4400 processors [30] before version 5.0, there is a hardware bug that avoids generating the timer interrupt if the counter register is read at the exact moment that it is matching the compare register. Thus, the Linux/MIPS always checks the PID to avoid reads to the counter register on R4000/R4400 processors. In the MIPS architecture, it is easy to obtain the PID performing a mfc0 instruction. However, a virtualized Linux/MIPS instance will suffer a huge overhead impact. Thus, it was necessary to modify the Linux/MIPS source code to avoid the frequent reads of the PID register. This modification is restricted to three punctual routines: can_use_mips_counter(), get_cycles() and random_get_entropy(). Because the kernel already keeps the processor identification number obtained in the early boot stages in a data structure, the GCP0 reads were substituted by reads in memory. This punctual modification resulted in an important reduction of the number of guest exceptions, as shown in Table 6.2 . It is important to highlight that such a modification does not imply para-virtualization, since the privileged GCP0 read was not substituted by a hypercall. Instead, the processor identification number is obtained from a privileged read to the GCP0 (that is emulated by the hypervisor), but the subsequent reads were substituted by a memory access. The remaining 6 guest exceptions are read/write to GCP0 privileged registers including the PID register itself.

6.4 Overhead Impact on Linux

The virtualization overhead caused by the hypervisor for CPU-bound and I/O-bound applications in the Linux guest was analyzed. Experiments were also conducted for different hypervisor scheduler time slots in order to better understand the influence of our hypervisor on guest performance. Thus, the non-virtualized system was compared to virtualized performance with different scheduler quanta: 10, 20 and 30 milliseconds (ms). For all experiments, only one Linux guest was used, because we were focused on obtaining the direct hypervisor influence over the performance. Additionally, Linux was executing with 32Mbytes of main memory in all experiments. Furthermore, the same Linux kernel binary file was utilized for virtualized and native executions since the Hellfire Hypervisor performs full-virtualization.

6.4.1 CPU-bound Benchmarks

UnixBench is a benchmark used to evaluate performance on Unix-Like systems providing indicators for different system aspects. According to the byte-unixbench[2] website, UnixBench development started in 1983 at Monash University, but many people have updated it over the years. For the best interpretation of the results, the benchmarks were divided into two different groups: user-land and intensive syscall applications. The user-land group is composed of synthetic applications that perform CPU-intensive computation in user-space, i.e., they do not require context-switching between the user and kernel-space. The intensive syscall group consists of synthetic applications that use syscalls, requiring context-switching between the user and kernel-space. However, these benchmarks do not require context-switching between the guest and hypervisor. Thus, the only intervention during guest execution is the hypervisor's scheduler interrupt timer.

Each benchmark was performed 10,000 times to determine the average execution time. Figures 6.1 and 6.2 shows the normalized performance results for the user-land and syscall application groups. As expected, the overhead increased slightly with a smaller quantum due the increasing hypervisor intervention. In the user-land group, dhry2 was the most affected application with a performance penalty of 5.57% when compared to a 10ms

[2]https://code.google.com/p/byte-unixbench/

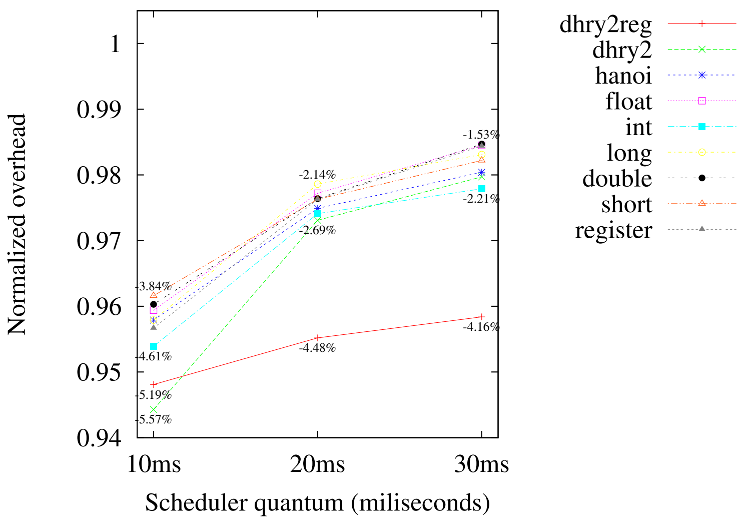


Figure 6.1: Performance overhead for user-land applications relative to non-virtualized performance.

scheduler quantum. Additionally, most of the applications suffered an overhead of less than 3% with a 30ms scheduler quantum. The syscall group suffered a greater performance impact since the benchmark force context-switching between the user and kernel-space in the Linux guest. In the worst case, the syscall close resulted in a penalty of 15.25% performance loss with a 10ms scheduler quantum. However, for the majority of the cases the overhead was lower than 9%.

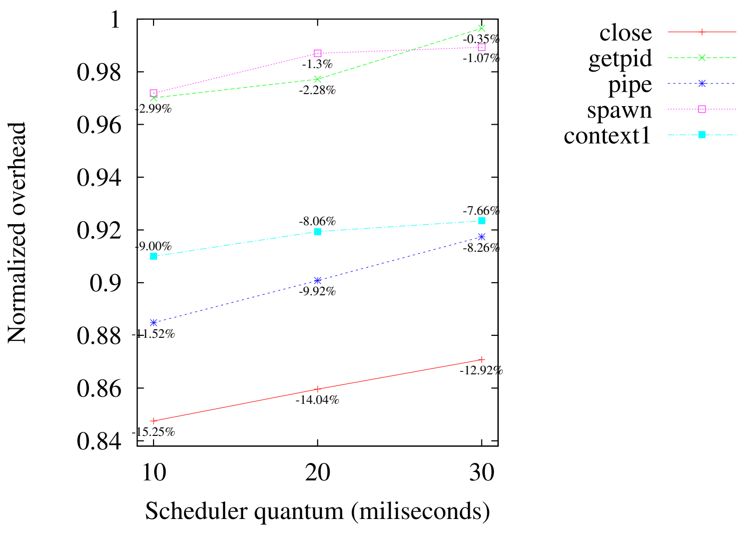


Figure 6.2: Performance overhead for syscall applications relative to non-virtualized performance.

6.4.2 IO-bound Benchmark

The Hellfire Hypervisor supports the SEAD-3's network device directly mapped to the Linux guest allowing it to receive network packages without any hypervisor intervention. The Iperf tool was used to measure the network bandwidth between the SEAD-3 board and a Linux host. Thus, it was possible to determine how our virtualization layer affects the I/O performance of the network device when directly mapped. Iperf consists of a client/server application originally developed by NLANR/DAST as a tool for measuring maximum TCP and UDP bandwidth performance [27]. In this experiment, the Iperf server was executed on a Linux host and the Iperf client on the SEAD-3 board. Thus, the Iperf TCP bandwidth was measured for 1 minute for each experiment case. Figure 6.3 shows the results. The throughput was 8.32 Mbits/second for the native Linux execution. With a 30 ms scheduler quantum, the overhead was 1.84%. As expected, the overhead increased slightly with a smaller quantum reaching 4.39% for a 10 ms quantum. Even with smaller quanta, the overall results are optimistic and can be attributed to minimal hypervisor intervention.

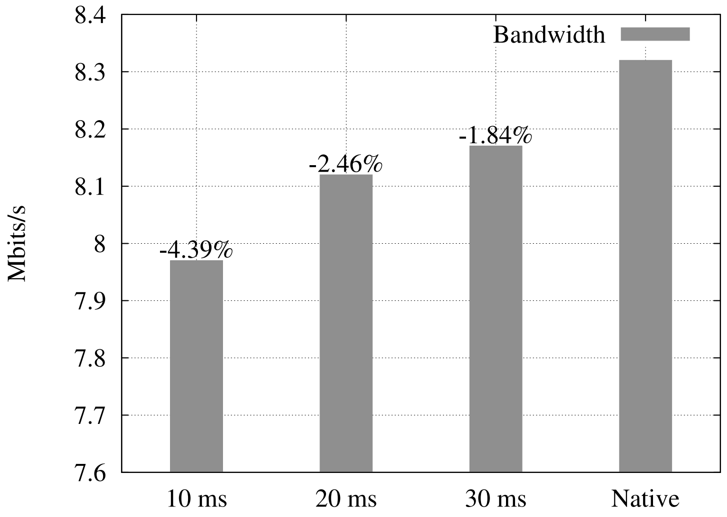


Figure 6.3: Iperf bandwidth results for TCP protocol comparing native versus virtualized execution with different hypervisor's scheduler quantum.

6.5 Overhead Impact on HellfireOS

This section evaluates the virtualization overhead imposed to the Hellfire OS. The HellfireOS's kernel executes only in kernel mode. Thus, it does not perform memory isolation between tasks because it does not implement memory management. However, the HellfireOS can execute on small processors without MMU support. Two different benchmarks were ported to the RTOS for evaluation purposes. The first algorithm was the adaptive differential pulse-code modulation (ADPCM) [21]. ADPCM is a technique to convert analog sound to binary information. It is used for voice communication and sound storage. The second was a data compression algorithm, where only compression is done in a buffer containing totally random data. The implementation of this algorithms was obtained from the WCET Project[3], which keeps a large number of benchmark programs. Thus, the source-code was ported to the HellfireOS' APIs.

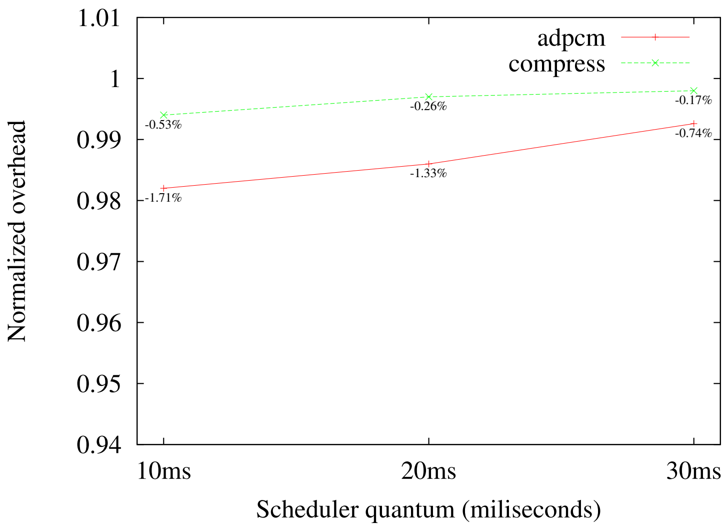


Figure 6.4: Performance overhead for CPU intensive applications relative to non-virtualized performance.

Both algorithms were performed natively and virtualized with hypervisor scheduler's quantum of 10, 20 and 30ms. The average time for 10,000 executions for each algorithm was obtained. Figure 6.4 shows the normalized results relative to non-virtualized execution. The worst case overhead was 1.71% for the ADPCM algorithm executing with a scheduler quantum of 10ms. The measured overhead is still smaller than the overhead for Linux CPU-Bound applications (see Figure 6.1). Despite the HellfireOS to be a lightweight RTOS requiring fewer computation resources, the set of processor resources saved during context-switching is the same of the Linux. A possible explanation for the lower overhead in the HellfireOS is cache effects since the code size impact over the processor's cache was not considered in this study. Linux has a larger code size than HellfireOS what can make it more susceptible to cache effects when submitted to hypervisor interventions. However,

[3]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

more research is necessary to determine the exact cause of the smaller overhead in the HellfireOS benchmarks.

6.6 Interrupt Delivery Delay

This experiment was performed to test the effectiveness of the fast interrupt delivery mechanism as described in Section 5.5. Thus, the following experiments show how the response delay of the OS's interrupt handler is affected by the hypervisor interference and how the proposed solution can improve overall system responsiveness. The virtualized system setup consisted of a Linux guest with a variable number of concurrent VMs executing HellfireOS instances. The Ethernet device was directly mapped to the Linux guest. For interrupt delay measurement, the Internet Control Message Protocol (ICMP) was used to send echo request messages from an Intel Xeon server running Debian Linux 7.8 directly connected to the SEAD-3 board. For all test cases, ICMP messages were sent at intervals of 50 milliseconds (20Hz) using the Linux ping application[4]. Each test case consisted of 10,000 echo request messages followed by their respective response messages (echo reply). Thus, the round-trip time (RTT) [77] of the messages for network communication with the SEAD-3 board was determined. Delays in the Linux interrupt handler caused by the hypervisor impacting directly in the RTT were expected. Finally, the average delay ($\overline{x}$), standard deviation (s), median (m) and the 95^{th} percentile (p^{th}) for the RTT of the messages were determined. Results for the Linux/MIPS native (non-virtualized) execution can be viewed in Table 6.3. Table 6.4 shows the results of all experiments performed using virtualization. Lines three and five depict the behavior of the recycled quantum and reset quantum policies with half of the VMs marked as non-preemptable. Figure 6.5 shows the corresponding histogram for the RTT of native Linux execution messages. The data sets used to generate the results of the lines 1 to 5 of the Table 6.4 are represented in the histograms of the Figures 6.6, 6.7, 6.8, 6.9 and 6.10.

Once the native response time was obtained, the results were compared to virtualized instances of the Linux/MIPS. For each test case, the guest Linux/MIPS shared the processor with VCPUs in different system configurations: 2 VMs (one Linux/MIPS and one BE-VCPU), 4 VMs (one Linux/MIPS and three BE-VCPUs) and 6 VMs (one Linux/MIPS

[4]http://linux.die.net/man/8/ping

Table 6.3: Average ($\overline{x}$), standard deviation (s), median (m) and 95th percentile (p^{th}) for RTT of the messages in milliseconds for Linux native execution.

Strategy	$\overline{x}$	s	m	p^{th}
Linux interrupt police	1.073	0.107	1.05	1.24

Table 6.4: Average ($\overline{x}$), standard deviation (s), median (m) and 95th percentile (p^{th}) for RTT of the messages in milliseconds for virtualized Linux execution.

#	**Strategy**	**2 VMs**				**4 VMs**				**6 VMs**			
		$\overline{x}$	s	m	p^{th}	$\overline{x}$	s	m	p^{th}	$\overline{x}$	s	m	p^{th}
1	**Without interrupt police**	9.35	10.21	2.73	29.5	30.7	29.6	86.4	66.29	66.29	49.40	66.90	145
2	**Recycled quantum (R.Q.)**	1.87	4.09	1.2	1.83	12.05	1.56	5.52	6.90	6.90	21.04	1.44	50.4
3	**R.Q. with non-preempt. VCPU**	9.28	10.16	2.70	29.1	12.90	1.66	29.1	9.72	9.72	19.83	1.68	31.5
4	**Reset quantum (Res.Q.)**	1.29	0.81	1.11	1.69	2.49	1.55	1.85	2.50	2.50	10.39	1.22	1.82
5	**Res.Q. with non-preempt. VCPU**	2.86	3.17	1.1	9.89	8.61	8.19	2.07	21.1	10.49	12.32	1.71	30.7

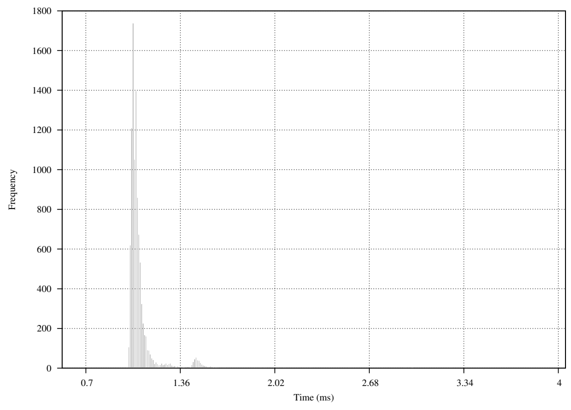
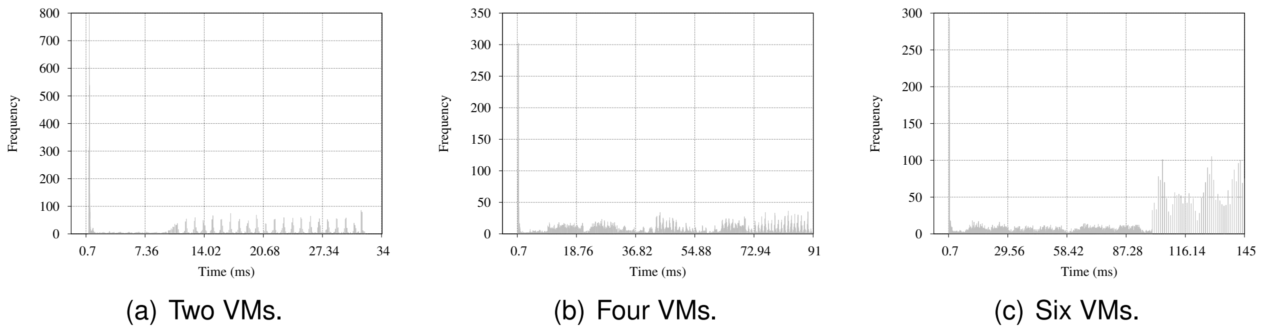


Figure 6.5: Histogram for RTT of the messages for native execution of the Linux/MIPS.

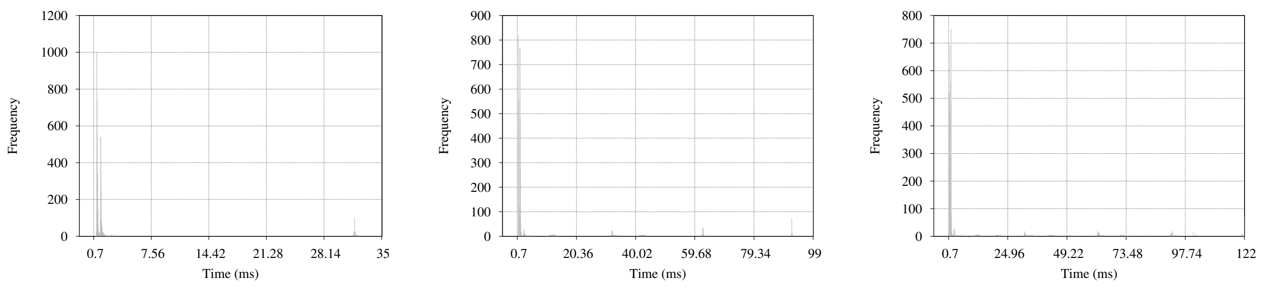
and five BE-VCPUs). Each BE-VCPU was performing the HellfireOS as a guest during the tests. For each configuration, tests were performed without any specific policy, consequently increasing the response delay. Line one of the Table 6.4 shows that the delay growth is proportional to the increasing number of concurrent VMs. Figures 6.6(a), 6.6(b) and 6.6(c) show the corresponding histogram to the RTT of the messages to the configuration of 2, 4 and 6 VMs, respectively. Observe the increased horizontal scale of the histogram in the figures. These results show that the hypervisor significantly affected the response time when compared to native results. This was expected since the VMs were scheduled in a round-robin algorithm, and an interrupt on the Ethernet device must wait for the next Linux/MIPS execution. Moreover, the hypervisor scheduler time quantum is 30 milliseconds. For example, in a system configuration of 4 VMs the delay to handle an interrupt can be up to 90 milliseconds if the guest Linux/MIPS is the last in the round-robin queue.



(a) Two VMs.
(b) Four VMs.
(c) Six VMs.

Figure 6.6: Histogram for RTT of the messages without policy.

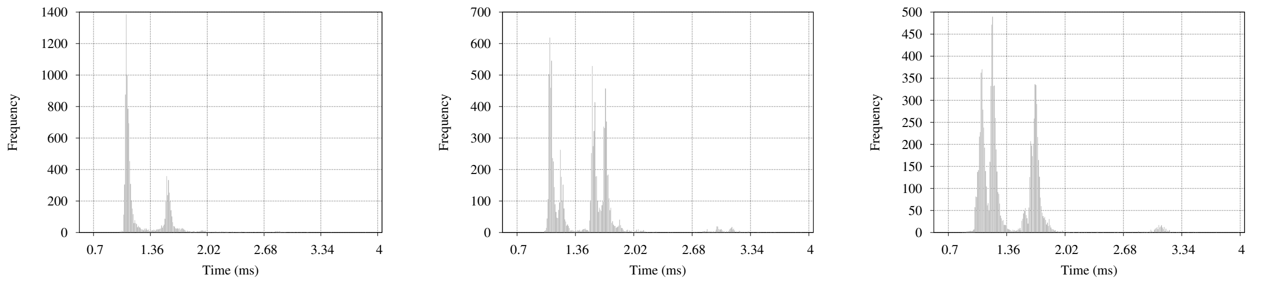
After determining how the hypervisor affects the interrupt response time, two different policies were tested to discover which one gives the best approximation to the native response time. Line two of Table 6.4 (Recycled quantum) shows the results when applied the fast interrupt policy together with the recycled quantum scheme (as explained in Figure



(a) Two VMs. (b) Four VMs. (c) Six VMs.

Figure 6.7: Histogram for RTT of the messages with fast interrupt policy and quantum recycling.

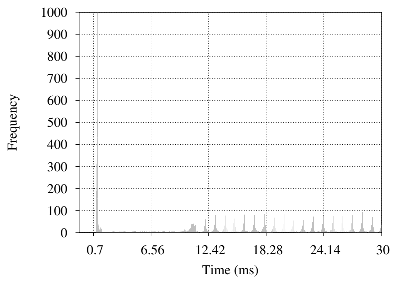
5.5(b)). This technique improves the overall results. However, the recycled quantum mechanism has a drawback. If the remaining of quantum time is not sufficient to finish the interrupt handler routine, it will be finished only in the next guest's execution. Figures 6.7(a), 6.7(b) and 6.7(c) depict the corresponding histogram of the RTT of the messages for the configuration of 2, 4 and 6 VMs, respectively. Most interrupts have a fast response as shown by the peak near 0.7 milliseconds. For such cases, the interrupt assertion happened during the guest execution or the remaining quantum time was enough for the interrupt handler to finish its job. However, some messages experience a long delay and the response time is close to multiples of 30 milliseconds. Thus, the remaining quantum time was not enough and the interrupt handler only finished in the next execution.



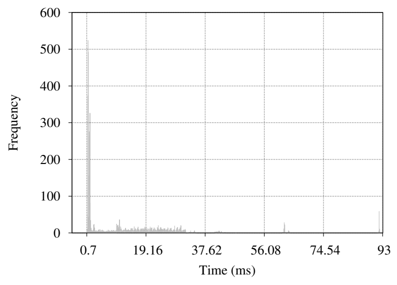
(a) Two VMs. (b) Four VMs. (c) Six VMs.

Figure 6.8: Histogram for RTT of the messages with fast interrupt policy and quantum reset.

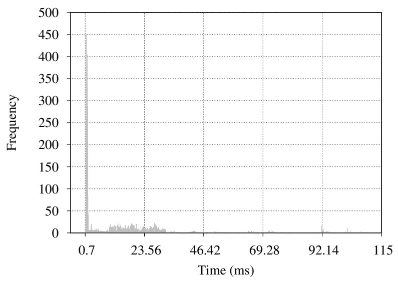
Line four of Table 6.4 (Reset quantum) shows the results of the fast interrupt policy with a reset quantum scheme. For 2 VM configurations, the average and standard deviation delays are close to native execution. In the 4 and 6 VM system configurations, the results are slightly higher but better than results for the recycled quantum scheme. Figures 6.8(a), 6.8(b) and 6.8(c) show the corresponding histogram of the RTT of the messages for the configuration of 2, 4 and 6 VMs, respectively. Observe that the horizontal scale in the figure is similar to native execution. These histograms show two peaks. The peak for delays under one millisecond represents interrupt assertions that happened when the target guest OS was executing. The second peak for delays greater than one millisecond represents interrupt assertions that caused a context switch because the target guest was not executing.



(a) Two VMs, one non-preempt.



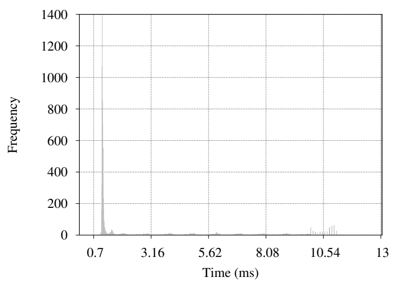
(b) Four VMs, two non-preempt.



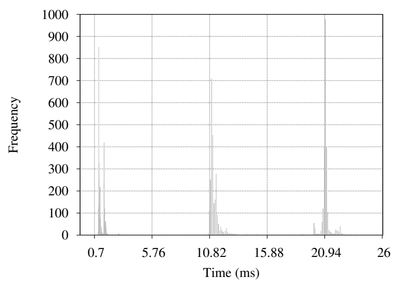
(c) Six VMs, three non-preempt.

Figure 6.9: Histogram for RTT of the messages with fast interrupt policy and quantum recycling with non-preemptable VCPUs.

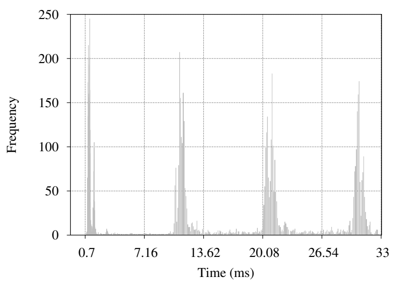
Finally, lines 3 (R.Q. with non-preempt. RTOS) and 5 (Res.Q. with non-preempt. RTOS) of Table 6.4 show the results for recycled quantum and reset quantum schemes in the presence of non-preemptive guests. For each configuration, half of the guests were marked as non-preemptive. The presence of non-preemptive guests increased the average and standard deviation delays, but the results are better than the approach without any interrupt policy. Figures 6.9(b), 6.9(c) and 6.9(c) show the histograms for recycled quantum with non-preemptive guests for 2, 4 and 6 VMs. Figures 6.10(b), 6.10(c) and 6.10(c) show the histograms for reset quantum with non-preemptive guests for 2, 4 and 6 VMs.



(a) Two VMs, one non-preempt.



(b) Four VMs, two non-preempt.



(c) Six VMs, three non-preempt.

Figure 6.10: Histogram for RTT of the messages with fast interrupt policy and quantum reset with non-preemptable VMs.

The reset quantum scheme associated with the fast interrupt policy presented better results than the recycled quantum. However, this scheme may be disruptive for RTOSs with strict timing constraints and can not be combined with the EDF scheduler algorithm.

The EDF implemented computes the next time slot execution of the RT-VCPUs based on the number of quanta performed. The reset quantum scheme breaks the correspondence between time and the completed number of quanta. The recycle quantum mechanism keeps a substantial time isolation between guests. Thus, it must be used when predictability is more important than low response time. Additionally, the fast interrupt policy allows for the reduction of hypervisor interference for values close to non-virtualized systems in certain configurations. Finally, the proposed mechanism is flexible and allows for customization of the hypervisor for specific applications for any number of guest OSs. Section 6.8.2 describes a scenario where the fast interrupt delivery policy is combined with RT-VCPUs.

6.7 Inter-VM Communication Response Time

In order to calculate the RTT of the messages to the inter-VM communication mechanism, it was implemented an echo server application, which replayed all messages received, in the HellfireOS. A client application was implemented in the Linux to send 16, 32 and 64 byte messages after reading the server's response. The measurement of the average RTT for 10,000 messages for each different size resulted in 59.90, 59.90 and 59.99ms with a standard deviation of 0.71ms, 0.71ms and 0.69ms, respectively. An increase in the message size from 16 to 64 bytes did not significantly impact the RTT. The imposed overhead was caused by message copies from the sender's buffer to the target VCPU and to the receiver's buffer. The message exchange mechanism caused an additional copy but allowed the hypervisor to have better communication control, similar to pipes in Linux. Figure 6.11 depicts the histogram of the RTT's distribution for 16, 32 and 64 byte messages. In this experiment, we applied our fast interrupt delivery policy. Thus, once the Linux guest sent a message, the hypervisor rescheduled the VMs to the target guest resulting in a faster response and reducing the standard deviation.

6.8 Real-time Services Performance

This section depicts experiments related to real-time services. Thus, RT-VMs are used to provide tasks performed by RT-VCPUs. Subsection 6.8.1 describes an experiment

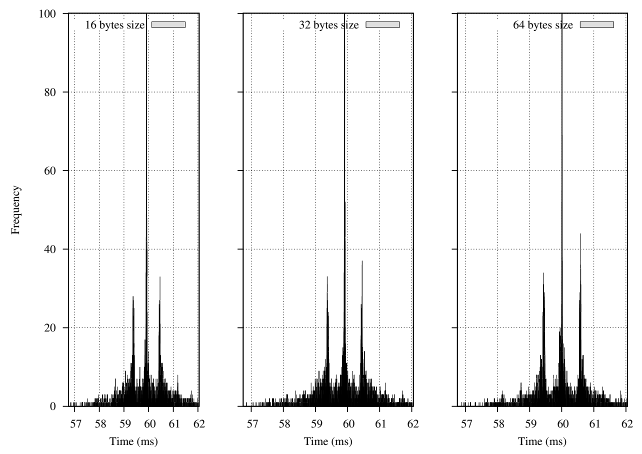


Figure 6.11: RTT's histograms for inter-VM communication using the Hellfire Hypervisor communication mechanism.

to determine the real-time scheduler delay caused by the EDF implementation in a system with an increasing number of RT-VCPUs. Subsection 6.8.2 determines how RT-VCPUs are influenced by external system interrupts.

6.8.1 RT-VCPUs Execution Delay

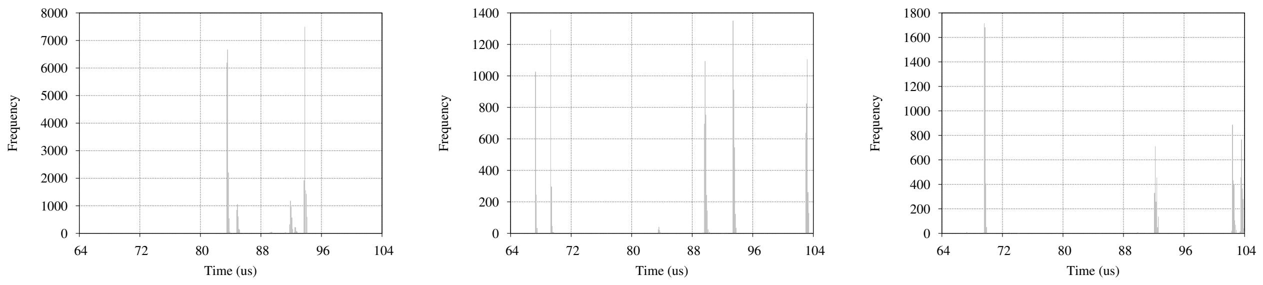
It is important to determine how long an RT-VCPU takes to start its execution when it is ready. The real-time scheduling involves the timer interrupt handling and the EDF algorithm execution. As mentioned before, the scheduling process adds overhead to the system and the time expended in this process is discounted from the RT-VCPU quantum. To understand how an increasing number of RT-VCPUs affect their execution is important for further improvements in the scheduler algorithm.

The system configuration for this experiment consisted of a HellfireOS guest responsible for starting and stopping the RT-VCPUs. The RT-VCPUs were managed by a RT-VM. The number of RT-VCPUs in execution in the system were 1, 4 and 8. External interrupts were masked during this experiment. The time needed for RT-VCPU scheduling was determined by reading the counter register when entering the interrupt handler and calculating the time spent to return to the execution of the VM.

Table 6.5 shows the average, standard deviation, median and the 95^{th} percentile after 10,000 scheduling rounds for each number for RT-VCPUs in microseconds. The corresponding histograms for the execution of the 1, 4 and 8 RT-VCPUs are shown in Figures

Table 6.5: Average ($\overline{x}$), standard deviation (s), median (m) and 95th percentile (p^{th}) for the execution delay on the EDF scheduler in microseconds.

# RT-VCPUs	$\overline{x}$	s	m	p^{th}
1	88.35	4.84	85	94
4	88.66	12.75	91.9	103.2
8	91.44	16.31	97.45	111



(a) System with one RT-VCPU. (b) System with four RT-VCPUs. (c) System with eight RT-VCPUs.

Figure 6.12: Histogram of the execution delay for RT-VCPUs in microseconds.

6.12(a), 6.12(b) and 6.12(c) The average time expended to schedule a RT-VCPU is less than 100 microseconds. When compared to a scheduler quantum of 30ms this represents only 0.003% of the quantum. However, the standard deviation and the 95th percentile increases with a higher number of RT-VCPUs. This is caused by the EDF algorithm implementation that has a complexity of $O(n)$. Despite the increase in the response time to represent only a few microseconds, this can be minimized with a predicable EDF implementation, like proposed by [69].

6.8.2 Interrupt Handling Interference on RT-VCPUs

A sequence of four experiments were performed to show how the Hellfire Hypervisor can be configured to support real-time services while minimizing the interrupt delivery delay on BE-VCPUs. Additionally, these tests determined how RT-VCPUs are influenced by external system interrupts. For this, the extended real-time services and fast interrupt delivery capabilities were combined in the same virtualized system. The system configuration for this experiment consisted of a Linux guest, a HellfireOS guest and a RT-VM. The Linux guest was responsible for network communication with external devices because it was configured with TCP/IP support and the Ethernet device was directly mapped to it. The HellfireOS was responsible for managing the real-time services (start and stop). The RT-VM implemented a real-time service to perform the ADPCM algorithm (see Subsection 6.5). When the algorithm is encoding or decoding for sound reproduction purposes, it must keep a constant bit rate to avoid sound glitches. Thus, the hypervisor must guarantee that concurrent events will not affect the ADPCM execution.

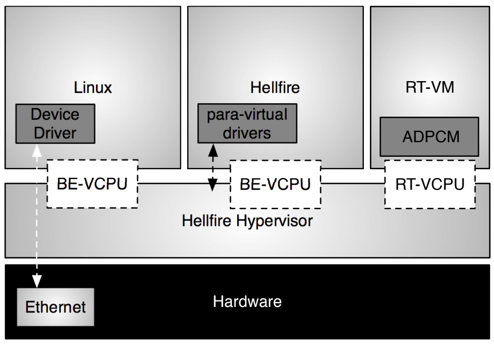


Figure 6.13: System configuration for the ADPCM execution.

On the RT-VM, the ADPCM algorithm was mapped to a RT-VCPU that is scheduled by the EDF algorithm, as explained in Subsection 5.3.1. The real-time parameters of the RT-VCPU were: deadline 10, period 10, and capacity 1. Thus, a tenth of the processor's time is reserved for the RT-VCPU. The remaining time is available to the BE-VCPUs responsible for executing the Linux and HellfireOS guests. The final system configuration consisted of two BE-VCPUs and 1 RT-VCPU. Figure 6.13 shows this arrangement. Similar to the experiment shown in Section 6.6, the ping tool was used to generate echo request messages from a host computer to the SEAD-3 board at a rate of 20Hz.

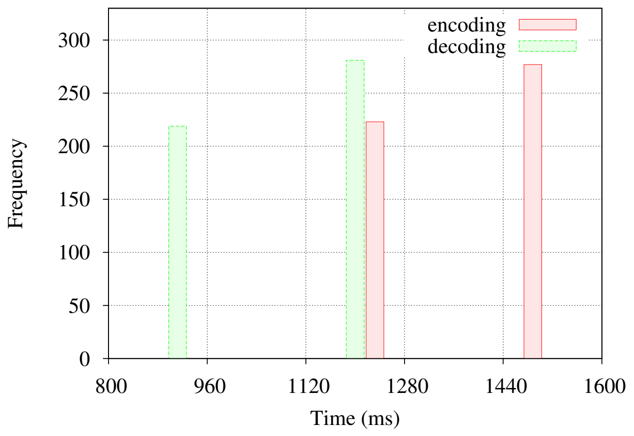


Figure 6.14: Histogram of encoding and decoding execution time for the ADPCM algorithm without external interferences.

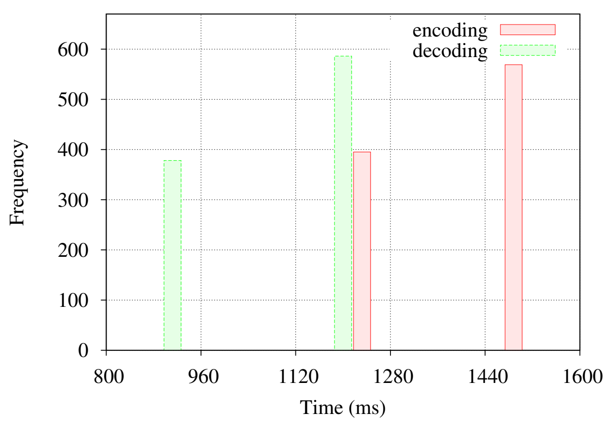
Experiment 1. The first experiment determined how much time the ADPCM algorithm takes to encode and decode a data array. The algorithm was performed during

Table 6.6: Average ($\overline{x}$), standard deviation (s), median (m), 95th percentile (p^{th}), worst execution case (WEC) and best execution case (BEC) to ADPCM encoding and decoding and the RTT of the messages for different system configurations in milliseconds.

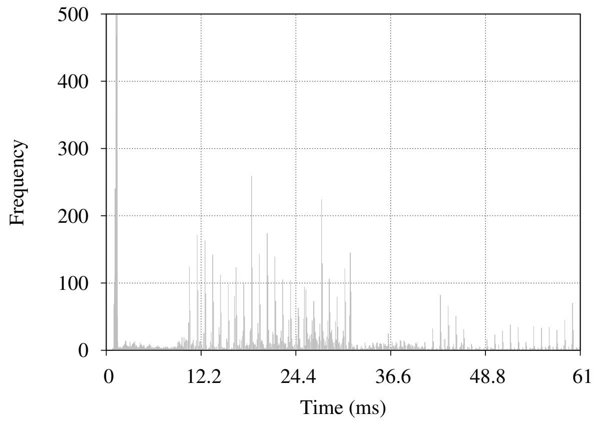
#	System Conf.	Test	$\overline{x}$	s	m	p^{th}	WEC	BEC
1	**Execution time**	**Encoding**	1366.58	134.21	1487	1487	1487	1217
		Decoding	1068.94	133.1	1187	1188	1188	917
		RTT	1.073	0.107	1.05	1.24	3.25	0.99
2	**Non policy**	**Encoding**	1375.27	133.16	1487	1488	1488	1216
		Decoding	1082.17	148.63	1188	1188	1189	917
		RTT	13.37	15.1	5.78	45.5	66.6	0.78
3	**Fast policy - non-RT**	**Encoding**	2004.25	546.48	2082	2705	2997	1217
		Decoding	1505.45	497.15	1211	2390.1	2677	917
		RTT	2.077	4.8	1.45	1.74	63.6	1.07
4	**Fast Policy - RT**	**Encoding**	1378.48	132.55	1487	1488	1490	1217
		Decoding	1081.0	132.12	1188	1188	1188	917
		RTT	3.60	7.26	1.48	20.81	65.8	1.05

1,000 encoding and decoding cycles and the time of each execution was recorded. Table 6.6 shows the numeric results for this experiment in line 1. Additionally, it was plotted in the histogram shown in Figure 6.14. During this experiment, the RT-VCPU was performed alone and all external interrupts were masked. Thus, obtaining the execution time without the influence of other VCPUs or interrupts. The worst case execution time for encoding was 1,487 ms and 1,217 ms for decoding. Moreover, line one of Table 6.6 shows the RTT of the messages for native Linux execution for comparison purposes with the upcoming experiments.

Experiment 2. The second experiment showed the behavior of the ADPCM execution time and the RTT of the messages when the fast delivery policy was not used. This is the simplest configuration, where all interrupts are postponed to the next execution of the target VCPU. This scheme guarantees a temporal isolation to the RT-VCPU since interrupts to other VMs do not preempt its execution. The histogram for 1,000 encoding and decoding cycles is shown in Figure 6.15(a). Additionally, line 2 of the Table 6.6 shows the numeric results. It confirms that the ADPCM's execution time was not affected by the external interrupts since it is similar to the histogram in Figure 6.14. However, this scheme had an undesired effect over the RTT of the messages: the average response time increased substantially. Figure 6.15(b) is a histogram for 10,000 messages recorded during the experiment. The histogram for the RTT of the messages for native Linux execution can be seen in Figure 6.5.



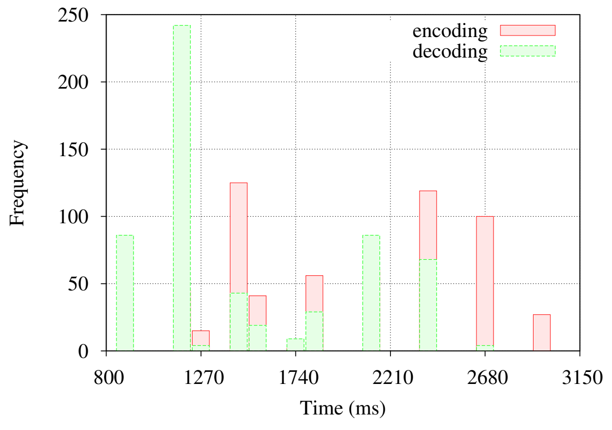
(a) Histogram of encoding and decoding time for the ADPCM algorithm.



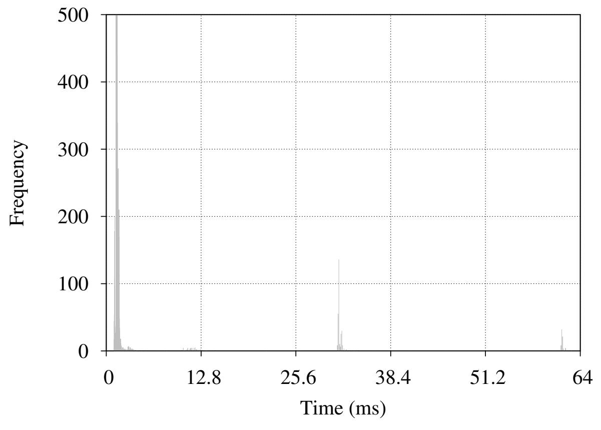
(b) Histogram for RTT of the messages.

Figure 6.15: System response time without the fast interrupt deliver policy and a preemptable RT-VM.

Experiment 3. The third experiment tried to minimize the RTT of the ICMP messages applying the fast interrupt delivery policy. However, the RT-VM was kept as preemptable for an evaluation of how system interrupts can intervene on the RT-VCPU's execution. Thus, both the RT-VCPU and the HellfireOS could be preempted for interrupts targeting the Linux guest. Similar to the previous experiments, 1,000 ADPCM encoding and decoding cycles and 10,000 messages RTT were recorded. Figure 6.16 shows the resulting histograms. The numerical results can be seen in line 3 of Table 6.6.



(a) Histogram of encoding and decoding time for the ADPCM algorithm.

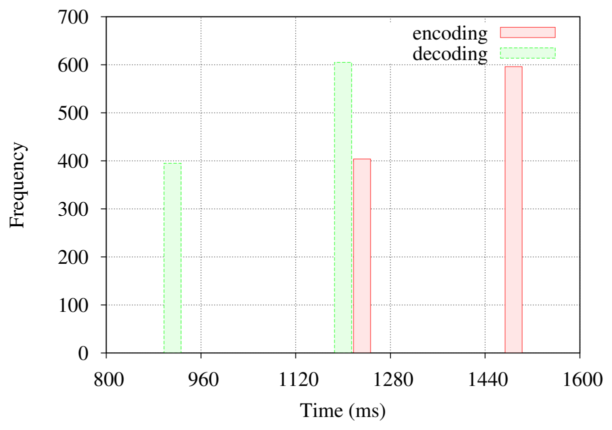


(b) Histogram for RTT of the messages.

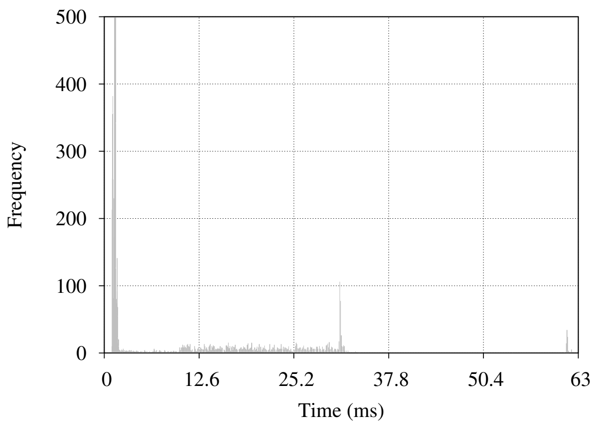
Figure 6.16: System response time with the fast interrupt deliver policy and a preemptable RT-VM.

Figure 6.16(b) and the numerical results show that the RTT of the messages improved significantly. Most of the RTTs were near native execution (see Figure 6.5). However, some RTTs suffered a long delay, around 31 ms and 61 ms. This is result of the fast interrupt policy with recycled quantum. The recycled quantum was used to preserve the scheduling time coherency to the EDF algorithm. Despite the reset quantum scheme resulting in an improved interrupt response time, as shown in Section 6.6, it can not be used in combination with RT-VCPUs. Thus, the long delay is due to the insufficient time remaining in the current quantum. Moreover, the interrupt handling finishes only on the next Linux guest execution. On the other hand, Figure 6.16(a) showed that ADPCM execution time was affected, increasing significantly. The numerical results can be seen in line 3 of the Table 6.6.

Experiment 4. The fourth experiment consisted of combining the fast interrupt delivery policy with a non-preemptable RT-VM. This system configuration provides the best temporal isolation for RT-VCPUs while improving the response for external interrupts. This is possible because the fast interrupt delivery policy preempts the VCPUs for faster response time. However, the RT-VM is marked as non-preemptable resulting that its RT-VCPUs cannot be preempted. This guarantees a tenth of the CPU to the RT-VCPU executing the ADPCM algorithm.



(a) Histogram of encoding and decoding time for the ADPCM.



(b) Histogram for the RTT of the messages.

Figure 6.17: System response time with the fast interrupt deliver policy and a non-preemptable RT-VM.

Line 4 of Table 6.6 shows the numerical results for this experiment. The worst execution case to encoding and decoding the ADPCM algorithm is similar to the execution time presented in line 1 (non external interference). Moreover, the histogram for the execution time shown in Figure 6.17(a) is similar to the histogram in Figure 6.14. This shows the effectiveness of the temporal isolation provided by the hypervisor. In addition, the RTT of the messages improved significantly. The histogram in Figure 6.17(b) shows that some RTTs are spread between 0.7 and 31ms. This is caused by the RT-VCPU in the RT-VM. The ICMP messages arrived during its execution are delayed until the end of the quantum. The peak near 31ms and 61ms is due to the recycling quantum scheme, as explained before.

6.9 Final Considerations

This chapter studied the effectiveness of the hypervisor implementation. A sequence of experiments was performed in the SEAD-3 development board to evaluate different aspects of the hypervisor. One important aspect discussed was the footprint. A direct comparison with all other hypervisors studied is not possible because some of them do not mention their footprint. However, the Hellfire Hypervisor's footprint is acceptable for IoT devices, and is similar to the smallest hypervisors with a known footprint. The overhead impact on Linux was determined using benchmarks for virtualized and non-virtualized executions. User-land applications suffered lower impact than applications that use syscalls intensively. However, the overall overhead is small and comparable to other hypervisors. Moreover, the hypervisors present a low overhead for directly mapped devices. The fast interrupt delivery mechanism was evaluated showing the effectiveness of this policy. This experiment showed that the hardware-assisted virtualization can be used to improve responsiveness while keeping the simplicity of the implementation. The inter-VM communication mechanism presented a considerable overhead caused by the data copies between the VMs and the hypervisor. However, the usage of the hypervisor as a communication arbiter improves the security. Also, shared memory can be used to implement inter-VM communication with lower overhead. Finally, the real-time services were evaluated. The average time expended for RT-VCPU scheduling is lower than 100us in a system with 8 VCPUs. The temporal isolation was tested showing that the fast interrupt policy with non-preemptable RT-VMs is an effective method to combine real-time tasks and fast interrupt handling. The overall results are promising and prove that the hypervisor implementation is efficient in performance and responsiveness.

7. CONCLUSIONS

This dissertation started by defining the research goal. The objective was to verify hypothesis that is possible to have an embedded lightweight hypervisor capable of supporting the major embedded virtualization requisites. After the study of the state-of-the-art and the current trends in embedded virtualization, this hypothesis was considered true. Thus, the importance of the new hardware-assisted virtualization to accomplish this goal was considered. The hypothesis was refined into different research questions, which were addressed in the chapters of this dissertation. Now, the answers will be summarized. Then, concluding remarks and possible directions for future work will be presented.

First, the major existing embedded hypervisors were presented. This study provided an understanding the state of the art for embedded virtualization. Several different hypervisors were proposed for embedded virtualization, including modified versions of hypervisors for server virtualization. This study exposed the complexity of the embedded virtualization. The wide range of applications and the variety of embedded processors pushed the appearance of different hypervisor approaches. However, it was possible to identify trends in the current virtualization methods. The footprint is a recurrent challenge in embedded virtualization since its adoption for IoT devices require hypervisors with only a few dozen of kilobytes. Embedded virtualization was firmly based on para-virtualization due to the absence of hardware-assisted virtualization in the previous generations of embedded processors. Embedded processor families have adopted hardware for virtualization. Additionally, spatial isolation between VMs is critical to improve reliability and security. Finally, support for real-time on embedded virtualization is an obligatory feature because real-time is intrinsically related to embedded systems. This study highlighted the need for a different embedded virtualization approach to better fit the needs of embedded systems.

Based on these findings, the primary embedded virtualization requisites were determined. These requirements included the need to adoption full-virtualization using hardware-assisted virtualization. Moreover, previous experience with VHH showed that full-virtualization applied on hardware subsets without proper virtualization support results in excessive overhead. Thus, the model provides full-virtualization of the CPU (covered by hardware-assistance), and para-virtualization for I/O and extended services. The result was a hybrid model that combines full and para-virtualization. Hence, different aspects of embedded virtualization are covered by the model. For example, the model uses the hardware-assisted virtualization to provide spatial isolation while para-virtualization is used for inter-VM communication and to manage real-time services. Temporal isolation is provided by the separation of BE and RT-VCPUs. Finally, the major embedded virtualization features were accommodated in a well-defined virtualization model.

The proposed embedded virtualization model guided the development of a innovative hypervisor, called Hellfire Hypervisor, to the MIPS M5150 processor. This processor was chosen for different reasons. First, it implements the MIPS-VZ module for hardware-assisted virtualization. Second, the other hypervisors presented do not implement suitable support for MIPS virtualization. Lastly, an instruction accurate simulator and a hardware development board for M5150 was available to the GSE group. The development showed that the model can be implemented with relatively simple algorithms and a small number of lines of code resulting in a small footprint. For example, the virtual memory management was simplified and just a few instructions needed to be emulated. Moreover, almost all proposed embedded virtualization features were supported by the hypervisor. The only exception was the multicore support because the M5150 is a single-core processor. The evaluation of the hypervisor showed that it is efficient and presents low overhead. Additionally, the resulting footprint is acceptable for IoT applications.

7.1 Concluding Remarks

Based on the research work presented in the dissertation, the main conclusion is that the proposed virtualization model was successful to accommodate the major embedded system requirements. Additionally, the model proved to be efficiently implemented on a hardware platform. Finally, the implementation resulted in a hypervisor with performance and footprint comparable to the currently embedded hypervisors. While the majority of embedded hypervisors adopt para-virtualization exclusively, this work has shown that hardware-assisted virtualization associated with para-virtualization can benefit embedded systems by simplifying the hypervisor and improving performance.

The main contributions (including technical and scientific contributions) of this work are:

- A study of state-of-the-art for embedded virtualization and its main trends;
- The characterization of the major features required for embedded virtualization;
- A proposal of an embedded virtualization model;
- An implementation and evaluation of a hypervisor guided by the proposed model.

Lastly, from a more elevated perspective, the resulting hypervisor can benefit the industry and can also be used in commercial applications.

7.2 Future Research

There are many possible directions for future work based on this research. Firstly, this study revealed that hardware-assisted virtualization can be associated with embedded system requirements for the development of hypervisors that better fit embedded virtualization. Thus, when I/O virtualization is supported in embedded hardware, it may be necessary to update the virtualization model to adapt to this new hardware.

Security is a trend in embedded virtualization. IoT devices will be adopted in large scale in industrial and residential applications. These devices are susceptible to hacker attacks like any other device connected to the network. Thus, techniques to improve security must be adopted for the embedded hypervisors. For example, secure boot methods must be implemented to guarantee the authenticity of the hypervisor and VMs.

In server virtualization, VMs on the same host can perform different services for different users without any relation between them. On the other hand, embedded systems are customized devices with a determined goal. Thus, VMs in an embedded hypervisor can perform different services to accomplish a common goal. As a consequence, the VMs must interact using the hypervisor communication mechanism. Therefore, more efficient and secure communication services must be proposed.

Multicore processors are widely adopted for embedded devices. Thus, the proposed hypervisor must be ported to a multicore platform. However, multicore support adds concurrent execution at the hypervisor's kernel, which requires synchronization primitives. This increases significantly the hypervisor's kernel complexity, and it can impact on performance. Thus, the port to multicore processors must be carefully designed.

7.3 List of Publications

This section resumes all publications related to this dissertation in chronological order. Some of these publications are direct-related (D), i.e., publications resulted from this dissertation. The remainder of the publications are indirect-related (I), and they resulted from the work with the VHH.

- (D) Moratelli, C.; Filho, S.; Hessel, F."Exploring Embedded Systems Virtualization Using MIPS Virtualization Module," in Computing Frontiers (CF, 2016 ACM International Conference on), 16-18 May 2016.
- (D) Moratelli, C.; Filho, S.; Hessel, F."Hardware-assisted interrupt delivery optimization for virtualized embedded platforms," in Electronics, Circuits, and Systems (ICECS), 2015 IEEE International Conference on), 6-9 December 2015.
- (D) Zampiva, S.; Moratelli, C.; Hessel, F., "A hypervisor approach with real-time support to the MIPS M5150 processor," in Quality Electronic Design (ISQED), 2015 16th International Symposium on , vol., no., pp.495-501, 2-4 March 2015.
- (I) Moratelli, C.; Zampiva, S.; Hessel, F., "Full-virtualization on MIPS-based MPSOCs embedded platforms with real-time support," in Integrated Circuits and Systems Design (SBCCI), 2014 27th Symposium on , vol., no., pp.1-7, 1-5 Sept. 2014.
- (I) Aguiar, A.; Moratelli, C.; Sartori, M.; Hessel, F., "Adding virtualization support in MIPS 4Kc-based MPSoCs," in Quality Electronic Design (ISQED), 2014 15th International Symposium on , vol., no., pp.84-90, 3-5 March 2014.
- (I) Aguiar, A.; Moratelli, C.; Sartori, M.L.L.; Hessel, F., "A virtualization approach for MIPS-based MPSoCs," in Quality Electronic Design (ISQED), 2013 14th International Symposium on , vol., no., pp.611-618, 4-6 March 2013.
- (I) Aguiar, A.; Moratelli, C.; Sartori, M.L.L.; Hessel, F., "Hardware-assisted virtualization targeting MIPS-based SoCs," in Rapid System Prototyping (RSP), 2012 23rd IEEE International Symposium on , vol., no., pp.2-8, 11-12 Oct. 2012.

REFERENCES

[1] Abeni, L.; Buttazzo, G. “Integrating multimedia applications in hard real-time systems”. In: 19th IEEE Real-Time Systems Symposium, 1998, pp. 4–13.

[2] Aguiar, A.; Moratelli, C.; Sartori, M.; Hessel, F. “Adding virtualization support in mips 4kc-based mpsocs”. In: 15th International Symposium on Quality Electronic Design (ISQED), 2014, pp. 84–90.

[3] Aguiar, A. C. P. “On the virtualization of multiprocessed embedded systems”, Ph.D. Thesis, Pontifical Catholic University of Rio Grande do Sul - Faculty of Informatics, 2013.

[4] Airlines Electronic Engineering Committee. “Avionics Application Software Standard Interface”, Technical Report, 2006.

[5] Appleton, I. “The gsm protocol stack”. In: IEEE Colloquium on the Design of Digital Cellular Handsets, 1998, pp. 9/1–9/5.

[6] ARM. “ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition”, 2012.

[7] Asberg, M.; Forsberg, N.; Nolte, T.; Kato, S. “Towards real-time scheduling of virtual machines without kernel modifications”. In: 16th Conference on Emerging Technologies Factory Automation (ETFA), 2011, pp. 1–4.

[8] Avanzini, A.; Valente, P.; Faggioli, D.; Gai, P. “Integrating linux and the real-time erika os through the xen hypervisor”. In: 10th IEEE International Symposium on Industrial Embedded Systems (SIES), 2015, pp. 1–7.

[9] Barham, P.; Dragovic, B.; Fraser, K.; Hand, S.; Harris, T.; Ho, A.; Neugebauer, R.; Pratt, I.; Warfield, A. “Xen and the art of virtualization”, *SIGOPS Oper. Syst. Rev.*, vol. 37–5, Oct 2003, pp. 164–177.

[10] Beleg, G. “Scheduling operating-systems”, Technical Report Mat.-No.: 2854416, Technical University Dresden, 2007.

[11] Bellard, F. “Qemu, a fast and portable dynamic translator”. In: Annual Conference on USENIX Annual Technical Conference, 2005, pp. 41–41.

[12] Bovet, D.; Cesati, M. “Understanding The Linux Kernel”. Oreilly & Associates Inc, 2006.

[13] Brun, A.; Guo, C.; Ren, S. “A note on the edf preemption behavior in rate monotonic versus edf: Judgment day”, *Embedded Systems Letters, IEEE*, vol. 7–3, Sept 2015, pp. 89–91.

[14] Buttazzo, G. C. “Rate monotonic vs. edf: Judgment day”, *Real-Time Syst.*, vol. 29–1, Jan 2005, pp. 5–26.

[15] Chen, H.; Jin, H.; Hu, K.; Yuan, M. “Adaptive audio-aware scheduling in xen virtual environment”. In: IEEE/ACS International Conference on Computer Systems and Applications (AICCSA), 2010, pp. 1–8.

[16] Cheng, K.; Bai, Y.; Wang, R.; Ma, Y. “Optimizing soft real-time scheduling performance for virtual machines with srt-xen”. In: 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2015, pp. 169–178.

[17] Corbet, J.; Rubini, A.; Kroah-Hartman, G. “Linux Device Drivers, 3rd Edition”. O’Reilly Media, Inc., 2005.

[18] Cottet, F.; Mammeri, Z.; Delacroix, J.; Kaiser, C. “Scheduling in real-time systems”. Chichester, West Sussex, England: J. Wiley, 2002.

[19] Crespo, A.; Ripoll, I.; Masmano, M. “Partitioned embedded architecture based on hypervisor: The xtratum approach”. In: Dependable Computing Conference (EDCC), 2010 European, 2010, pp. 67–72.

[20] Cucinotta, T.; Anastasi, G.; Abeni, L. “Respecting temporal constraints in virtualized services”. In: 33rd Annual IEEE International in Computer Software and Applications Conference (COMPSAC ’09)., 2009, pp. 73–78.

[21] Cummiskey, P.; Jayant, N.; Flanagan, J. “Adaptive quantization in differential pcm coding of speech”, *Bell System Technical Journal, The*, vol. 52–7, Sept 1973, pp. 1105–1118.

[22] Dall, C.; Nieh, J. “Kvm/arm: The design and implementation of the linux arm hypervisor”. In: 19th International Conference on Architectural Support for Programming Languages and Operating Systems, 2014, pp. 333–348.

[23] Ding, X.; Ma, Z.; Da, X. “Dynamic time slice of credit scheduler”. In: IEEE International Conference on Information and Automation (ICIA), 2014, pp. 654–659.

[24] Duda, K. J.; Cheriton, D. R. “Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler”, *SIGOPS Oper. Syst. Rev.*, vol. 33–5, Dec 1999, pp. 261–276.

[25] Elphinstone, K.; Heiser, G. “From l3 to sel4 what have we learnt in 20 years of l4 microkernels?” In: 24th ACM Symposium on Operating Systems Principles, 2013, pp. 133–150.

[26] Freescale. “e500mc Core Reference Manual”, Technical Report, 2013.

[27] Gates, M.; Tirumala, A.; Dugan, J.; Gibbs, K. “Iperf version 2.0.0”. NLANR applications support, University of Illinois at Urbana-Champaign, Urbana, IL, USA, 2004.

[28] Gubbi, J.; Buyya, R.; Marusic, S.; Palaniswami, M. “Internet of things (iot): A vision, architectural elements, and future directions”, *Future Gener. Comput. Syst.*, vol. 29–7, Sep 2013, pp. 1645–1660.

[29] Heath, S. “Embedded Systems Design”. Newton, MA, USA: Butterworth-Heinemann, 2002, 2nd ed..

[30] Heinrich, J. “MIPS R4000 Microprocessor User’s Manual”, Technical Report, 1994.

[31] Heiser, G. “The role of virtualization in embedded systems”. In: 1st workshop on Isolation and integration in embedded systems, 2008, pp. 11–16.

[32] Heiser, G.; Leslie, B. “The okl4 microvisor: Convergence point of microkernels and hypervisors”. In: 1st ACM Asia-pacific Workshop on Workshop on Systems, 2010, pp. 19–24.

[33] Hesselink, W. H.; Tol, R. M. “Formal feasibility conditions for earliest deadline first scheduling”, Technical Report, 1994.

[34] Hu, Y.; Long, X.; Zhang, J.; He, J.; Xia, L. “I/o scheduling model of virtual machine based on multi-core dynamic partitioning”. In: 19th ACM International Symposium on High Performance Distributed Computing, 2010, pp. 142–154.

[35] Imagination Technologies Ltd. “MIPS Architecture For Programmers Volume I-B: Introduction to the microMIPS32 Architecture”, Technical Report, 2013.

[36] Imagination Technologies Ltd. “MIPS32® Architecture for Programmers Volume IV-i: Virtualization Module of the MIPS32® Architecture”, Technical Report, 2013.

[37] Imperas Software Limited. “Ovpsim and imperas cpumanager user guide”, Technical Report, 2015.

[38] Kaiser, R. “Combining Partitioning and Virtualization for Safety-Critical Systems”, Technical Report, 2009.

[39] Kanda, W.; Yumura, Y.; Kinebuchi, Y.; Makijima, K.; Nakajima, T. “Spumone: Lightweight cpu virtualization layer for embedded systems”. In: IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC ’08), 2008, pp. 144–151.

[40] Kato, S.; Rajkumar, R.; Ishikawa, Y. “A loadable real-time scheduler framework for multicore plafforms”. In: 6th IEEE International Conferece on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010.

[41] Kiszka, J. “Towards linux as a real-time hypervisor”. In: 11th Real-Time Linux Workshop, 2009.

[42] Lee, J.; Xi, S.; Chen, S.; Phan, L. T. X.; Gill, C.; Lee, I.; Lu, C.; Sokolsky, O. “Realizing compositional scheduling through virtualization”. In: 18th Real Time and Embedded Technology and Applications Symposium, 2012, pp. 13–22.

[43] Lee, M.; Krishnakumar, A. S.; Krishnan, P.; Singh, N.; Yajnik, S. “Supporting soft real-time tasks in the xen hypervisor”. In: 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 2010, pp. 97–108.

[44] Liedtke, J. “A persistent system in real use-experiences of the first 13 years”. In: 3rd International Workshop on Object Orientation in Operating Systems, 1993, pp. 2–11.

[45] Liedtke, J. “On micro-kernel construction”. In: 50th ACM Symposium on Operating Systems Principles, 1995, pp. 237–250.

[46] Lindholm, T.; Yellin, F. “Java Virtual Machine Specification”. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999, 2nd ed..

[47] Loongson Technology Corp. Ltd. “Loongson 3a processor manual”, 2009.

[48] Masrur, A.; Drossler, S.; Pfeuffer, T.; Chakraborty, S. “Vm-based real-time services for automotive control applications”. In: 16th International Conference on Embedded and Real-Time Computing Systems and Applications, 2010, pp. 218–223.

[49] Masrur, A.; Pfeuffer, T.; Geier, M.; Drössler, S.; Chakraborty, S. “Designing vm schedulers for embedded real-time applications”. In: 7th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2011, pp. 29–38.

[50] MIPS Technologies, Inc. “MIPS SEAD-3 Board User’s Manual”, Technical Report, 2010.

[51] Mitake, H.; Kinebuchi, Y.; Courbot, A.; Nakajima, T. “Coexisting real-time os and general purpose os on an embedded virtualization layer for a multicore processor”. In: ACM Symposium on Applied Computing, 2011, pp. 629–630.

[52] Mitake, H.; Lin, T.-H.; Kinebuchi, Y.; Shimada, H.; Nakajima, T. “Using virtual cpu migration to solve the lock holder preemption problem in a multicore processor-based virtualization layer for embedded systems”. In: 18th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012, pp. 270–279.

[53] Moratelli, C.; Zampiva, S.; Hessel, F. “Full-virtualization on mips-based mpsocs embedded platforms with real-time support”. In: 27th Symposium on Integrated Circuits and Systems Design (SBCCI), 2014, pp. 1–7.

[54] Muench, D.; Paulitsch, M.; Herkersdorf, A. “Temporal separation for hardware-based i/o virtualization for mixed-criticality embedded real-time systems using pcie sr-iov”. In: 27th International Conference on Architecture of Computing Systems (ARCS), 2014, pp. 1–7.

[55] Noergaard, T. “Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers”. Newnes, 2005.

[56] Ongaro, D.; Cox, A. L.; Rixner, S. “Scheduling i/o in virtual machine monitors”. In: 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 2008, pp. 1–10.

[57] Ousterhout, J. "Scheduling techniques for concurrent systems". In: 3rd International Conference on Distributed Computing Systems, 1982, pp. 22–30.

[58] Palopoli, L.; Cucinotta, T.; Marzario, L.; Lipari, G. "Adaptive quality of service architecture", *Softw. Pract. Exper.*, vol. 39–1, Jan 2009, pp. 1–31.

[59] Patel, A.; Daftedar, M.; Shalan, M.; Watheq El-Kharashi, M. "Embedded hypervisor xvisor: A comparative analysis". In: 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2015, pp. 682–691.

[60] Popek, G. J.; Goldberg, R. P. "Formal requirements for virtualizable third generation architectures", *Commun. ACM*, vol. 17–7, Jul 1974, pp. 412–421.

[61] prpl Fundation. "Security Guidance for Critical Areas of Embedded Computing", Technical Report, 2015.

[62] RedHat. "Kvm - kernel based virtual machine", Technical Report, RedHat Enterprise, 2013.

[63] Renesas Techology. "Sh-4a software manual", Technical Report, Renesas Techology, 2004.

[64] Russell, R. "Virtio: Towards a de-facto standard for virtual i/o devices", *SIGOPS Oper. Syst. Rev.*, vol. 42–5, Jul 2008, pp. 95–103.

[65] Sandstrom, K.; Vulgarakis, A.; Lindgren, M.; Nolte, T. "Virtualization technologies in embedded real-time systems". In: 18th Conference on Emerging Technologies Factory Automation (ETFA), 2013, pp. 1–8.

[66] Saranya, N.; Hansdah, R. "Dynamic partitioning based scheduling of real-time tasks in multicore processors". In: 18th International Symposium on Real-Time Distributed Computing (ISORC), 2015, pp. 190–197.

[67] Sefraoui, O.; Aissaoui, M.; Eleuldj, M. "Article: Openstack: Toward an open-source solution for cloud computing", *International Journal of Computer Applications*, vol. 55–3, October 2012, pp. 38–42.

[68] Sha, L.; Lehoczky, J. P.; Rajkumar, R. "Solutions for some practical problems in prioritized preemptive scheduling". In: IEEE Real-Time Systems Symposium'86, 1986, pp. 181–191.

[69] Singh, J. “An algorithm to reduce the time complexity of earliest deadline first scheduling algorithm in real-time system”, *CoRR*, vol. abs/1101.0056, 2011.

[70] Smith, J.; Nair, R. “Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)”. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[71] SPARC International Inc. “The SPARC Architecture Manual - Version 8”, Technical Report, 1992.

[72] Sprunt, B.; Sha, L.; Lehoczky, J. P. “Aperiodic task scheduling for hard real-time systems”, *Real-Time Systems*, vol. 1–1, 1989, pp. 27–60.

[73] Steinberg, U.; Kauer, B. “Nova: A microhypervisor-based secure virtualization architecture”. In: 5th European Conference on Computer Systems, 2010, pp. 209–222.

[74] STMicroelectronics. “Loongson2f user manual”, 2007.

[75] Strosnider, J. K.; Lehoczky, J. P.; Sha, L. “The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments”, *IEEE Trans. Comput.*, vol. 44–1, Jan 1995, pp. 73–91.

[76] Tai, Y.; Cai, W.; Liu, Q.; Zhange, G. “Kvm-loongson: An efficient hypervisor on mips”. In: 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2013, pp. 1016–1022.

[77] Tanenbaum, A. S.; Wetherall, D. J. “Computer Networks”. Prentice Hall, 2011, 5th ed..

[78] Technologies, M. “Mips32® m5150 processor core family software user’s manual”, July 2014.

[79] Tommaso, C.; Dhaval, G.; Dario, F.; Checconi, F. “Providing performance guarantees to virtual machines using real-time scheduling”. In: Euro-Par 2010 Parallel Processing Workshops, 2011, pp. 657–664.

[80] Trujillo, S.; Crespo, A.; Alonso, A. “Multipartes: Multicore virtualization for mixed-criticality systems”. In: Euromicro Conference on Digital System Design (DSD), 2013, pp. 260–265.

[81] Uhlig, V.; LeVasseur, J.; Skoglund, E.; Dannowski, U. “Towards scalable multiprocessor virtual machines”. In: 3rd Virtual Machine Research and Technology Symposium, 2004, pp. 43–56.

[82] Vigeant, G.; Beaulieu, A.; Givigi, S. “Hard real-time scheduling on a multicore platform”. In: 9th Annual IEEE International Systems Conference (SysCon), 2015, pp. 324–331.

[83] Waldspurger, C.; Rosenblum, M. “I/o virtualization”, *Commun. ACM*, vol. 55–1, Jan 2012, pp. 66–73.

[84] Xi, S.; Wilson, J. a. C. L.; Gill, C. “Rt-xen: Towards real-time hypervisor scheduling in xen”. In: International Conference on Embedded Software (EMSOFT), 2011, pp. 39–48.

[85] Xi, S.; Xu, M.; Lu, C.; Phan, L. T. X.; Gill, C.; Sokolsky, O.; Lee, I. “Real-time multi-core virtual machine scheduling in xen”. In: 14th International Conference on Embedded Software, 2014, pp. 27:1–27:10.

[86] Xia, T.; Prévotet, J.-C.; Nouvel, F. “Microkernel dedicated for dynamic partial reconfiguration on arm-fpga platform”, *SIGBED Rev.*, vol. 11–4, Jan 2015, pp. 31–36.

[87] Xia, T.; Prevotet, J.-C.; Nouvel, F. “Mini-nova: A lightweight arm-based virtualization microkernel supporting dynamic partial reconfiguration”. In: IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015, pp. 71–80.

[88] Yang, J.; Kim, H.; Park, S.; Hong, C.; Shin, I. “Implementation of compositional scheduling framework on virtualization”, *SIGBED Rev.*, vol. 8–1, Mar 2011, pp. 30–37.

[89] Yoo, S.; Yoo, C. “Real-time scheduling for xen-arm virtual machines”, *Mobile Computing, IEEE Transactions on*, vol. 13–8, Aug 2014, pp. 1857–1867.

[90] Zampiva, S. “Um hypervisor com suporte a tempo-real para a arquitetura MIPS32R5”, Master’s Thesis, Pontifical Catholic University of Rio Grande do Sul - Faculty of Informatics, 2015.

[91] Zeng, L.; Wang, Y.; Shi, W.; Feng, D. “An improved xen credit scheduler for i/o latency-sensitive applications on multicores”. In: International Conference on Cloud Computing and Big Data (CloudCom-Asia), 2013, pp. 267–274.

[92] Zhang, D.; Liu, D.; Liang, L.; Yao, L.; Zhong, K.; Shao, Z. “Nv-cfs: Nvram-assisted scheduling optimization for virtualized mobile systems”. In: 7th International Symposium on Cyberspace Safety and Security (CSS), 2015, pp. 802–805.

[93] Zhou, R.; Ai, Z.; Yang, J.; Chen, Y.; Li, J.; Zhou, Q.; Li, K.-C. “A hypervisor for mips-based architecture processors - a case study in loongson processors”. In: IEEE International Conference on High Performance Computing and Communications, 2013, pp. 865–872.

[94] Zuo, B.; Chen, K.; Liang, A.; Guan, H.; Zhang, J.; Ma, R.; Yang, H. “Performance tuning towards a kvm-based low latency virtualization system”. In: 2nd International Conference on Information Engineering and Computer Science (ICIECS), 2010, pp. 1–4.

APPENDIX A – VIRTUAL MACHINE AND VIRTUAL CPU DATA STRUCTURES

This appendix shows the data structures used to built the virtual machine and the VCPU abstractions.

A.1 Virtual Machine Data Structure

The fast_int is given by the developers, and it determines which interrupts are eligible to the fast interrupt delivery policy in the VM. During system initialization, for each VM, the hypervisor makes an or bit-wise operation over the fast_int with all other VMs resulting in the value of apply_fast_int variable. The apply_fast_int is the interrupt mask. It determines which interrupts are enabled during the VM's executing allowing the VM to be preempted by the hypervisor. Non-preemptable VMs have the apply_fast_int variable equal to zero (0).

```
typedef struct vm_t {
        unsigned int id; /*VM ID*/
        unsigned int base_addr; /*Base address where the contiguous
        mapping starts.*/
        unsigned int size;      /* VM size in bytes. */
        linkedlist_t vcpus; /*List of VCPUs associated to the VM.*/
        unsigned int os_type; /* OS executing in the VM.*/
        unsigned int ntlbent; /* Number of TLB entries. */
        unsigned int init;      /* Flag to indicate if the VM is
        already initialized. */
        unsigned int fast_int; /* Interrupts market to the fast
        interrupt delivery to this VM. */
        unsigned int apply_fast_int; /* Interrupts that can cause
        preemption of this VM.*/
        unsigned int non_preemptable; /* Non preemptable VM. */

        struct tlbentry *tlbentries; /* TLB entries. Gives the
        information of the valid memory regions for this VM. */
}vm_t;
```

A.2 VCPU Data Structure

The message_t data structure defines the internal message formatting to the hypervisor communication mechanism. The message is kept in a circular buffer, defined in the message_buffer_t data structure, until the receiver is able to receive it. Each VCPU has its own circular buffer dedicated to its messages. The message size (MESSAGE_SZ) and the number of messages in the circular buffer (MESSAGELIST_SZ) impacts directly in the hypervisor footprint.

```
typedef struct{
        unsigned int source_id; /*ID of the sender VM.*/
        unsigned int size; /*Size of the message.*/
        unsigned int message[MESSAGE_SZ]; /*Message.*/
} message_t;

/*Circular buffer.*/
typedef struct {
        unsigned int in;
        unsigned int out;
        unsigned int num_messages;
        message_t message_list[MESSAGELIST_SZ];
} message_buffer_t;


typedef struct vcpu_t {
        unsigned int id; /*VCPU ID.*/
        unsigned int rootcounter; /*counter register of the root
        context.*/
        unsigned int offseTelapsedTime; /*Elapsed time to determine
        how long the VM is in the waiting queue.*/
```

```c
        unsigned int gprshadowset; /*GRP shadow page. */
        unsigned int cp0_registers[32][4]; /*CP0 registers */
        unsigned int gp_registers[34];  /*CPU registers */
        unsigned int guestclt2; /* */
        unsigned int pip; /* Indicates which interrupts are
        allowed to the interrupt pass-through */
        vm_t *vm; /*VM data structure */
        unsigned int pc; /*Program counter */
        unsigned int sp; /* Stack pointer */
        unsigned int gp; /* Global pointer */
        message_buffer_t messages; /* Circular buffer. */
}vcpu_t;
```