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

# ON THE VIRTUALIZATION OF MULTIPROCESSED EMBEDDED SYSTEMS

ALEXANDRA DA COSTA PINTO DE AGUIAR

Dissertation presented as partial requirement for obtaining the degree of Ph. D. in Computer Science at Pontifical Catholic University of Rio Grande do Sul.

Advisor: Prof. PhD. Fabiano Passuelo Hessel

**Porto Alegre**
**2014**

**Dados Internacionais de Catalogação na Publicação (CIP)**

A282o Aguiar, Alexandra da Costa Pinto de
On the virtualization of multiprocessed embedded systems / Alexandra da Costa Pinto de Aguiar. – Porto Alegre, 2014.
109 p.

Tese (Doutorado) – Fac. de Informática, PUCRS.
Orientador: Prof. Fabiano Passuelo Hessel.

1. Informática. 2. Máquinas Virtuais. 3. Microprocessadores. I. Hessel, Fabiano Passuelo. II. Título.

CDD 004.36

**Ficha Catalográfica elaborada pelo**
**Setor de Tratamento da Informação da BC-PUCRS**

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

# TERMO DE APRESENTAÇÃO DE TESE DE DOUTORADO

Tese intitulada *"On the virtualization of multiprocessed embedded systems"*, apresentada por Alexandra da Costa Pinto de Aguiar, como parte dos requisitos para obtenção do grau de Doutor em Ciência da Computação, aprovada em 30/08/2013 pela Comissão Examinadora:

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

Prof. Dr. Tiago Coelho Ferreto — PPGCC/PUCRS

Prof. Dr. Rodolfo Jardim de Azevedo — UNICAMP

Prof. Dr. Rômulo Silva de Oliveira — UFSC

Homologada em 24/04/2014 conforme Ata No. 006 pela Comissão Coordenadora.

Prof. Dr. Luiz Gustavo Leão Fernandes
Coordenador.

PUCRS
**Campus Central**
Av. Ipiranga, 6681 – P. 32 – sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320-3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

I dedicate this work and everything it represents to my parents. (Dedico este trabalho e tudo que ele representa a meus pais.)

"Stay hungry, stay foolish."
(Steve Jobs)

# ACKNOWLEDGMENTS

This text is written in Brazilian Portuguese, so my beloved ones can read it eventually.

Este momento é sempre bastante esperado. Quando se chega no ponto de agradecer alguém é porque todo (ou uma parte significativa) o caminho foi percorrido. É claro que a vontade intrínseca de cada ser o ajuda a conquistar seus objetivos... mas não há quem alcance qualquer nível de sucesso sem ajuda do outro. E agora, é chegado o momento de agradecer.

À Deus, ao Universo, a esta força misteriosa e superior que me abençoa constantemente em tantos aspectos da minha vida. A começar pela maior dádiva que se pode ter: uma família. Meus pais, incansáveis. Ajudando em absolutamente tudo o que podiam e, por vezes, até exagerando e se doando mais do que deviam. Bárbara Maix disse: "Mostremos com exemplos aquilo que com palavras ensinamos". E posso com 100% de certeza afirmar: não há no mundo melhores exemplos de pais do que vocês. Amo muito e o agradecimento por tudo será sempre eterno.

À meu maninho querido, Muca, que me surpreende constantemente com seu desprendimento das coisas que eu não consigo me desprender. Sempre trazendo alegria, irreverência e, por que não, constrovérsia à minha vida... afinal, irmão é pra essas coisas... te amo piá.

Aos meus "meio-irmãos" que estão sempre enviando energias positivas para que tudo se encaminhe bem, ainda que a correria da vida moderna nos impeça de conversar mais e termos mais tempo em convívio.

Ao meu orientador, Fabiano, que me acolheu nesta longa jornada desde que vim "do interior" sempre com paciência, liderança e conselhos tão necessários. Definitivamente, eu tive o melhor orientador que poderia querer.

À Felipe e Sérgio por serem tão parceiros e não me jogarem do sétimo andar depois de mais um "gente, vamo fazer esse paper"... Aos colegas Marcos e Carlos que tiveram participação direta na construção deste trabalho que, definitivamente, não poderia ser feito sem a parceria e colaboração de vocês. Aos demais colegas do laboratório, que sempre ajudaram quando necessário. A diversos colegas professores que aguentaram por tanto tempo a frase "hoje não, tem a tese".

Agradeço em especial à Andriele e ao Ricardo que tantas vezes toparam tomar um café no térreo apenas para amenizar o estresse do trabalho acumulado enquanto falávamos de algum seriado, filme ou ator que eu havia me apaixonado. À Sabrina que se mostrou uma grande amiga nos últimos anos e, sim Sá, ainda vamos ter tempo para mais coisas :) Ao Rafael que está passando por tudo que eu passei com dois anos de atraso... se prepara :P Ao Marcelo por ter sido um exemplo de persistência e um grande amigo :D

A muitos amigos de longa data que finalmente vão poder parar de perguntar "e a tese, como vai?!" :-) Aos meus primos, tios, tias e sobrinhas com quem não pude estar tão presente quanto gostaria em função desta imensa "empreitada".

Pode parecer ridículo aos que não entendem esta relação, mas aos meus cachorros. Primeiro pro Zé que teve que ser doado ao longo desta jornada e acabou se adaptando tanto à casa da vó que nem voltou. Aos meus bebês mais novos, Tony e Jorge, que pegaram os últimos meses loucos disso tudo.

Às minhas séries e filmes favoritos que ajudaram a relaxar quando não havia outra alternativa. À minha biblioteca imensa de músicas que faz parte da minha vida quase que na mesma proporção do oxigênio.

Infelizmente, não conseguiria expressar os nomes de todas as pessoas que deveria. Apenas que, um doutorado pode ser comparado à uma maratona. É você que treina, que sofre as contusões, que precisa persistir. Mas você não alcança à linha de chegada sem um bom técnico, sem bons tênis, sem água, sem pessoas te animando e incentivando quando cada metro parece insuperável. Talvez você nem lembre de todas essas pessoas, mas elas estiveram lá. Talvez, você não tenha como agradecer a cada um individualmente, mas todos fizeram parte do seu sucesso. Assim, eu reconheço aqui a participação de cada um que me ajudou a chegar até aqui. E, a todos que tiveram participação nisso tudo, meu mais sincero <muito obrigada>.

# VIRTUALIZAÇÃO EM SISTEMAS EMBARCADOS MULTIPROCESSADOS

## RESUMO

Virtualização surgiu como novidade em sistemas embarcados tanto no meio acadêmico quanto para o desenvolvimento na indústria. Entre suas principais vantagens, pode-se destacar aumento: (i) na qualidade de projeto de software; (ii) nos níveis de segurança do sistema; (iii) nos índices de reuso de software, e; (iv) na utilização de hardware. No entanto, ainda existem problemas que diminuíram o entusiasmo com relação ao seu uso, já que existe um overhead implícito que pode impossibilitar seu uso. Assim, este trabalho discute as questões relacionadas ao uso de virtualização em sistemas embarcados e apresenta estudos voltados para que arquiteturas MIPS multiprocessadas tenham suporte à virtualização.

**Palavras-Chave:** máquinas virtuais, hypervisores, virtualização, MPSoC.

# ON THE VIRTUALIZATION OF MULTIPROCESSED EMBEDDED SYSTEMS

## ABSTRACT

Virtualization has become a hot topic in embedded systems for both academia and industry development. Among its main advantages, we can highlight (i) software design quality; (ii) security levels of the system; (iii) software reuse, and; (iv) hardware utilization. However, it still presents constraints that have lessened the excitement towards itself, since the greater concerns are its implicit overhead and whether it is worthy or not. Thus, we discuss matters related to virtualization in embedded systems and study alternatives to multiprocessed MIPS architecture to support virtualization.

**Keywords:** Virtual machines, hypervisors, virtualization, MPSoC.

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF ACRONYMS

ABI – Application Binary Interface

ADU – Application Domain Unit

AMP – Asymmetric Multiprocessing

AMT – Active Management Technology

API – Application Programming Interface

ASID – Address Space IDentifier

BE – Best-effort

DBT – Dynamic Binary Translation

DMA – Direct Memory Access

EDF – Earliest Deadline First

EPC – Exception Program Counter

EPT – Extended Page Tables

ES – Embedded Systems

FOTA – Firmware Over the Air

FPGA – Field Programmable Gate Array

GPL – General-purpose License

HAL – Hardware Abstraction Layer

HFOS – HellfireOS

HVM – Hardware-assisted Virtual Machine

IMA – Integrated Modular Avionics

IP – Intellectual Property

IPC – Inter-process communication

ISA – Instruction-Set Architecture

LL – Load Linked

LLVM – Low Level Virtual Machine

LOC – Lines of Code

MILS – Multiple Independent Levels of Security

MIPC – Multicore/multi-OS Interprocess Communication

MMU – Memory Management Unit

NI – Network Interface

NOCS – Networks-on-Chip

OS – Operating System

OVP – Open Virtual Platform

RT – Real-time

RTOS – Real-time Operating System

SC – Store Conditional

SMP – Symmetric Multiprocessing

SVM – Secure Virtual Machine

TC – Translator Cache

TCO – Total Cost of Ownership

TLB – Translation Lookaside Buffer

TXT – Trusted Execution Technology

VCPU – Virtual CPU

VHH – Virtual-Hellfire Hypervisor

VT – Virtualization Technology

WCET – Worst Case Execution Time

# CONTENTS

# 1. INTRODUCTION

Multi-purpose Embedded Systems (ES) are now considered as a solid reality in everyday's lives since each more new and exciting features and devices are available to customers. However, such a wide range of applications impacts directly on their design, constraints and goals. Also, embedded systems are increasingly counting on typical general-purpose computers' characteristics, such as the possibility for the final user to develop and download new applications onto the device throughout its lifetime [ZM00]. Within this context, embedded software itself has become a subjacent layer in the design flow unlike older approaches where hardware itself, or custom firmware used to be more prominent.

Despite this shift in perception, some traditional differences between general-purpose and embedded systems still remain [LP05]. Usually timing constraints are present along with a limited energy consumption budget and limitations regarding memory size. Still, the wide variety of predominant architectures present in ESs also contributes to increasing the difficulty in their design.

A relevant characteristic of embedded systems is the broad use of multiprocessed solutions, notably in the last decade [JTW05]. Combined with the aforementioned particularities, the use of multicore hardware requires a true change in the way embedded developers design their systems.

Therefore, virtualization, rather a successful technique exclusively applied on general-purpose computers, arises as a possible solution to many of these problems, as it can increase ESs' performance, software design quality and security levels while reducing their manufacturing costs [Hei11]. However, due to the typical embedded constraints mentioned earlier, a lot of effort has been spent in order to demonstrate that virtualization can indeed improve the overall system quality at a reasonable cost [IO07], [BDB+08], [SCH+09], [CR10], [AFNK11].

Among all these efforts to apply virtualization on embedded systems, the main problems concern some of these systems' characteristics and the functionalities required by a virtualized platform, which can be conflicting most of the time. For example, Heiser [Hei08] highlights the need to run unmodified guest OS and applications besides providing strong spatial isolation to improve security. In [AG09], the need for low overhead components are said to be fundamental. The main problem, however, is that it is very difficult to target all such constraints at once.

Another challenge is that these conflicting needs have a strong relationship with the hypervisor's implementation as it depends completely on the underlying hardware. Thus, the architecture itself (and the characteristics of its Instruction-Set Architecture - ISA) can make the use of embedded virtualization feasible.

## 1.1 Motivational Examples for Embedded Virtualization

The first case for virtualization on embedded systems consists of allowing several operating systems to be executed concurrently. Here, the use of different operating systems can be specially classified in two different scenarios:

1. when legacy software must co-exist with current and incompatible applications; and
2. when it is desired to separate real-time software from user interface applications, by using either different or separated Operating Systems (OSs).

In this first case, software development quality can be increased, since the designer can choose, among several OSs, the one most suitable for the application or even the one with the best cost/performance ratio. Moreover, the time required to develop an application can be reduced, since in the case it is offered the support to a given OS, any former application can be reused without the onus of rewriting it [Sub09]. Figure 1.1 shows the basic use of virtualization, with the consolidation of several OSs in the same hardware platform.

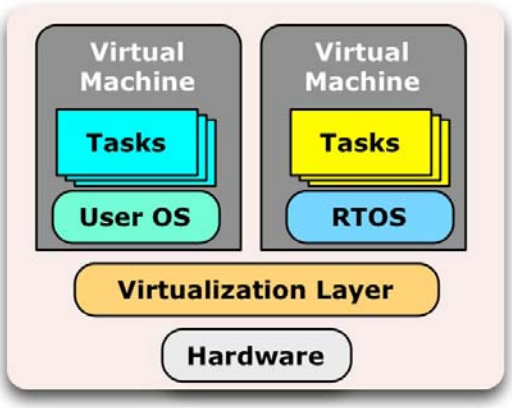


Figure 1.1 – Consolidation of several OSs using virtualization

Furthermore, this approach offers advantages in achieving a unified software architecture that can be executed on multiple hardware platforms. In this case, a current issue in embedded systems - software portability - could be widely affected and developers would be able to satisfy the increasingly restricted time-to-market. The combination of real-time, legacy and general-purpose operating systems in the same device can be easily achieved with virtualization, as Figure 1.2 shows.

Besides, security is another important issue to be managed by virtualization, since it provides an environment that protects and encapsulates embedded operating systems and other software components. Initially, the idea of using an application-specific operating system apart from the Real-Time OS (RTOS) is encouraged as user attacks would only be able to cause damages at the user OS, thus keeping the RTOS and specific system components safe.

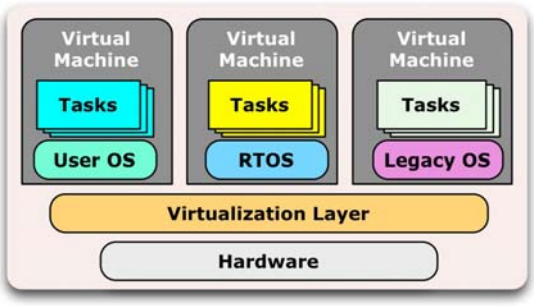


Figure 1.2 – Legacy software coexist with newer applications

This approach is depicted in Figure 1.3, where the scenario containing separate OSs and an ongoing user attack is shown. Nevertheless, in order to actually guarantee that virtualization will improve security, the underlying virtualization layer has to be significantly more secure than the guest OS.

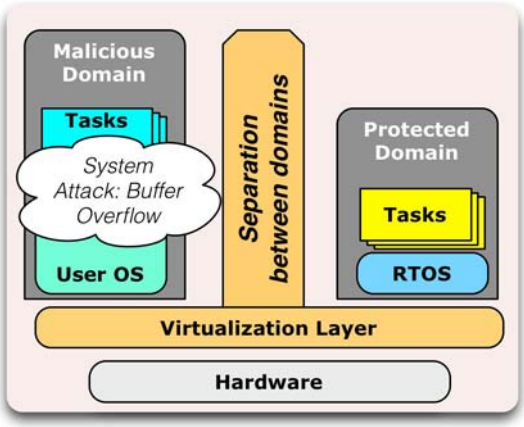


Figure 1.3 – User attack blocked by virtualization's inherent isolation

In multi-core architectures, there are different ways of utilizing the many system processors. A very common arrangement consists of running a single OS onto the processors, thus creating a symmetric multiprocessing (SMP) configuration. This approach brings the main advantage of making load balancing across the processors straightforward. When virtualization is adopted in multiprocessed architectures, there is the possibility of reducing the total number of physical processors. Figure 1.4 illustrates a configuration in which this is possible.

However, using different and multiple OSs in the same MPSoC can be an attractive option. Virtualization allows different virtual machines to provide services independently, which is a safer approach. A single software in control represents a single failure point and whenever the system crashes all the cores must be restarted. Although the hypervisor may represent a single point of failure it can be a safer approach provided it is implemented carefully.

In this case, an asymmetric multiprocessing (AMP) configuration is used, where each processor has its own separate OS, responsible for scheduling its own tasks. AMP is a configuration

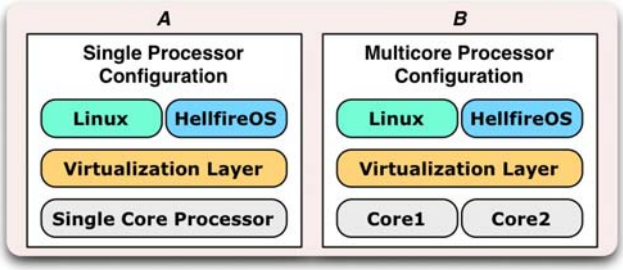


Figure 1.4 – Different processor configurations: a) single physical core; b) multicore

that takes advantages of virtualization since it provides the arbitration of resources' usage between the virtual machines, avoiding the user OS to cause unexpected behavior on the RTOS [Her09]. If no virtualization is used, the only way to achieve such separation is by doing it manually, which is complex and more suitable to errors. Still, the virtualization layer can be responsible for mapping every virtual machine on each core of the multi-core processor or even map a single OS onto multiple cores, creating an SMP subset of cores [SLB07].

Thus, the reliability of AMP systems can be increased by guaranteeing resources (memory, devices) separation and the ability to, independently, restart virtual machines, as depicted in Figure 1.5. It is important to notice that this can also be used when an AMP subset of processors is present in the MPSoC.

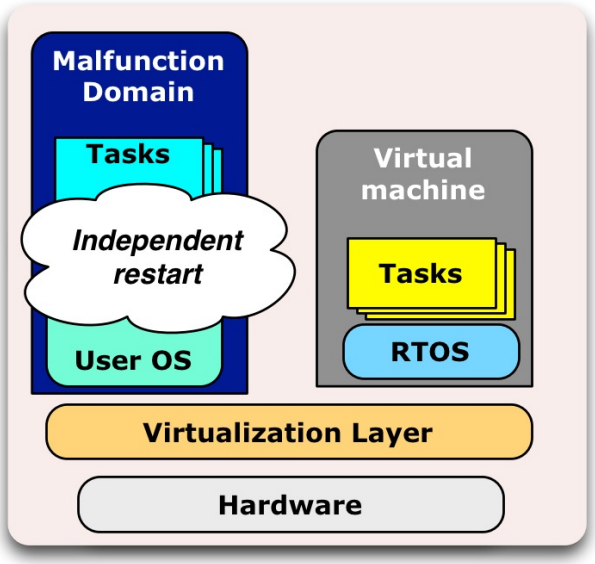


Figure 1.5 – Improved reliability for AMP architectures

When working with SMP configurations, where several equal OSs are executing in different virtual machines, workload balancing can be improved, by migrating application between machines. It is then possible to exchange functionality between virtual machines, providing the opportunity for reuse and innovation. This is showed in Figure 1.6. The advantages of migration in embedded systems has been widely proved throughout the years [SP09], [BABP06], [NAMV05].

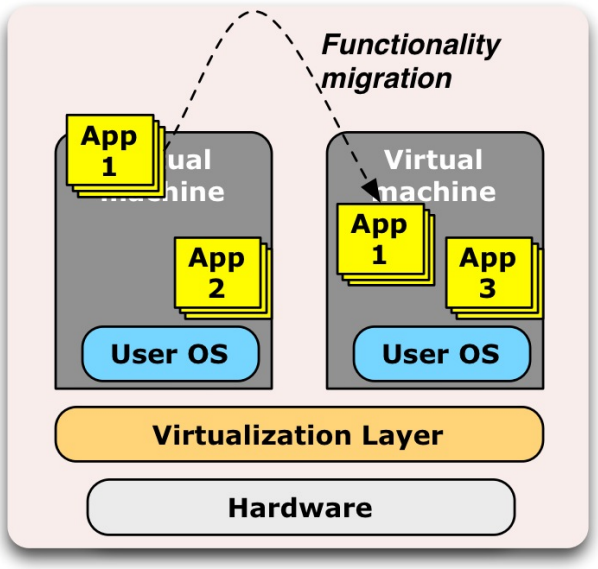


Figure 1.6 – Migration between virtual machines using the same OS on the same ISA

All these scenarios are suitable enough to be used in non-critical embedded systems, as multimedia mobile devices [ABK09], [YKY10]. Specially, as consumer demands continues to grow, stricter time-to-market tends to be more present and virtualization can enable their achievement.

Despite of typically being a very resource consuming technique, virtualization has been considered for some researchers to be used in critical embedded systems, such as in avionics [KW08]. Usually, security sensitive or mission critical systems need a protected environment [For10]. Then, these sensitive parts have their own OS and the virtualization layer separates them from non-trusted OSs and applications.

The separation provided by virtualization allows other scenarios. For example, when some parts of the system are required to boot up faster than others. For instance, a car or a camera must have some of their functions available at a really fast pace (tens of milliseconds after power on). A general-purpose OS will take much longer, therefore, virtualization can separate the functions to be run in exclusive virtual machines, boosting their boot time.

Separation also allows license protection to be achieved because proprietary application can be completely isolated from GPL OS. Intellectual Property (IP) protection can rely on virtualization's inherit separation, since private modules are safe from user's inappropriate handling. Firmware over the air (FOTA) upgrades could also be easier to be made with virtualization besides allowing that only a given part of the system reboots after the upgrade. Easier migration would allow extensive use cases for pervasive computers, as virtual machines could migrate among different devices, leading to a whole new level of remote device usage [Rud09].

## 1.2 Research goal and questions

**Research goal:** The goal of this research consists in studying the virtualization technique to the point of proposing a suitable way for it to be adopted in multiprocessed embedded systems.

For this research goal to be achieved, two main research questions had to be made along the way.

1. *What is the impact of the architecture diversity typically found in ESs in employing the virtualization technique?*
2. *Considering the existing virtualization implementation approaches in general-purpose computers, which one is more suitable for a given embedded environment?*

**Research question 1:** There are some characteristics desirable in an ISA that ease the implementation of virtualization in it. Classic studies of Popek and Goldberg [PG74] introduce a classification of the instructions of an ISA (Instruction Set Architecture) into three different groups, in order to derive the virtualization requirements:

1. *Privileged instructions*: those that trap when used in user mode and do not trap if used in kernel mode[1];
2. *Control sensitive instructions*: those that attempt to change the configuration of resources in the system, and;
3. *Behavior sensitive instructions*: those whose behavior or result depends on the configuration of resources

Those works first stated that, in order to virtualize a given machine, sensitive (control and behavior) instructions must be a subset of privileged instructions.However, even if that is not the case of a given ISA, virtualization can still be achieved. For instance, the Intel's x86 family counts on hardware support to overcome the fact that not all of its sensitive instructions are a subset of privileged instructions.

In embedded systems, since there is more than one predominant architecture, each case should be analyzed individually. In this research we chose to study virtualization impacts on a MIPS-based platform, since this architecture is widely adopted, being present in video-games, e-readers, routers, DVD recorders, set-top boxes, etc. Moreover, this research is placed in the context of the Embedded Systems Group (in Portuguese, Grupo de Sistemas Embarcados) at PUCRS, and there were previous researches that employed the same architecture, thus reusing and expanding the group's knowledge in this matter. Still, MIPS is compliant with Popek and Goldberg virtualization requirements, easing its implementation.

[1] User and Kernel mode are operation modes provided by the architecture that allows safer execution of applications dividing what is privileged and what is not.

Unfortunately, the diversity of architectures used in embedded systems impacts on the fact that it is not possible to use a common virtualization solution to all of them. Since virtualization is also about improving the utilization of the underlying hardware, each platform should be carefully studied so a suitable proposal can be issued.

**Research question 2:** Mainly, the virtualization layer is responsible for providing several virtual machines and enabling them to execute their OSs individually, without one interfering in another. There are several ways of doing so, and the main techniques are:

- *Trap and emulate*, where each privileged instruction that is executed by the virtual machine is emulated by the virtualization layer, with large performance penalties;
- *Paravirtualization*, where each privileged instruction that would be executed by the virtual machine's OS is replaced by a proper and acknowledged system call, provided by the virtualization layer, so that no emulation is required. This approach requires access to the virtual machine OS source code and some extensive changes might be necessary;
- *Hardware support*, where the processor is aware that virtualization is a possibility and some level of support is added. In this case, the processor needs to offer support and virtual machine OSs can run without source code modification as long as they are ported to that architecture.

In many cases, paravirtualization offers the possibility of reducing virtualization overheads at the cost of needing to change the OS source code. However, we always believed this was not the best solution for embedded systems, since software complexity has already been increased due to the numerous features desired each more by users. So, using virtualization - which can improve design quality - at the cost of complex and numerous OS change is conflicting. Still, once the market started to see the virtualization potential, it was only a matter of time before leading market architectures such as ARM and PowerPC provided their own virtualization support. Finally, considering the timeframe in which this research has been developed, we propose some hardware modification so that a given MIPS platform counts on native virtualization support.

## 1.3 Contributions and original aspects

Historically, embedded systems have been inspired by many techniques previously deployed in general-purpose computers that presented specific challenges when thought for embedded environments. Virtualization fits in that category. However, its implicit processing overhead and expressive need for memory can be considered the main challenges when bringing it to embedded platforms.

By answering the proposed research questions, this research contributes by investigating suitable ways of using virtualization in multiprocessed embedded systems providing an architecture that offers hardware support. Still, a virtualization model was conceived to offer different mapping strategies, that can be extended in the future. Some initial investigation regarding real-time achievement in virtualized multiprocessed embedded environments was also performed. And last, but not

least, from that model two different implementations were made in MPSoCs, as a proof-of-concept. Our implementation modified MIPS-based processors and it is important to highlight that, by the time this dissertation was developed, there was no official MIPS support to virtualization, so that the modifications suggested in the architecture - aiming multiprocessed platform - are the main original aspects of the research.

## 1.4 Structure of this Dissertation

The remainder of this dissertation is organized as follows.

Chapter 2 - Background Concepts . This chapter presents background concepts especially focused on virtualization terms and known strategies that serve as basis to the entire research. This chapter is important considering this research context as it is placed in the embedded systems' field and virtualization concepts are not so widely known among researchers.

Chapter 3 - Literature Review. This chapter presents literature on embedded virtualization, showing that many researches prefer the use of paravirtualization. Also, since virtualization is also used by non-academic entities, commercial solutions are presented.

Chapter 4 - Virtualization Model Attemps. This chapter presents three different attempts to virtualization model in MPSoCs. First attempt considers using paravirtualization techniques while the following attempts propose hardware modification to MIPS-based processors.

Chapter 5 - This chapter presents the research validation process adopted throughout the research, discusses the contributions of this research, and disclose the threats to the validity of the contributions. The chapter concludes this dissertation with suggestions for future work.

# 2. BACKGROUND CONCEPTS

## 2.1 Embedded Systems

This section presents basic concepts regarding current embedded systems. The reader who is familiar with those concepts may skip to the next section.

A very common definition for embedded systems regards its strictness to a given application. Opposed to general-purpose systems, embedded systems used to be relatively simple devices, with limited and fixed number of software tasks. Embedded systems are present in a very wide range of applications, from consumer electronics to industrial automation among many others [Noe05], as shown in Table 2.1.

Table 2.1 – Embedded applications' examples
Source: Adapted from [Noe05]

| **Field** | **Embedded device** |
|---|---|
| Consumer electronics | Digital and analog TV<br>Games toys<br>Home appliances<br>Internet appliances |
| Medical | Dialysis machines<br>Infusion pumps<br>Cardiac monitors<br>Prosthetics |
| Industrial Automation and Control | Smart sensors, motion controllers<br>Man/machine interface devices<br>Industrial switches |
| Networking and communications | Cell phones<br>Pagers<br>Video phones<br>ATM Machines |
| Automotive | Entertainment centers<br>Engine controls<br>Security<br>Antilock break control<br>Instrumentation |
| Aerospace and Defense | Flight management<br>Smart weaponry<br>Jet engine control |
| Commercial Office/Home Office Automation | Printers<br>Scanners<br>Monitors<br>Fax machines |

Such a wide range of applications and devices, and their increasing amount of features has severely impacted on their design throughout the years. Embedded systems used to be relatively simple devices with severe hardware constraints, like memory use, processing power and battery life. Mostly, their functionality was determined by hardware modules with few software usage. In this case, software layers consisted, basically, of device drivers, task scheduler and some control logic, resulting in software with low complexity. Besides, these systems used to be closed, which means that during their lifetime no change in the software was required [HNN05]. This scenario, however, has already changed. It is very common that general-purpose applications are desired to be executed on embedded systems. Also, applications written by developers that have little or no knowledge at all about embedded systems constraints can also be employed [ZM00].

Nevertheless, we notice a rupture in a classical model of embedded systems' design [Noe05], as presented in Figure 2.1. Here, software and application layers are considered to be *optional* while current trends present the complete opposite. Each more, designers tend to implement system tasks in software and applications layers, since it allows higher flexibility, easier debug and higher reuse rates.

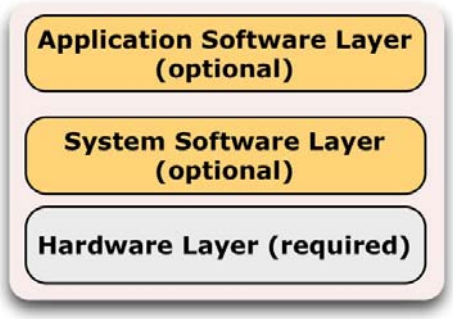


Figure 2.1 – Former Typical Embedded System Model
Source: Adapted from [Noe05]

Still, some of the traditional differences between general-purpose computers and embedded systems remain [LP05]. For example, even on high-end multimedia entertainment-driven embedded systems, some real-time constraints are present and the energy consumption is often a matter of concern. Since these devices are supposed to operate through several hours (up to days) without any battery recharge the processor frequency choice is impacted and usually lower frequency rates[1] are compatible with the energy consumption goals.

Another common restriction is memory usage. Modern embedded devices are designed to be cost effective and excessive memory opposes to that. Memories are a high energy consuming resource, and frequently represent a cost factor issue [HPHS04].

Embedded systems present a varied set o constraints that directly impact on their designs. While some ESs are more concerned in area and energy consumption reduction, such as cell phones,

[1] Valid when compared to higher processor frequencies used in no-battery dependent devices, such as desktop computers.

others need the most predictable and deterministic behavior in spite of pure performance levels, such as some avionic systems. Such constraints impact on the processor choice so that there are several predominant architectures in ESs, such as ARM, MIPS, PowerPC and even some Intel Atom versions. On the other hand, general-purpose systems are mainly implemented onto the x86 architecture.

Finally, embedded systems are becoming more and more part of everyday life, being increasingly used in mission and life-critical scenarios as well as entertainment gadgets [Wol03]. Such a wide range of applications enables new techniques to be deployed and virtualization can bring many advantages in this scenario.

## 2.2 Classical virtualization

This section presents classical virtualization concepts and the reader who is familiar with those concepts may skip to the next section.

Computer virtualization is an old technique invented by IBM in the early 1960's aiming to improve hardware utilization. Also, software compatibility was another requirement as software needed to be compiled to each specific hardware and could not be reused among other systems. They caused a revolution in terms of commercial success - the hardware could evolve and the software could remain the same.

In the early times mainframes were a common technology and IBM's virtualization was responsible for partitioning a single hardware into several virtual platforms. Each virtual platform was able to execute its own applications, including older versions of OSs. After a while, mainframes started being replaced by workstations and the need for this type of virtualization ended naturally throughout the years.

Later, in 1999 VMware released the VMware Workstation, a product tailored to the x86 architecture that allowed the user to run multiple operating systems on a single desktop computer. By that time, it already was well established that the x86 architecture was the leader in terms of standards and also, the architecture itself was mature enough to hold the overhead of a virtualized system.

Still, enterprise market took a lot of advantage in using virtualization techniques, since the same workload could be consolidated on fewer physical machines, which represented a huge cost reduction factor. Security is also improved, since a failure within a given virtual machine does not spread throughout the system [Hei08].

Nevertheless, although it causes a single hardware point of failure, virtualization still is considered a safer approach when comparing to non-virtualized systems. Usually, service interrupts are not caused by hardware failures. Instead, the main problem usually is the use of non-reliable software. In that case, failures can be often related to the software size, for example, operating systems, which are usually big, tend to have many flaws. Therefore, if the virtualization layer is small (when comparing to a classical OS) virtualization can be considered a safer approach.

### 2.2.1 Virtualization layer - the hypervisor

In virtualization, the leading role is played by a software known as the *hypervisor*, *Virtual Machine Monitor* - VMM, or *Virtualization Layer*. The hypervisor allows the creation and maintenance of multiple virtual machines, being responsible for their isolation by providing an abstract hardware layer. One of the most important roles played by the hypervisor is to arbitrate the access to the underlying hardware, so that *guest OSs* (OSs executing on each virtual machine) can share the physical resources. It is possible to say that the hypervisor manages virtual machines, composed by their OS and applications, in a similar way that a native OS manages its processes and threads. Figure 2.2 shows a generic idea of hypervisor, since it controls the hardware and provides the desired abstraction to the virtual machines.

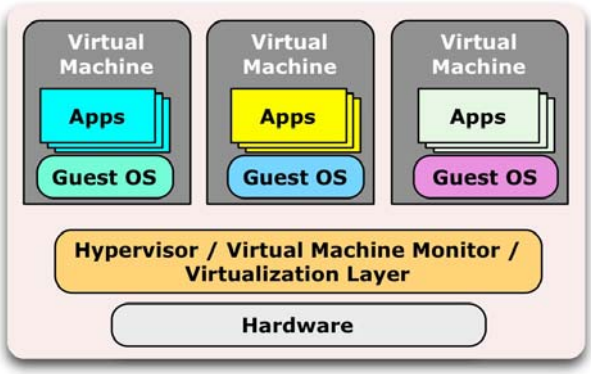


Figure 2.2 – Generic Hypervisor Model

Since this similarity exists, it is not surprising that details regarding the implementation of a hypervisor are also present in modern operating systems' implementation. Therefore, in order to understand how a hypervisor works, some OSs concepts are important, such as the *privilege scheme*. Most modern operating systems work within two modes:

1. *kernel or supervisor mode*, in which almost any CPU instructions are allowed to be executed, including *privileged* instructions - those that deal with interrupts, memory management, among others. Operating systems are executed in kernel mode, and;

2. *user mode*, that basically allows the execution of instructions needed to calculate and process data. User applications run in this mode, and can only access the hardware by asking the kernel through system calls.

Originally, the user/kernel mode scheme was adopted due to RAM's division into pages. Before executing a given privileged instruction, the CPU must check a right 2-bit code associated with that instruction. Privileged instructions require a *00* code whereas the least privilege is conceded with a *11* code[2].

[2] as a 4-bit code, four levels of privilege are allowed, with *00* being the highest and *11* the lowest

This scheme is often referred to as *protection rings* where rings are arranged in a hierarchy from the most to the least privileged. Therefore, in most operating systems, *Ring 0* (*00* 2-bit code) is the most privileged level and is able to interact directly with the physical hardware. Figure 2.3 depicts a common representation of the privilege rings scheme.

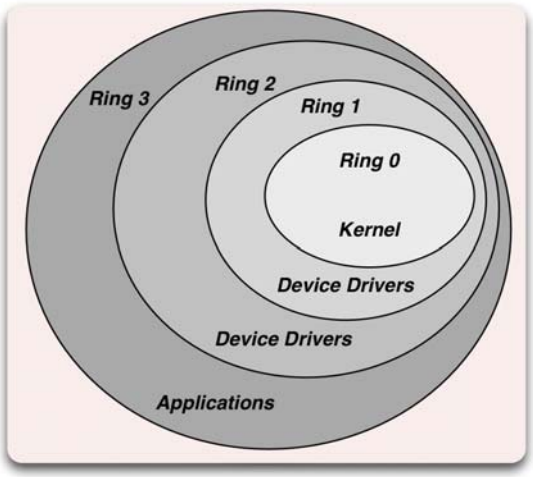


Figure 2.3 – Typical Privilege Rings of Modern CPUs

In a virtualized architecture, the hypervisor is the only software executed in the *Ring 0* privilege level. This has a severe consequence for guest OSs: they no longer run in Ring 0, instead, they run in Ring 1, with fewer privileges. This is known as ring de-privileging and allows the hypervisor to control guest OS accesses to resources, avoiding, for instance, one guest OS either to interfere in a neighbor's memory or to control hardware resources improperly. Ring de-privileging technique is exposed in Figure 2.4.

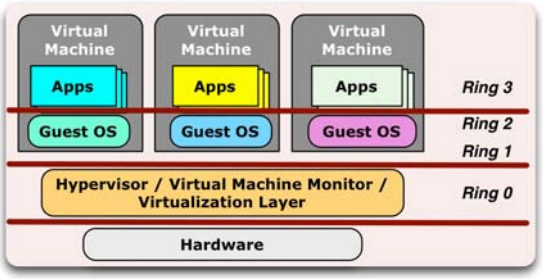


Figure 2.4 – Ring de-privileging caused by the hypervisor

*Hypervisor Typology.* According to [Ros04], there are two main approaches to implement the virtualization technique, by using either hypervisor of type 1 (depicted in Figure 2.5) or type 2 (depicted in Figure 2.6).

In hypervisor type 1, also known as *hardware-level virtualization*, the hypervisor itself can be considered as an operating system, since it is the only piece of software that works in kernel mode, like depicted in Figure 2.5. Its main task is to manage multiple copies of the real hardware just like an OS manages multitasking.

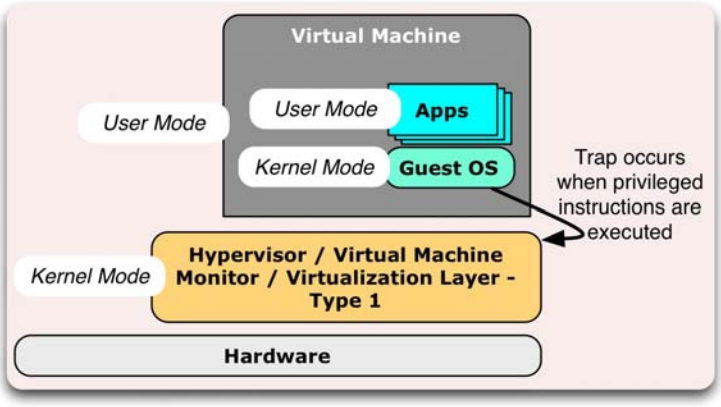


Figure 2.5 – Hypervisor Type 1

In type 2 hypervisors, also known as *operating system-level virtualization*, depicted in Figure 2.6, the hypervisor itself can be compared to another user application that simply "interprets" the guest machine's instructions.

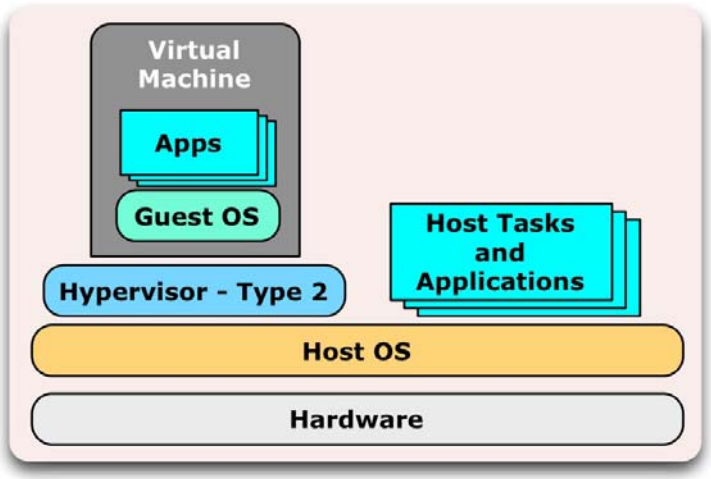


Figure 2.6 – Hypervisor Type 2

*Requirements.* The most common requirements and the goals to be achieved by a hypervisor are [Ros04]:

- compatibility, since the hypervisor needs to provide the execution of legacy software;
- performance, a measure of the virtualization overhead, where the ideal is to run the virtual machine at the same speed as the software would run on the real machine; and
- simplicity, since a hypervisor failure leads every virtual machine running on the computer to fail. Also, to provide secure isolation among virtual machines requires a free of bugs' hypervisor so that attackers could not use it against the system.

Virtualization Challenges

In its first years, virtualization provided by IBM implemented a hypervisor that allowed virtual machines to be executed, as expected. The main problem, is that it was an extremely robust approach in which every privileged instruction executed by a virtual machine caused a *trap*, since it was running in a less privileged ring. Virtualization must be as transparent as possible and the hypervisor is responsible for it.

One of the main problems faced by designers when virtualization started being noticed again, back in the 1990's, was that CPU architectures, like the popular x86, were not designed to be virtualizable. Certain instructions, when executed in unprivileged mode, are simply ignored instead of causing the CPU to trap. For instance, instructions like the POPF that disables and enables interrupts represent trouble for the hypervisor.

*Binary Translation.* When VMWare launched VMWare Workstation back in 1999, these aforementioned issues of the x86 architecture hadn't been solved yet. So, VMWare used a technique known as *binary translation*, used also in Intel Itanium processor. Nevertheless, VMWare version was a lot lighter than Itanium's, which had to translate from x86 Instruction Set Architecture to IA64 ISA while VMWare's binary translation was based on an x86 to x86 translator, which, in many cases, just had to copy the exact same instruction.

At run time, VMware translated the binary code coming from a guest OS and stored the adapted code in a memory structure known as Translator Cache (TC). It is important to highlight that user applications running on the top of guest OSs were not translated, since they continued to be executed in a non-privileged ring (as they would, if run natively). This scheme is depicted by Figure 2.7.

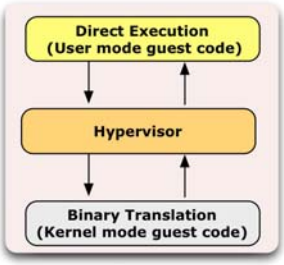


Figure 2.7 – Binary Translation for OS - Direct Execution for Applications

Analyzing this scheme, it is possible to see that the kernel code is the one that needs to be translated. This leads to the conclusion that the guest OS kernel is no longer executed - it is merely an input for the binary translator, as we can see in Figure 2.8.

The translation, in most cases, only copies the original code. Otherwise, when privileged instructions are desired or when hardware manipulation is required, the translator has to change the original code to either safer non-privileged instructions or to instructions with code reference to the virtual hardware. These manipulations can cause the translated code to be considerably larger than the original one.

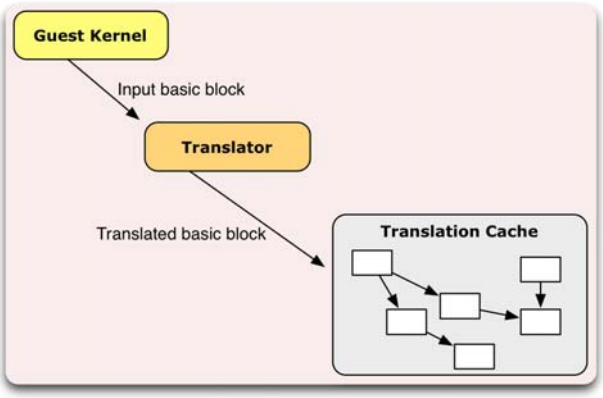


Figure 2.8 – Guest OS Kernel Code is an input to the Binary Translator

When compared to full virtualization, which causes traps at each privileged instruction, provided by IBM, the technique of replacing code with safer instructions presents higher performance. Although, in some cases, the overhead found for this virtualization approach is not negligible. This occurs in some special cases:

- system calls;
- I/O access, interrupts and DMA;
- memory management; and
- unusual code situations, such as self-modifying and indirect control flows, among others[3].

*System calls*. System calls are performed by user applications whenever they need to request the use of a privileged instruction. To perform system calls, x86 provides two simple instructions: SYSENTER (or SYSCALL) and SYSEXIT, that are responsible for beginning and finishing a system call, respectively.

Otherwise, additional work is needed to use virtualization due to the way virtualization (especially by binary translation) deals with privileged instructions: it replaces them with a slightly less privileged instruction that performs the same task. When using system calls, the main problem is that when SYSENTER is executed, it is sent to a privilege page in memory, where the OS is supposed to be. Instead, it finds the hypervisor which must emulate every system call, by translating its code and then coming back to the translated kernel code. It is quite clear that this is responsible for a big overhead, measured in VMWare [WCC+08], resulting that:

- a native system call takes, in average, 242 cycles, while
- the binary translated system call with a 32-bit guest OS takes, in average, 2308 cycles.

[3]This situation will not be considered for our analysis, since even in native OS, non-virtualized systems, it causes trouble for the operating system.

*Memory Management.* Another big challenge when implementing virtual machines is how the hypervisor interacts with the memory. Regular OSs maintain page tables that help during the translation of virtual memory pages into physical memory addresses. Since this is a much disseminated concept, modern x86 CPUs already provide support for this memory scheme directly in hardware. Thus, the translation itself is performed by the Memory Management Unit - MMU. For instance, in the x86 architecture, the current address is kept in the *CR3* register, also known as the hardware page table pointer, whereas the most used information are cached in the TLBs.

In virtualization, this has to be thought differently, since the guest OS cannot access the real page tables. Instead, it sees page tables executed on an emulated version of the MMU. This scheme gives the guest OS the illusion that it can translate the addresses itself, but truly, what happens is that the hypervisor is the only one to be dealing with it. Therefore, the real page tables are hidden from the guest OS and managed by the hypervisor, still being run on the real MMU in an approach named as *shadow page tables*.

Moreover, to maintain this approach, each time that the guest OS modifies its own page mapping, the virtual MMU module will trap the modification in order to adjust its shadow page table. Unfortunately, it causes a severe overhead: depending on the changes in the page table, the overhead takes from 3 to 400 times more cycles that a native execution [AA06]. Thus, memory intensive applications may suffer from a very big overhead due to memory management depending on how it is implemented.

*Paravirtualization.* When using virtualization at hardware level (type 1 hypervisor of 2.5) without hardware support, the hypervisor is in charge of translating instructions whenever the virtual machine tries to execute a privileged instruction (I/O request, memory write etc), which causes a trap into the hypervisor. This is known as pure virtualization and is often a very expensive way of dealing with virtual machines [Wal02].

Therefore, another option when dealing with hardware level virtualization is named as *paravirtualization* and it can be used to replace sensitive instructions of the original code by explicit hypervisor calls (known as *hypercalls*). In reality, the guest OS is acting like a normal user application running on a regular OS, with the difference that the guest OS is running on the hypervisor. When paravirtualization is adopted, the hypervisor must define an interface composed by system calls to be used by the guest OS. Still, it is possible to remove all sensitive instructions of the guest OS, forcing it to use only hypercalls. Besides working on hardware that is unsuitable for pure virtualization, it can also bring performance boost.

In that sense, paravirtualization is not so much different from binary translation. While binary translation changes privileged code instructions into "harmless" code at run time, paravirtualization does the same, but in the source code. Both approaches present pros and cons. For instance, changes made still in the source code are more flexible than those done on the fly, which must happen quickly. Paravirtualization eliminates many unnecessary traps to the hypervisor when compared to binary translation. On the other hand, it requires full access to the source code, which can be faced as a huge disadvantage in some cases.

In paravirtualization, the hypervisor provides hypercall interfaces for critical kernel operations such as memory management and interrupt handling. Although they exist for various operations, they will only be accessed when needed. Thus, the same way that VMware became known when adopting binary translation, Xen's paravirtualization solution was widely adopted.

One of the best features of the Xen implementation of virtualization is in the way I/O is handled: Xen proposes a concept of a privileged virtual machine responsible for dealing with those operations, named as *Domain 0*. This virtual machine links the simplified interfaces that appear to the VMs as the real native drivers, by requiring no emulation whatsoever. The concept proved out to be so good that even VMWare adopted it in newer versions of ESX server, by implementing paravirtualized network drivers. Figure 2.9 depicts Xen's scheme of paravirtualization.

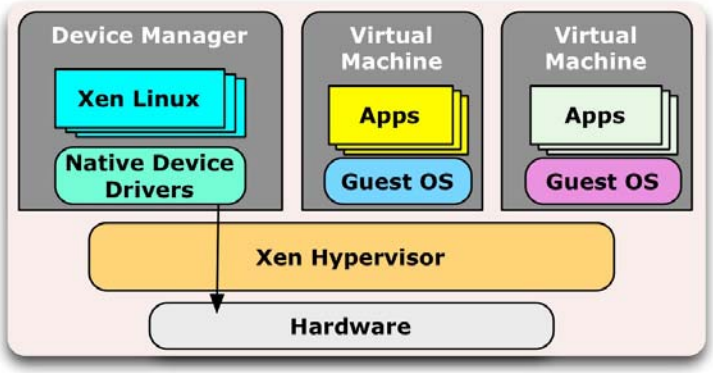


Figure 2.9 – Xen's paravirtualization approach

For comparison purposes, the difference between pure virtualization and paravirtualization is depicted in Figure 2.10. In part A of the figure, pure virtualization is shown. In this case, whenever the guest OS calls a sensitive instruction, a trap is caused to the hypervisor, which emulates the instruction behavior and returns the proper results. In part B, paravirtualization is showed. The guest OS has been modified in order to make hypercalls instead of containing sensitive instructions. In this case, the trap is similar to the one that occurs in non-virtualized systems, whenever a user application makes an OS system call.

*Hardware Accelerated Virtualization.* Since virtualization reached really noticeable use within the last years, challenges to achieve a better performance started to be a constant claim by the enterprise market, since software techniques were not good enough. Then, processor companies, like AMD and Intel, took advantage and increased their market share by releasing hardware support for virtualization.

Besides making existing execution modes virtualizable, basically, what Intel and AMD technologies did was to add a new execution mode to their processors, allowing the hypervisor to safely and transparently use direct execution when running virtual machines. This new execution mode increases the performance since the amount of traps needed to implement virtual machines is drastically reduced.

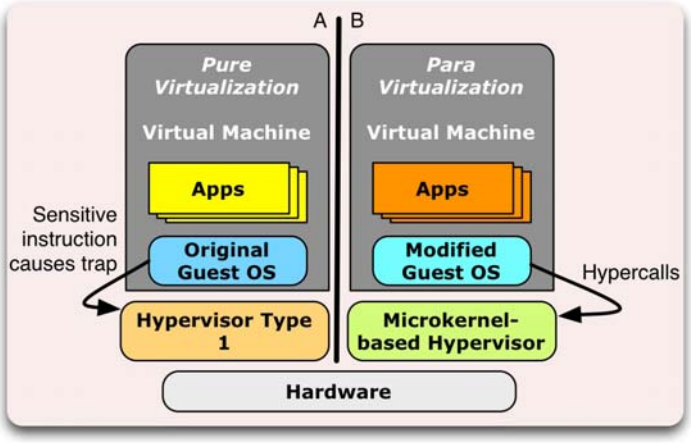


Figure 2.10 – Hypervisor control of pure virtualization (part A) and paravirtualization (part B)

However, the first generation of hardware virtualization support did not reduce all overheads to minimum levels, as it would be expected. This occurred, especially because this first release was not an improvement of neither binary translation nor paravirtualization. Instead, the first idea was to eliminate the original reasons that did not allow x86 based machines to be virtualized. Here, we can highlight the need to allow every privileged instruction to cause a trap, when executed in a different privilege level. Still, a new execution mode was created, forcing hypervisor to be executed in the -1 level of the privilege ring (root mode).

The biggest advantage is that the guest OS runs at its intended privilege level (ring 0) and the hypervisor runs at an even higher privileged ring. Therefore, guest system calls do not require the hypervisor to interfere: as long as they do not involve critical instructions, the guest OS can provide kernel services to the user applications as easily as if it was a native execution, as depicted in Figure 2.11.

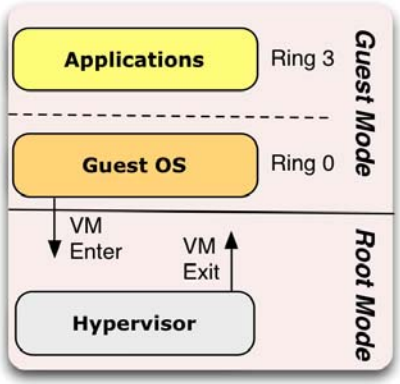


Figure 2.11 – Hardware support to virtualization

The problem faced by this implementation is that even though it is done in hardware, each transition from the virtual machine to the hypervisor (namely, VMexit and VMentry) requires

a fixed number of CPU cycles, which, depending on the internal CPU architecture can take from a few hundred cycles up to a few thousand cycles.

Thus, when Intel VT-x or AMD SVM (or AMD-V) has to handle with relatively complex operations such as system calls (which would be heavy anyway), the impact of the VMexit/VMentry switching can be considered light. On the other hand, if the actual operation to be emulated is simple, the overhead is significantly heavier.

*Hardware support improvement.* Managing the virtual memory of different guest OSs by translating their virtual pages into physical pages can be extremely CPU intensive. Each update of the guest OS page tables requires some update in the shadow page table. This is rather bad for the performance of software-based virtualization solutions but it definitively affects the performance of the earlier hardware virtualization solutions, since it causes a lot of VMexit and VMentry calls.

Then, with the second generation of hardware virtualization technologies, the AMD's nested paging and Intel's extended page tables (EPT), the problem was partially solved by providing a TLB that keeps track of both the guest OS and the hypervisor memory management. Figure 2.12 depicts how this approach works. The CPU with hardware support stores the virtual -to-physical memory mapping of the guest OS and also the physical memory to real physical memory transition of the guest OS. To do that, the TLB has a new specific tag for each virtual machine, called the Address Space IDentifier (ASID). It allows the TLB to know which TLB entry belongs to which virtual machine. Thus, the virtual machine does not flush the TLB: the entries of different virtual machines coexist in the TLB.

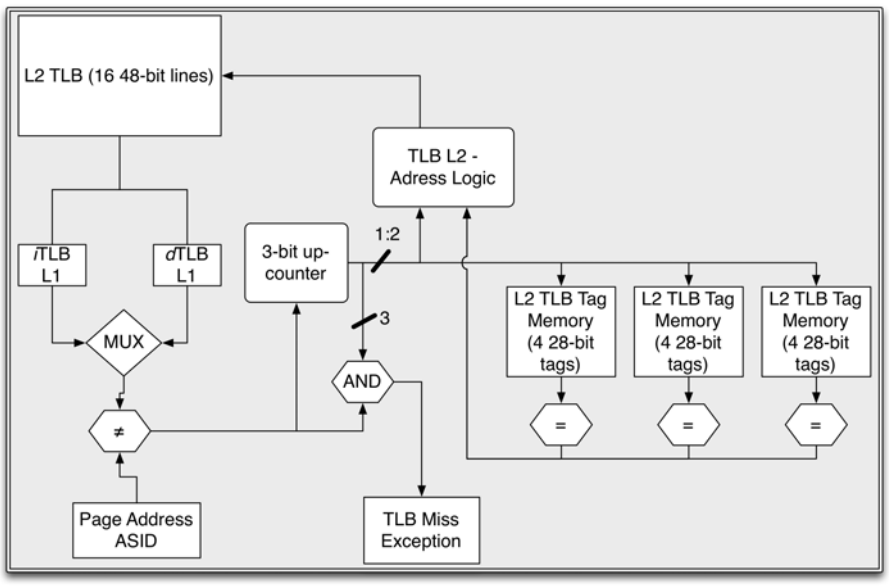


Figure 2.12 – TLB approach with Nested/Extended Page Tables

This approach makes the hypervisor implementation a lot simpler as it prevents the need to update the shadow page tables constantly. However, it makes the virtual to real physical address translation to be a lot more complex if the TLB does not have the right entry. In order to compensate it, larger TLBs are used since TLB misses become extremely costly.

# 3. LITERATURE REVIEW

Virtualization is a consolidated technique which dates back more than 30 years, being primarily proposed by IBM. Throughout the years, two main approaches have been adopted to implement it successfully. In *full virtualization* an almost complete simulation of the actual hardware is performed, enabling Guest OSes to run unmodified. In *paravirtualization* Guest OSes need to be modified to run avoiding the excessive amount of traps that occur when a Guest OS tries to execute a privileged instruction (when executing outside of its intended privilege ring).

However, full virtualization per se usually suffers from a large overhead from the emulation of privileged instructions while paravirtualization demands Guest OSes to be modified, which can increase both engineering cost and system's time to market. Hence, a viable solution is the use of hardware support for virtualization. For example, general-purpose processor vendors such as Intel and AMD have released, respectively, VT (Virtualization Technology) and SVM (Secure Virtual Machine) virtualization support for the x86 architecture. Next section presents the main architectures with hardware support to virtualization in the embedded systems' context.

## 3.1 Hardware support in embedded architectures

The embedded market has seen hardware-assisted virtualization being introduced in the last years. Intel itself has introduced the Intel VT technology also for its embedded processors [Int11]. In this case, many virtualization tasks are performed in hardware, such as memory address translation, which reduces the overhead and footprint of virtualization software improving its performance. For instance, switching between two OSes is significantly faster when memory address translation is performed in hardware compared to software. Still, it has unified the Intel VT-x along with the Intel AMT (Active Management Technology) technology that provides remote management and maintenance capabilities, and Intel TXT (Trusted Execution Technology) that protects embedded devices and virtual environments against rootkit and other system level attacks, to provide the Intel vPro support aiming to reduce the total cost of ownership (TCO) of embedded systems.

IntelVT-x provides several hypervisor assistance capabilities, including a true hardware hypervisor mode, enabling unmodified guest OSs to execute with reduced privileges. For example, Intel VT-x is able to prevent a guest OS from referencing physical memory beyond what has been allocated to the guest's virtual machine. In addition, VT-x enables selective exception injection, so that hypervisor-defined classes of exceptions can be handled directly by the guest OS without incurring the overhead of hypervisor software interposing. While VT technology became popular in the server-class Intel chipsets, the same Intel VT-x technology is available in Intel Atom embedded mobile processors [Moy13].

ARM has also introduced a virtualization support with an extension for its ARM v7-A architecture [Arc13], ARM VE. Basically, it consists of introducing a new execution mode for the

hypervisor with higher priority than the supervisor mode. This enables the hypervisor to execute at a higher privilege level than the Guest OSes, and the Guest OSes to execute with their traditional operating system privileges, removing the need to employ paravirtualization techniques. Still, improvements of mechanisms to aid interrupt handling are available, with native distinction of interrupt destined to secure monitor, hypervisors, currently active Guest OSes or non-currently-active Guest OSes. This dramatically reduces the complexity of handling interrupts using software emulation techniques and shadow structures inside the hypervisor. Finally, the provision of a System MMU that aids memory management and supports multiple translation contexts and two levels of address translation and hardware acceleration and abstraction.

ARM still offers the ARM TrustZone technology. It enables a specialized, hardware-based form of system virtualization. Basically, it provides two different "zones": *normal* and *trusted*. Using this technology, a multimedia operating system (like the ones seen by the user in some smartphones) runs in the normal zone, while security-critial software runs in the secure zone. While secure zone supervisor mode software is able to access the normal zone's memory, the reverse is not possible, as shown in Figure 3.1. Thus, the normal zone acts as a virtual machine under control of a hypervisor running in the trust zone. However, unlike other hardware virtualization technologies, such as Intel VT-x, the normal-zone guest OS incurs no execution overhead relative to running without TrustZone. Thus, TrustZone removes the performance barrier to the adoption of system virtualization in resource-constrained embedded systems. TrustZone is orthogonal to ARM VE: the hypervisor mode introduced in VE enabled cores only applies to the normal state of the processor, leaving the secure state to its two-level supervisor/user mode hierarchy. Thus the hypervisor executes de-privileged relative to the trust zone security kernel. ARM VE also supports the concept of a single entity that can control both the trust- and the normal- zone hypervisor mode [Moy13].

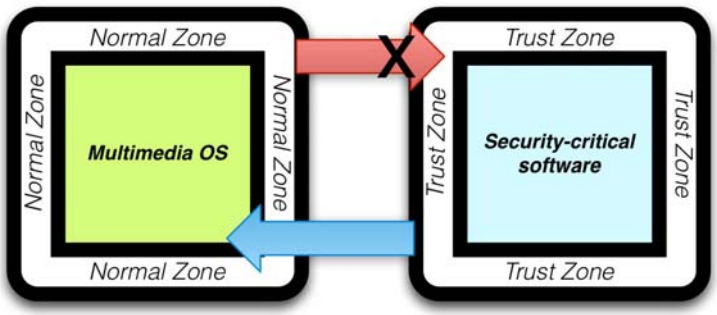


Figure 3.1 – ARM TrustZone

Power.org (Power Architecture technology), announced virtualization support in the release of Power Instruction Set Architecture Version 2.06 [Pow12]. The document provides support for virtualization and hypervisors including a new guest mode and MMU extensions that enable the efficient implementation of hypervisors on the embedded Power Architecture platform. It allows a more efficient implementation of virtualization, partitioning of embedded systems, isolation of

applications, and resource sharing. Freescale was the first to release platforms using processors that contain this extension.

Finally, in April of 2013 MIPS announced directions for future generations of processors to count with hardware support for virtualization. By the time this dissertation was written there were no commercial processors in the market using the technology. The main points concern offering another execution mode besides duplicating several structures to decrease virtualization overheads. It is important to notice that, by the time this research was developed, there was no MIPS hardware support to virtualization and that any similarity is just a mere coincidence.

## 3.2 Existing Hypervisors

Virtualization is a technique that offers many advantages and it is often found as a commercial solution. Therefore, this section brings some existing commercial and academic hypervisors.

**EmbeddedXEN Project.** EmbeddedXEN is an academic project of the XEN.org research group where the main target are embedded real-time applications. In this case, the hypervisor is executed in ARM cores, since this is one of the most used embedded processors. The EmbeddedXEN project provides to ARM developers a single multi-kernel binary image which includes XEN, Linux, miniOS and a XenomaiRT extension adapted to run onto embedded systems. Virtualization and isolation mechanisms are fully relied on the XEN hypervisor for general purpose computers. The main goal of this project is to provide viability and performance evaluation of the embedded virtualization. It is an open-source project and it can be used in any device, although the Server Xen version is not open-source which can restrict its use.

In terms of architecture, EmbeddedXEN creates a page table for each guest OS when the guest domain is created, in order to support virtual memory systems. Though some RTOSs do not use any virtual memory technique, using the physical memory itself, the hypervisor can map the physical memory allocated by a guest RTOS into the same virtual memory, statically. At run time, the guest OS is executed as if it was using a physical memory, being isolated one from another by the page table provided by the hypervisor. This is a very simple approach which enables to use unmodified OSs with the virtualization solution although it may cause paging failures [Pa09].

To summarize the main characteristics of the EmbeddedXEN approach, we can highlight: (i) in cases of exception handling, system calls are interpreted by the hypervisor, using software interrupts; (ii) memory mapping is done by mapping both hypervisor and visitors' domain (kernel and user process) at the same memory space, and; (iii) the access control domain is used to prevent users from accessing a given process in kernel space memory from the user space memory [org12]. Figure 3.2 depicts the internal structure of the hypervisor, based on the general purpose's version of Xen.

**OKL4.** Implemented by OK Labs (Open Kernel Labs), this hypervisor (named microvisor by the developers), is an Open Kernel system which offers support to the virtualization technique.

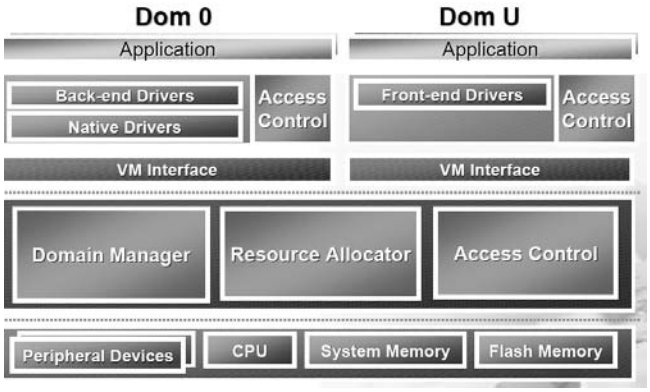


Figure 3.2 – EmbeddedXen Hypervisor Approach
Source: [org12]

Basically, it is an L4 family microkernel commercially distributed hypervisor with low overhead rates, [Hei09]. It has a high performance IPC (Inter-process communication) message exchange mechanism, which helps the low overhead virtualization. A system call, that causes a trap triggered by any virtual machine, calls the microkernel exception manager, converting this event into an IPC message to the guest OS. The client deals with this process as a normal system call and the answer is returned through another IPC message [Hei09]. Still, the OKL4 uses IPC to manage guest OS interrupt calls and to allow the communications of device drivers and the synchronization of system components including the virtual machine's. OKL4 uses an efficient resource sharing management, where memory regions can be shared by different address spaces mapping. In order to prevent unprivileged access, these memory regions respect system's permissions [Hei09]. Figure 3.3 depicts an example of this approach.

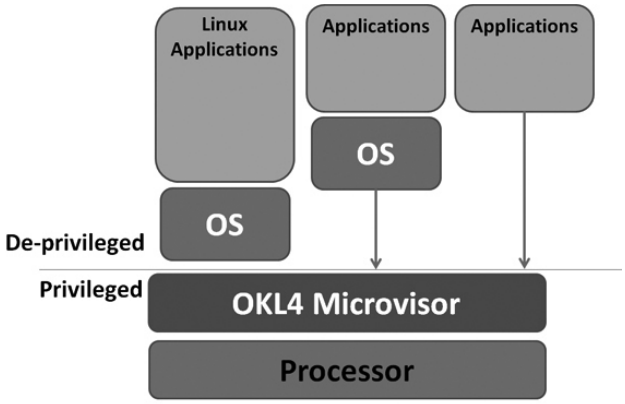


Figure 3.3 – OKL4 Hypervisor Approach
Source: Adapted from [Hei09]

**Wind River Hypervisor.** An integral part of Wind River's Multicore Software Solution, Wind River Hypervisor focuses on high performance, small footprint, determinism, low latency and high reliability [Riv13]. In terms of processor, it supports single and multicore processors based on Intel and PowerPC architectures and it integrates with VxWorks and Wind River Linux. It also

enables devices to be assigned to virtual boards as it provides device and memory protection between virtual boards.

To ease configuration, it uses XML-based system and changes made in this configuration scheme do not require rebuilding the guest operating systems or the user applications. As a debug facility, multiple virtual boards can be checked by a physical Ethernet connection. Regarding core scheduling, it provides a priority-based scheduler and custom schedulers can be used too.

Communication is achieved by using MIPC (multicore/multi-OS interprocess communication), a message-passing protocol designed for communication between cores and virtual boards. It allows virtual board management, by enabling functions as start, stop and reload/restart of the guest operating systems. Figure 3.4 depicts an example of this approach.

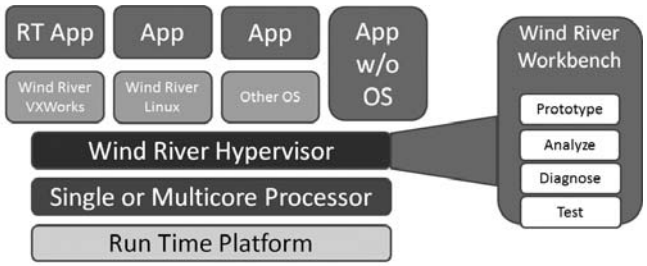


Figure 3.4 – Windriver Hypervisor Approach
Source: Adapted from [Riv13]

**VirtualLogix VLX.** Product of VirtualLogix, VLX is a virtualization software that decouples hardware management (intended for ARM and Intel architectures) and application environments (Android, Linux, proprietary, Symbian, Windows), enabling separation of design and functionality concerns. This allows OS/device independence and fault tolerance with minimal overhead, as well as improved performance for multimedia and gaming, besides enhanced device security through isolation [VLX13].

The main characteristics and advantages of VLX are:

- implemented in less than 50,000 lines of code;
- isolation technology makes it easy to implement new policies without changing the software, thus enabling smartphone OS functionality on a low-cost hardware platform;
- allows trusted and proven software to be supported with minimal work;
- reduces engineering and development effort by 35% to 50%, and;
- includes advanced system level policies for scheduling, memory, power and security management.

Figure 3.5 depicts an example of this approach.

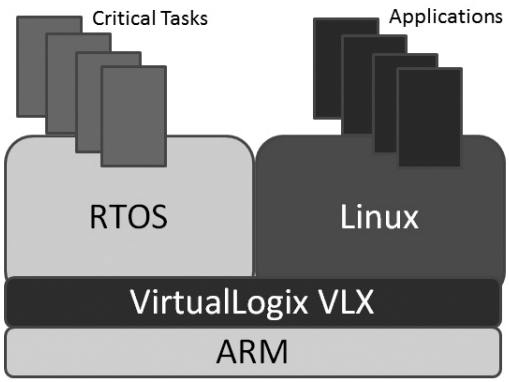


Figure 3.5 – VLX Hypervisor Approach
Source: Adapted from [VLX13]

**Trango.** This approach provides an efficient thin layer of code that gives system designers greater flexibility to extend the functionality of an existing system or to use only one CPU to handle multiple OSs, for example [Tra13].

With the hypervisor product provided by TRANGO, only a single CPU is needed to keep the OS and multiple environments separated. This means that a system designer working with an existing system can create trusted areas where secure processes like key management or secure boot can run without adding another CPU. Still, in systems where there is more than one CPU, the hypervisor can extend functionality without major hardware changes.

This approach is applicable to a wide range of systems including DVD players, printers, cable and DSL modems, routers, medical equipment, electronic payment systems, video and data processing, set-top boxes and digital televisions.

For mobile phone applications, this allows multimedia, real-time and trusted applications to be easily integrated, reducing production and development costs, and enhancing the security and integrity of users personal data. In networking equipment, the hypervisor allows the secure integration of Linux and market standards into existing embedded systems, supporting symmetric multi-processing operating systems, high-availability and OS redundancy policies. Figure 3.6 shows an example of this approach.

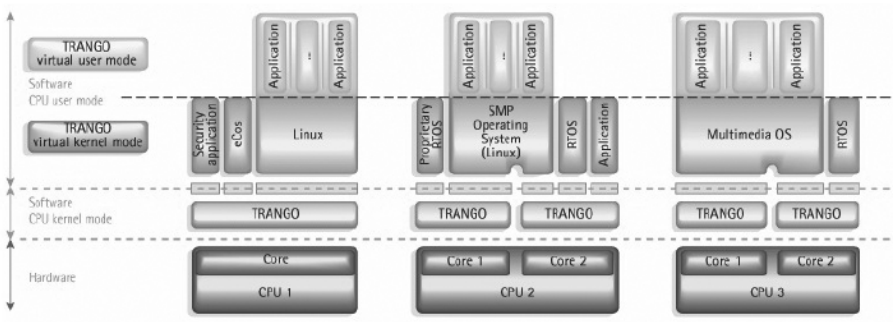


Figure 3.6 – Trango Hypervisor Approach
Source: Adapted from [Tra13]

**XtratuM.** XtratuM is an open source hypervisor specially designed for embedded real-time systems available for x86, PowerPC, MIPS and recently for LEON2 (SPARC v8) processors. It provides a framework to run several operating systems in a robust partitioned environment. XtratuM can be used to build a MILS (Multiple Independent Levels of Security) architecture [CRM10].

It follows several design criteria, which are specific for critical real-time embedded systems, such as:

- strong temporal isolation, implemented as a fixed cyclic scheduler;
- strong spatial isolation, that is, all partitions are executed in processor user mode and do not share memory;
- basic resource virtualization, such as clock and timers, interrupts, memory, CPU and special devices;
- real-time scheduling policy for partition scheduling;
- efficient context switch for partitions;
- deterministic hypercalls (hypervisor system calls);
- health monitoring support;
- robust and efficient inter-partition communication mechanisms (sampling and queuing ports);
- low overhead;
- small size, and;
- system definition (partitions and allocated resources) defined via an XML configuration file.

In the case of embedded systems, particularly avionics systems, the ARINC 653 standard defines a partitioning scheme. Although this standard was not designed to describe how a hypervisor must operate, some parts of the model are quite close to the functionality provided by a hypervisor. Thus the XtratuM API and internal operations resemble the ARINC 653 standard. Figure 3.7 depicts an example of this approach.

**SPUMONE**. A virtualization layer that works with paravirtualized systems but claims to have small engineering cost in terms of needed modifications in the guest OS[KYK+08]. The solution provides the VCPU and the idea that multiple virtual processors can be associated with a single application domain. However, SPUMONE provides a virtualization layer that executes in privileged space as does the guest OS.

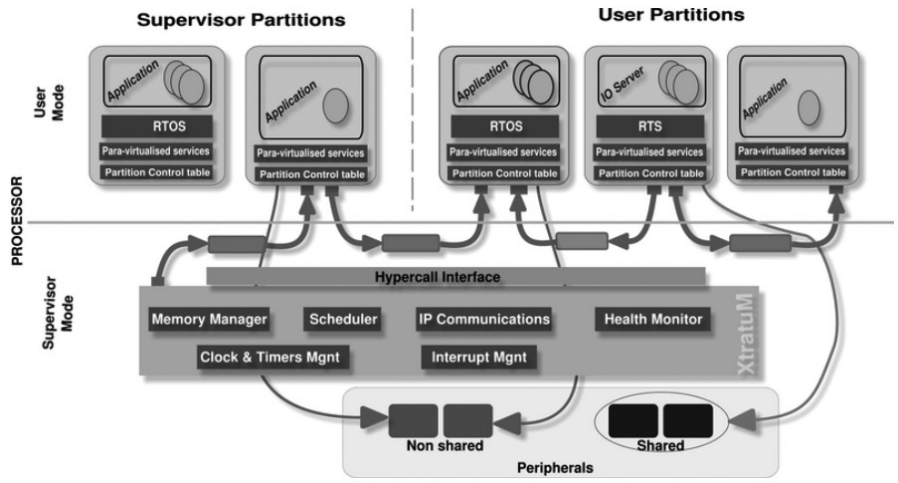


Figure 3.7 – Xtratum Hypervisor Approach
Source: Adapted from [CRM10]

## 3.3 Academic research

Virtualization interest has grown in the last few years and several research have been published. This section briefly describes some of them, in chronological order.

Mahmoud [EHMAZAR05] proposes the use of virtual machines to provide more chances of achieving a given schedule. Basically, the idea is to divide a single processor into several virtual processors that can be managed individually to improve schedulability. It creates partitions so tasks WCET (Worst Case Execution Time) can be analyzed independently from each other.

Ito [IO07] brings a lightweight virtualization technique focused on embedded and ubiquitous devices. It is a paravirtualized approach where the authors claim to need few modifications for the guest OS. Results were taken comparing a Linux port to this architecture against a XEN-ported Linux.

Hwang [HbSH+08] focus the research on ARM-based secure mobile phones. It is a paravirtualized implementation of Xen architecture that claims to have a moderate overhead comparing to non-virtualized platforms. The main challenge of the research consists in adapting the Xen platform to an ARM-based processor.

Brakensiek [BDB+08] shows many aspects of virtualization that are suitable for making mobile systems safer. The paper highlights some points such as sandboxes, which improve security of different OSs that coexist in the same platform. Still, it argues about some of the options regarding virtualization implementation and paravirtualization was used in the case-studies.

Heiser [Hei08] discusses the usage of virtualization in embedded systems, also highlighting the security as one of the main reasons for that. Some limitations are shown and details of the OKL4 platform are given.

Kanda [KYK+08] introduces the SPUMONE platform. In this particular paper, they use a monoprocessed hardware platform where the hypervisor offers an API for the guest OSs in a

paravirtualized approach. The authors claim that the amount of modification in the guest OSs are not prohibitive. Results highlight the overhead of the approach and comparisons are made with native executions. Kanda [KYK+08] also proposes the use of virtualization as a management layer in MPSoCs systems. To provide better performance, each guest OS is allocated in one core of the multiprocessed platform.

Inoue [ISE08] propose the use of virtualization to provide dedicated domains for preinstalled applications and virtualized domains for downloaded native applications. With it, security-oriented next-generation mobile terminals can provide any number of domains for native applications. This asymmetric configuration uses processor separation logic between domains and enables low overhead virtualization, shown in the results.

Yoo [YLH+08] proposes a virtual machine monitor to address three specific issues in mobile phones: real-time support, resource limitation, and power efficiency. Basically, the applications running on the top of this platform are categorized and each category runs in different virtual machines. The approach is based in paravirtualization.

Kleidermacher [KW08] presents a virtualization platform that aims avionic systems. This approach is based in the Integrated Modular Avionics (IMA) norm, that aims to manage and protect high value and complex information and applications across the aircraft. A paravirtualization proposal is discussed and the authors claim to assure the safety and security of critical applications while incorporating highly functional general purpose virtualized environments.

Cereia [CB09] uses virtualization as a way to separate multiple OSs, with different purposes, in the same system. It is a paravirtualized approach and aims to keep real-time execution as accurate as possible. The ARM architecture is used and comparisons are made between a TrustZone-based and a minimalist-based implementation.

Su [SCH+09] proposes the use of a Type-2 hypervisor basing the implementation on the use of KVM and QEMU. It works with Intel Atom processors and intends to offer a close to native-exeuction performance without using paravirtualization. The authors also introduce a mechanism named DirectShadow, which uses the guest page table directly for Hardware-assisted Vir- tual Machine (HVM) guest decreasing significantly the VMExit overhead.

Park [Pa09] presents an RTOS used in a virtualized platform based in a paravirtualized implementation of Xen in an ARM architecture. Some issues regarding the port of an RTOS are discussed such as memory support and timer management.

Heiser [Hei09] uses the aforementioned OKL4 as a possibility for enabling virtualization in Consumer Electronics. This paper focused on some aspects that are said to be fundamental in enterprise embedded hypervisors: low-overhead communication, real-time capability, small memory footprint, small trusted computing base, and fine-grained control over security. Then, the authors claim that their solution addresses these aspects satisfactorily.

Moore [MBC+09] presents the use of DBT (Dynamic Binary Translation) on ARM-based architectures. DBT can suffer from severe overhead penalties, so the authors propose some cache

and TLB changes. Overhead was reduced when comparing to a DBT-based platform without these modifications.

Armand [AG09] presents some differences of using virtualization in embedded systems highlighting the differences between hypervisor- and micro-kernel-based approaches. The analysis is centered in virutalization on consumer electronics systems and mobile technologies and aspects such as tasks and threads, scheduling, memory, communications, device drivers, and virtual machine management were analyzed.

Wang [WZS+10] proposes a scheduling mechanism based in messages in a Xen-based virtualized real-time platform. To achieve this, authors have improved the existing real-time scheduling algorithm in Xen. Each real-time guest domain sends information about individual real-time tasks so the Xen scheduler can arrange the tasks accordingly.

Yoo [YKY10] discusses a method for keeping real-time constraints in a virtualized environment in embedded systems. Their main contribution is that to provide a new abstract periodic interface to a real-time virtual machine so that the virtual machine can meet the physical execution condition. They use a compositional framework to deal with the two-level scheduling problem.

Steinberg [SK10] presents a microhypervisor-based virtualization architecture - NOVA - that consists of the microhypervisor and a user-level environment that contains the root partition manager, virtual-machine monitors, device drivers, and special-purpose applications that have been written for or ported to the hypercall interface. It is a parvirtualized approach.

Ryu [RKKE10] presents a hypervisor for mobile environments. It works on x86 and ARM architectures. The authors use the technique of kernel-user address space separation to provide a safe environment for mobile devices. This has the strong ability of memory protection, and provides efficient device management.

Rivas [RAN10] discusses about a cross-layer soft real-time architecture for virtualization where a new scheduling approach is proposed. The architecture of the proposed platform is composed by three main entities: the Distributed Kernel Coordinator (DKC), the Real-Time Scheduler and the Janus Executive. It is a Xen-based paravirtualized implementation.

Lee [LbSC10] focuses on improving security in 3G/4G mobile devices. It is based on Xen and uses paravirtualization. Basically, several improvements were made in terms of I/O treatment so attacks to the VMM could be easily avoided.

Forneaus [For10] explores the opportunities and challenges in using virtualization in embedded devices. The author presents several requirements that must be achieved to successfully combine hypervisors and multicore processors. It highlights the need for reasonable code size, performance and throughput, determinism, hardware support for virtualization among others.

Srinivasan [SPKG10] describes an approach to virtualizing SoC platform and explores the opportunities for shared use of virtualized SoC devices by multiple concurrently executing services. The implementation is paravirtualized using Xen hypervisor and x86 architecture.

Asberg [AFNK11] proposes a framework for scheduling (soft real-time) applications residing in separate operating systems (virtual machines) using hierarchical fixed-priority preemptive scheduling, without the requirement of kernel modifications. The authors use hierarchical scheduling to improve the performance of soft real-time virtual machines.

Nakajima [NKS+11] proposes an extension of the SPUMONE platform, with multicore advances. Still, paravirtualization is used and no architectural modification were suggested.

Xi [XWLG11] shows the RT-Xen platform that provides effective real-time scheduling to guest Linux operating systems at a 1ms quantum, while incurring only moderate overhead for all the fixed-priority server algorithms. Comparisons with other algorithms such as Sporadic Server and Deferrable Server were made to achieve the results.

Lee [LHC11] propose an optimization technique, called inline emulation, to reduce the cost spending on paravirtualizing operating systems. Inline emulation can also be used in various virtualization environments to increase the performance of emulating privileged instructions. The implementation is based in the ARM architecture.

Yang [YKP+11] proposes the implementation of a compositional scheduling framework on virtualization. They use a two-level scheduling framework in a L4/Fiasco microkernel that works as a hypervisor. They aim to provide better support to real-time in virtualized embedded systems.

Ost [OVI+12] explores the use of virtualization to enable mechanisms like task migration and dynamic mapping in heterogeneous MPSoCs, targeting the design of systems capable of adapt their behavior to time-changing workloads. They propose the use of Low Level Virtual Machine (LLVM) to postcompile the tasks at runtime depending on their target processor. A novel dynamic mapping heuristic was proposed, aiming to exploit the advantages of specialized processors while taking into account the overheads imposed by virtualization.

Li [LKM+12] extends the SPUMONE to provide the virtualization layer to multicore processors. The authors aim to use rich functional embedded systems.

### 3.3.1 Summary

With the growing hardware support for virtualization in embedded architectures, the use of full-virtualization must be better investigated. Currently, related works focus mainly in paravirtualization approaches (changes in the guest OS are required). However, full-virtualization is desired in embedded devices as long as it has proper hardware support [BGDB10]. Thus, the main contribution of our work consists in investigating the behavior of a hardware-assisted MIPS-based MPSoC platform, while rendering a secure and transparent environment for the guest OS and embedded applications.

Analyzing the aformentioned research, it is possible to see that the majority focus attention to paravirtualized solutions. These approaches depend on guest OS modifications to be successful. We decided not to use such approach. Also, many research use ARM or x86 as architecture and we

chose to study and improve a MIPS-based processor to provide some support for virtualization. Our solution is focused in MPSoCs, since that is a well-established trend in embedded systems. Finally, some real-time support was added, since that is also a great concern. Table 3.1 summarizes some of the aforementioned approaches.

Table 3.1 – Comparison among different virtualization approaches

| **Hypervisor** | **Processors** | **Virtualization Technique** | **RT Support** | **Multi-processor support** | **License** |
|---|---|---|---|---|---|
| OKL4 | ARM, MIPS, Intel | Para-virtualization | Yes | Yes | Proprietary |
| Embedded Xen | ARM | Para-virtualization | Not mentioned | No | GPLv2 |
| Xen-ARM | ARM | Mixed | To be available | To be available | GPLv2 |
| WindRiver | ARM, PowerPC, Intel | Para-virtualization | Yes | Yes | Proprietary |
| SPUMONE | SH-4A | Para-virtualization | Soft-real-time | Yes | Proprietary |
| Mobi-VMM | ARM | Para-virtualization | Yes | No | Proprietary |
| Our proposal | MIPS | full virtualization | Yes, with bare-metal applications | Yes | Proprietary |

# 4. VIRTUALIZATION IN MPSOCS - PROPOSALS AND IMPLEMENTATIONS

While the research to this doctoral dissertation was being performed, there were three different approaches to achieve virtualization in MIPS-based systems. Each one has its own advantages and disadvantages and are described in the next sections.

## 4.1 First Attempt - a HellfireOS improvement and Paravirtualized approach

This was the initial attempt we made to provide virtualization in embedded systems. This research is inserted in the Embedded Systems Group at PUCRS, where the Hellfire Project already existed. Since a lot of the literature pointed out that a hypervisor could be based in a microkernel, the first attempt was to adapt the existing HellfireOS into a hypervisor.

### 4.1.1 Hellfire Framework

Hellfire Framework (HellfireFW) [AFM+10] allows a complete deployment and test of parallel critical and non-critical embedded applications in mono- and multi-processed systems, defining the HW/SW architecture to be employed by the designer. The HellfireFW follows a design flow where several steps can be performed aiming to develop the HW/SW solution for a given application. This design flow is presented in Figure 4.1.

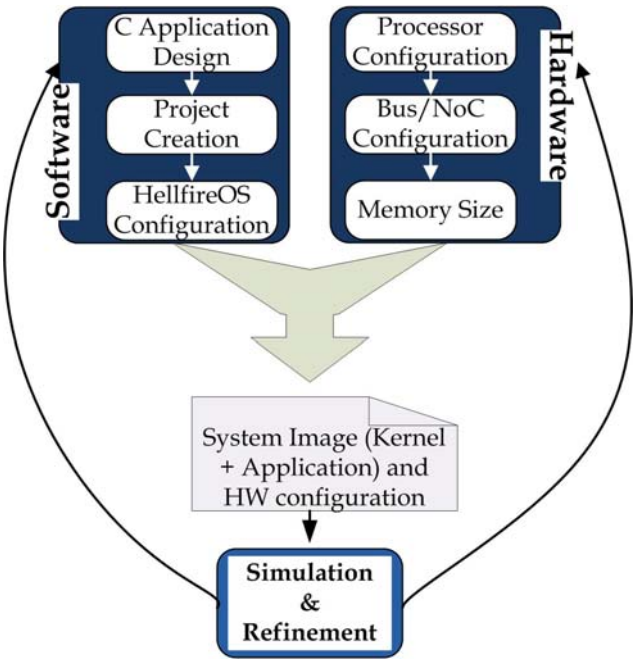


Figure 4.1 – Hellfire Framework Design Flow

In terms of application design, the entry point is in C language, where an application is manually divided into a set of tasks. Each task $\tau_i$ is defined as a n-uple $(id_i, r_i, WCET_i, D_i, P_i)$ and the parameters stand for identification, release time, worst case execution time, deadline and period of task $\tau_i$, respectively. They can communicate either through shared memory (in the same processor) or message passing (in different processors).

After designing the application, the HellfireFW project must be created. This is the step where the initial HW/SW platform configuration is defined. The C application is executed on the top of the HellfireOS stack. HellfireOS [AFM+10] is a micro-kernel based Real-time Operating System - RTOS, highly configurable and easily portable. To ease the OS port to other architectures, HellfireOS uses a modular structure as depicted in Figure 4.2.

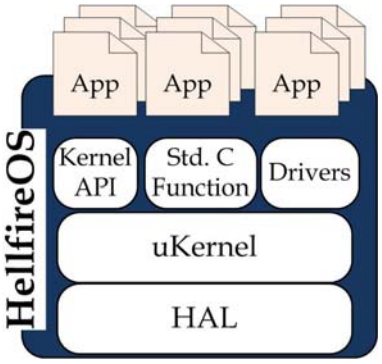


Figure 4.2 – HellfireOS Structure Stack

All hardware specific functions and definitions are implemented on the Hardware Abstraction Layer (HAL), which is unique for a specific hardware platform solution, simplifying the port of the kernel onto different platforms. The micro-kernel itself is implemented on top of this layer. Features like standard C functions and the kernel Application Programming Interface (API) are implemented on top of the micro-kernel. Communication and migration drivers, memory management and mutual exclusion facilities are implemented on top of the kernel API and the user application is the highest level in the HellfireOS development stack.

After following these steps, an MPSoC platform configuration is expected with a given number of processors, a personalized instance of HellfireOS on each processor, and a static task mapping. The user must then trigger the simulation of the system, which runs for a given time window and then generates several graphical results for the designer to analyze. If the results are satisfactory, the SW part of the platform can be easily ported to a prototype, such as an FPGA. Otherwise, the designer can change the HW/SW settings and rerun the simulation as refinements are needed.

### 4.1.2 Virtual-Hellfire Hypervisor (VHH), a HellfireOS-based Hypervisor

In this section we describe the Virtual-Hellfire Hypervisor architecture, based in the HellfireOS structure. The main advantages of VHH are:

- temporal and spatial isolation among domains (each domain contains its own OS);
- resource virtualization: clock, timers, interrupts, memory;
- efficient context switch for domains;
- real-time scheduling policy for domain scheduling;
- deterministic hypervisor system calls (hypercalls).

VHH considers a **domain** as an execution environment where a guest OS can be executed and it offers the virtualized services of the real hardware to it. For embedded systems where no hardware support is offered, paravirtualization tends to present the best performance results. Therefore, in VHH, domains need to be modified before being executed on top of it. As a result, they do not manage hardware interrupts directly. Instead, the guest OS must be modified to allow the use of virtualized operations provided by the VHH (hypercalls).

Figure 4.3 depicts the Virtual-Hellfire Hypervisor structure. In this figure, the hardware continues to provide the basic services as timer and interrupt but they are managed by the hypervisor, which provides hypercalls for the different domains, allowing them to perform privileged instructions.

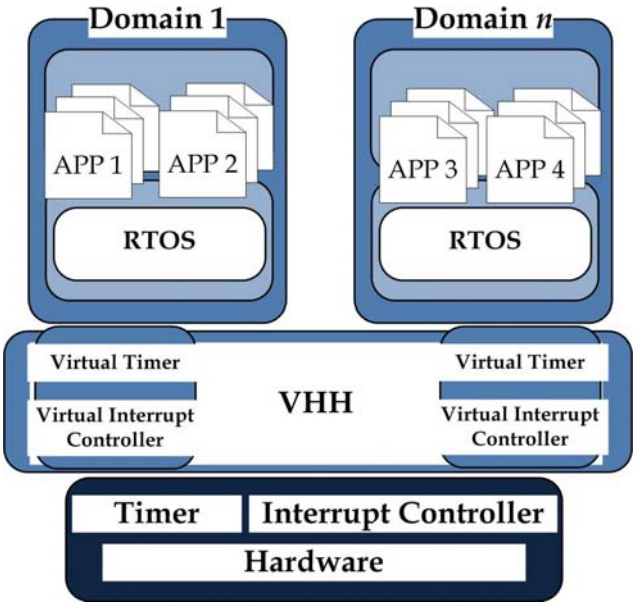


Figure 4.3 – Virtual-Hellfire Hypervisor Domain structure

In terms of memory management, in MMU-less processors a possible choice is to implement a software virtual memory management, as proposed in [CG05]. A second viable strategy is to use

a fixed partition memory scheme. In this case, the required amount of memory is allocated to each domain at boot (or load) time, meaning that its size cannot grow or shrink at run time. If the application code of the guest OS of a given domain requires dynamic memory (such as with *malloc* or *free* C primitives), the heap needs to be managed by the domain's code itself. VHH uses this second option (fixed partition memory), as we can see in Figure 4.4, where each processor (Processing Element - PE, in the figure) of the MPSoC has its own Local Memory (LM). This memory is divided according to the amount of partitions that this processor will hold.

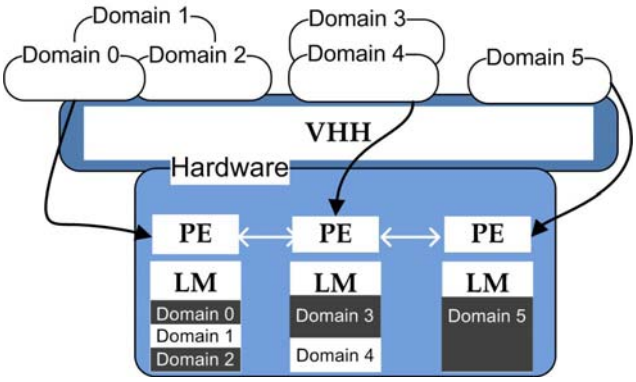


Figure 4.4 – Virtual-Hellfire Hypervisor Memory Management

Another point of concern is dealing with I/O peripherals. Xen [org12] is one of the most successful paravirtualized implementations for desktop systems and it uses the concept of a specific I/O domain (known as *Domain 0*). This is needed because most peripherals must be managed by a single software driver, which is aware of its current status.

VHH also uses this concept, so I/O ports and interrupt lines of peripherals are managed by a specific domain, named *I/O Domain*. This approach is depicted in Figure 4.5, where the highlighted domain is responsible for handling I/O issues. This limits the use of peripherals to the processor that holds the I/O Domain. One possible improvements is to allow other domains to handle it, so that any processor is able to have its own I/O peripheral.

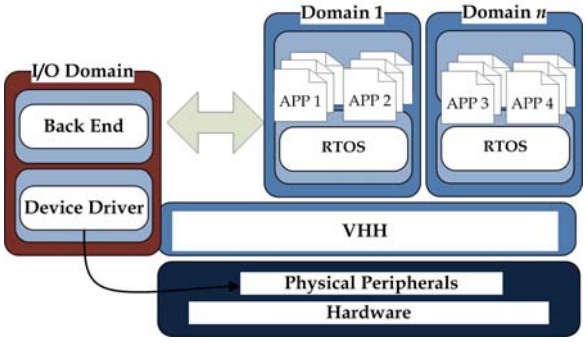


Figure 4.5 – Virtual-Hellfire Hypervisor I/O Handling

The internal architecture of HellfireOS had to be modified to guarantee the use of virtualization. As a matter of fact, we kept some of the original features and took advantage of its

highly modular implementation by adding the necessary modules to provide virtualization. Thus, Virtual-Hellfire Hypervisor is implemented based on the following layers:

- **Hardware Abstraction Layer - HAL**, responsible for implementing the set of drivers that manage the mandatory hardware, like processor, interrupts, clock, timers etc;
- **Kernel API and Standard C Functions**, which are not available to the partitions;
- **Virtualization layer**, which provides the services required to support virtualization and paravirtualization services. The hypercalls are implemented in this layer.

In this new layer responsible for allowing virtualization to be used, there are some mandatory modules, such as:

- *domain manager*, responsible for domain creation, deletion, suspension etc;
- *domain scheduler*, responsible for scheduling domains in a single processor;
- *interrupt manager*, that handles hardware interrupts and traps. It is also in charge of triggering virtual interrupt and traps to domains;
- *hypercall manager*, responsible for handling calls made from domains, being analogous to the use of system calls in conventional operating systems;
- *system clock provider*, in which two clocks per domain are implemented: one that only advances while the domain is being executed (virtual) and a real, counted from the boot time.
- *timer provider*, similar to clock implementation, provides virtual and real timers, both accessible by hypercalls;
- *memory manager*, divided in virtual and physical management, according to the underlying hardware;
- *system output facility*, where all messages are queued and can be redirected to hardware peripherals, such as a serial port.

The described architecture of VHH is depicted in Figure 4.6.

A very interesting point of the VHH is the use of an MPSoC as underlying hardware. We assume the use of a Symmetric MultiProcessor and the hypervisor acts as a MultiProcessor RTOS (MP-RTOS). The hypervisor is aware of the several domains and respects their own scheduling policies.

Each processor has its ready task queue, which can contain tasks from different virtual domains. For each processor, the highest-priority task in the ready queue is executed. To avoid starvation of non real-time tasks (when allocated in the same processor of real-time tasks), it is

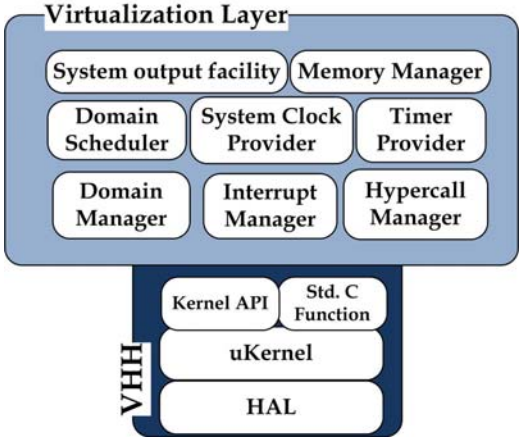


Figure 4.6 – VHH System Architecture

possible to adopt a scheduling policy that guarantees the execution of best-effort tasks, such as R-EDF [YNK01].

The mapping of virtual domains onto real processors is done at design time. It is the designer's responsibility to associate virtual domains and real processors. In the future, there is the possibility of using virtualization even as a load balancing solution, where, dynamically, virtual domains can migrate among the several processors of the MPSoC to improve a given measure, as performance or energy consumption. When more than one domain is mapped for a single processor, the scheduling among domains occurs according to a fixed priority scheduling. Then, domains are scheduled by the hypervisor considering its priority level.

Since HellfireOS is integrated in the Hellfire Framework with several simulation facilities, VHH is also integrated in it and requires the designer to choose whether virtualization is enabled. In this case, although user-transparent, the design flow presented by Figure 4.7 is employed and extended. This flow starts with the configuration of the VHH, where the number of domains is informed and the VHH core is generated. Following, each of the desired domains is configured in a very similar way Hellfire Framework used to do with non-virtual HellfireOS edition: application tasks are added and put together with the OS image. Finally, all system is assembled and executed by an ISS-like (Instruction Set Simulator) simulator.

### 4.1.3 Cluster-based MPSoCs

Since we aimed to use MPSoCs, different architectural arrangements can be thought using virtualization. In this context, cluster-based MPSoCs is a technique that has gained notoriety in the last few years [GIZG+09], [JSZ10]. In this approach the best of both worlds are intended to be placed together: Networks-on-Chip (NoCs) allow higher scalability rates but buses keep the design simpler even with more processors on the system. To better understand the concept, Figure 4.8 depicts a 2x2 sized NoC, which contains a bus located at each local port. Each bus carries along four processors that communicate in simpler ways inside and, if needed, can communicate with other

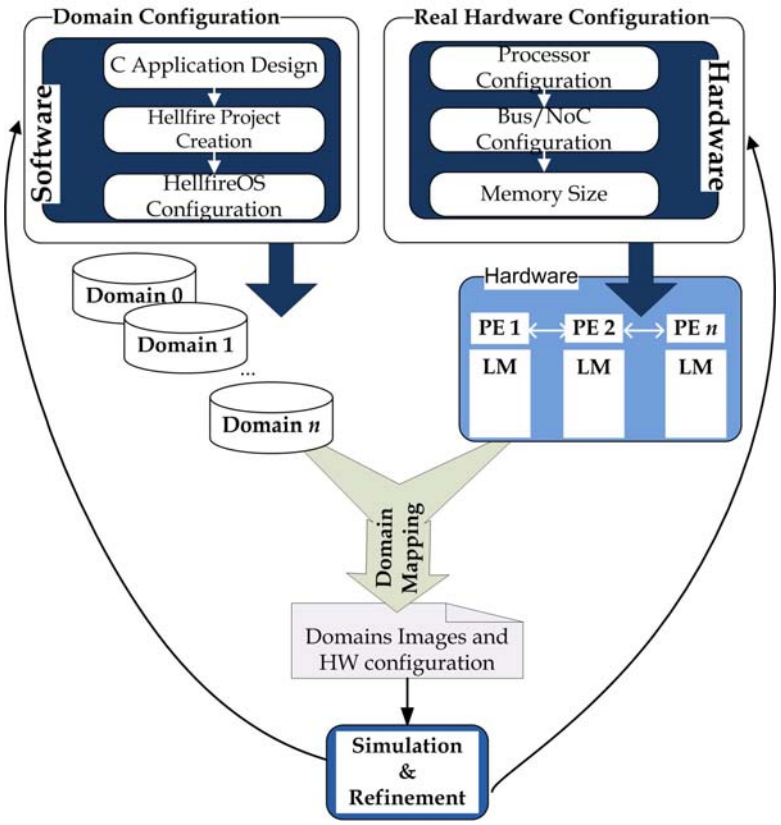


Figure 4.7 – VHH Integrated in the Hellfire Framework

clusters through the NoC. Dotted lines represent the wrappers needed to connect the bus to the NoC.

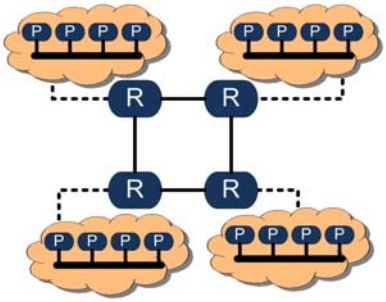


Figure 4.8 – Cluster-based MPSoC concept

So, one spin-off proposal of this dissertation was the unification of both concepts: cluster-based MPSoCs and virtualization. However, instead of using buses on each router of the NoC, we propose a single processor holding a hypervisor, providing the emulation of several virtual processors. Since buses are poorly scalable, hypervisors do not need to support more processors than a simple bus would. The main contribution of this proposal, named as Virtual Cluster-based MPSoCs, is to provide multiprocessed systems with less area occupation.

Virtual Cluster-Based MPSoCs

Since our work is based on the Hellfire Project, we also use the Plasma [Cor13] processor, a MIPS-like architecture. Therefore, the VHH is placed on a Plasma processor as the basis of our cluster. Then, the VHH is responsible for managing several virtual domains. In our case, each VHH is responsible for managing its own processing cluster and it allows the internal communication of these processors through shared memory.

Figure 4.9 is divided in two parts. Part *A* shows the current version for memory division in which only a single memory partition per virtual domain is used. This means that this partition is considered to be the local memory for a given virtual domain. In *B*, it is possible to see that an extra partition was added: the *shared* partition. Here, the idea is to provide easy and low overhead communication inside the cluster.

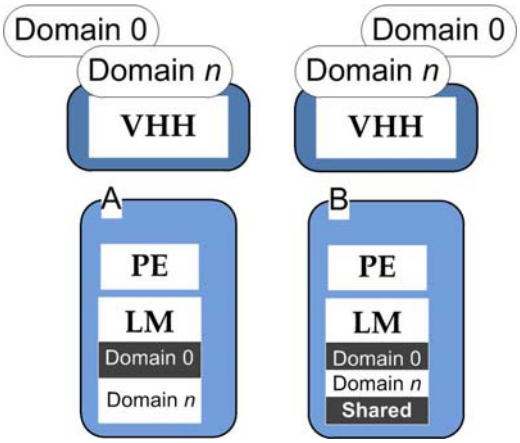


Figure 4.9 – VHH Memory for (A) Non-clustered systems (B) Clustered systems

The VHH was extended to allow the communication in two levels. The first level is named as *intracluster* communication and occurs through shared memory. Currently, this is not user transparent and a specific hypercall must be used for this communication. In this hypercall, a single CPU identification (CPU_ID) must be used, which means they belong to the same processing cluster.

These hypercalls are similar to the communication functions provided by the HellfireOS and have the following parameters: VHH_SendMessage (*cpu_id, task_id, message, message_length*) used to send a message through the shared memory and VHH_ReceiveMessage (*source_cpu_id, source_task_id, message, message_length*) used to receive it.

The second communication level is done among clusters, through the NoC. In our case, we use the HERMES NoC [MCM+04] and a MIPS-like processor in each router. We adopted a Network Interface (NI) as a wrapper that connects the NoC router to the processor located in its local port. This interface works in a similar way that the non-virtualized approach. This increases the possibility of using several NoC infrastructures as the underlying architecture. Figure 4.10 depicts this approach.

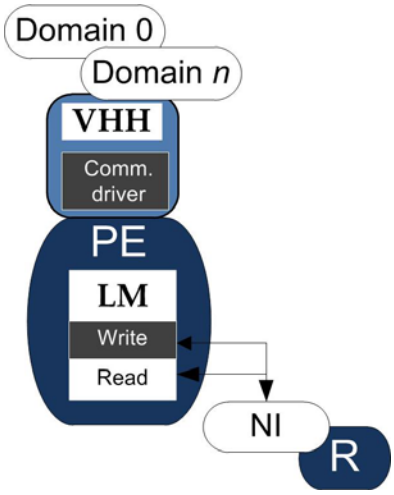


Figure 4.10 – VHH Communication Infrastructure with NoC based Systems

The wrapper is connected to the Plasma core through specific memory addresses: read and write. Still, a communication VHH driver allows the integration between the wrapper and the virtual cluster. Also, the hypercalls provided by the VHH allow a virtual processor to send or receive messages with an extra parameter: the Virtual CPU ID, as an identification of the virtual CPU on a specific cluster.

Thus, the hypercalls to be used to the inter-cluster communication are: VHH_SendMessage-NoC (*cpu_id, virtual_cpu_id, task_id, message, message_length*) used to send a message through the NoC and VHH_ReceiveMessage (*source_cpu_id, source_virtual_cpu_id, source_task_id, message, message_length*) used to receive it.

The complete vision of the system is depicted in Figure 4.11. In the Figure, VHH is the Virtual Hellfire Hypervisor. LM stands for Local Memory. NI stands for Network interface and PE, for Processing Element. R represents each router of the NoC.

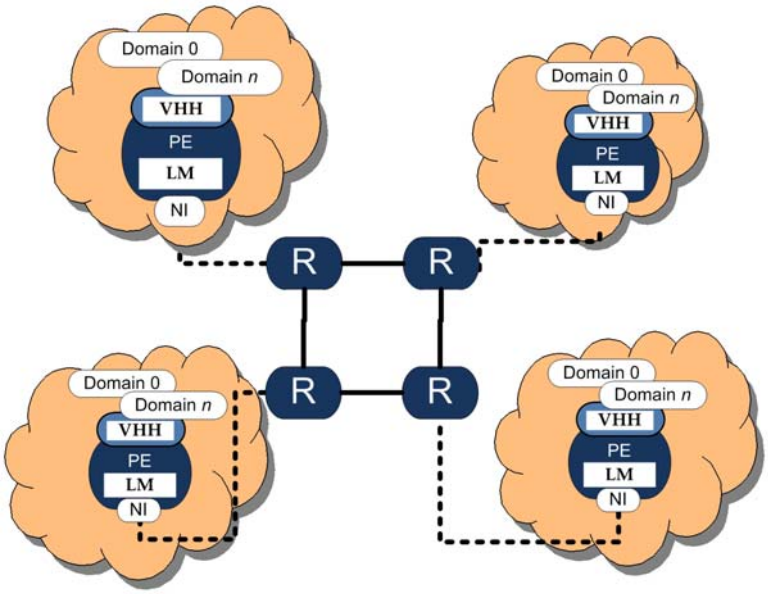


Figure 4.11 – Virtual Cluster-Based MPSoC proposal

Use Cases and Results

The main use for a Cluster-based MPSoC is the possibility for field specialization. In this case, each cluster is responsible for executing a set of tasks with a common purpose. For instance, it is possible to execute a JPEG decoder in one cluster, a MPEG decoder in another and so on. In this case, the greatest advantage is to simplify the communication of similar tasks, since they share a given memory area, but still allowing a great number of processors, increasing system scalability through the NoC usage. Figure 4.12 depicts an example of cluster-based MPSoC with application specialization.

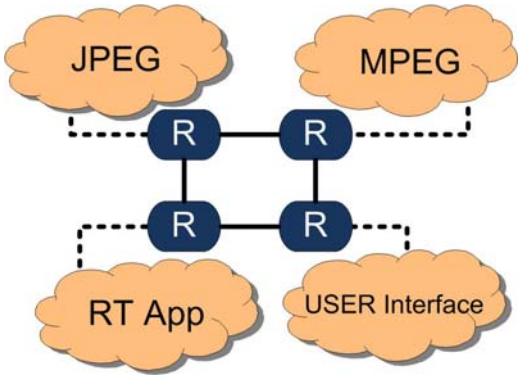


Figure 4.12 – Virtual Cluster-Based MPSoC with Application Specialization

Another possible use of the Virtual Cluster-based MPSoC is when decreasing area with guaranteed system scalability is needed. Scalability is assured by NoC usage and the cluster-based MPSoC itself allows an easier use of real-time tasks with no extra communication penalties. Regarding area occupation, we prototyped some possible configurations to illustrate the benefits of our approach in this issue. We used the Xilinx Virtex-5 XC5VLX330T FPGA.

First, when using the HellfireOS with a Plasma processor, we usually indicate a processor with at least 16KB of local memory. HellfireOS is a much optimized kernel and depending on the application even such a small memory can fulfill the expected needs. When using the VHH, more memory is required and the total memory size depends especially on the number of virtual domains that are required. Although greater memory sizes infer more block RAMs, it does not affect the FPGA area measured in LUTs. In all experiments performed, the total system memory could be inferred as block RAMs.

We used three different MPSoC configurations, all with 16 processors (physical or virtual). First, we have a 16 processor MPSoC, distributed in a 4x4 NoC where each router carries its own processor, known as *Pure 4x4 NoC* approach. The second MPSoC configuration is a 2x2 NoC with bus-based clustering system, known as *Bus Clustered* approach. Here, each router has a wrapper to connect it to the clustered-bus, and each bus carries four processors. Finally, the last approach is the Virtual cluster-based (*V-Cluster 2x2 NoC*) where a 2x2 NoC was used again and each router contains a single physical processor. This processor runs the VHH, where 4 virtual domains are emulated per cluster, totaling the 16 processors of the MPSoC.

In the first two solutions, each processor has 16KB of local memory. The last, for the virtual cluster approach, 4 processors with 128KB of memory each were employed. In Table 4.1, it is possible to see the prototyping results for three different MPSoCs.

Table 4.1 – Area results for MPSoCs configuration

| **Configuration** | **Area occupation (LUTs)** |
|---|---|
| Pure 4x4 NoC | 60934 |
| Bus Clustered 2x2 NoC | 56099 |
| V-Cluster 2x2 NoC | 17179 |

These results show a decrease of the area occupation in up to 70%, depending on the processor local memory configuration and the original MPSoC configuration. Also, depending on the bus structure used for the Bus-based clustered version, the bus communication overhead is similar to the virtualization overhead.

### 4.1.4 Summary

The first attempt consisted in using a previous research of the research group and extend it so that virtualization was possible. However, by the time the implementation of VHH was being performed, we decided to pursuit another path. We made this decision because, observing the use of virtualization in desktop systems and considering some researches [Hei09], [BGDB10], [SCH+09] it became clear that hardware support to virtualization was mandatory. Then, considering that by the time this research was being developed, MIPS had no announcements regarding virtualization support, we decided to modify an existing platform to provide virtualization support.

## 4.2 Second Attempt - Virtualization Model and Hardware Implementation

By the time we decided to add support for virtualization in an embedded architecture, we firstly defined a model that would lead the following implementation attempts.

### 4.2.1 Virtualization Model

This model is focused on multiprocessed embedded systems and on providing some real-time support to applications. The overall model is depicted in Figure 4.13 and detailed in the remainder of this section. Some key concepts of our model are described hereafter.

- ***Application Domain Unit - ADU***[1]. Each Application Domain Unit corresponds to a virtual machine and is intended to be used to divide the system into specialized pieces.

[1] Also referred during this paper as *virtual domain*, *virtual machine*, and *application domain*

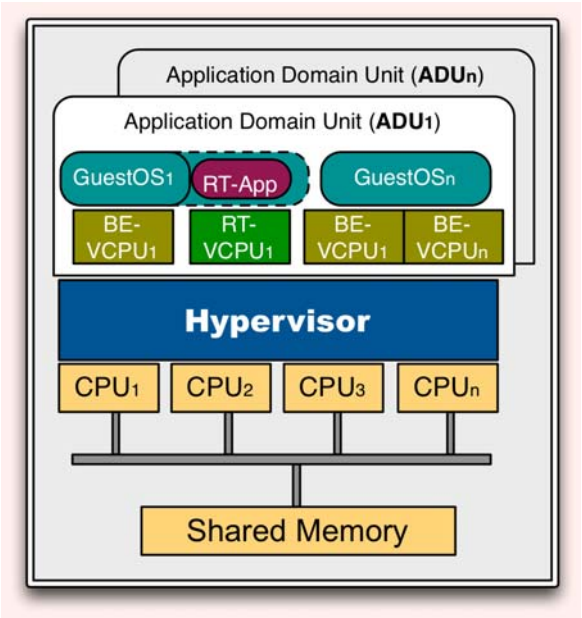


Figure 4.13 – Virtualization model for embedded systems

- ***Non-Real-Time or Best-Effort Virtual Processing Unit - BE-VCPU***. Each application domain can be composed by one or several Virtual Processing Units, which are individually scheduled onto physical processing units. Thus, different mapping strategies can be used. For example, VCPUs from the same virtual domain could be placed onto different physical processors to increase their performance. Still, a VCPU contains a copy of the physical processor's registers.

- ***RT-VCPU***. A given domain may require support to real-time. In that case, they need to count on real-time VCPUs that also have execution privileges to accomplish their constraints. The strategy to offer real-time is addressed later in this section.

- ***RT-App***. Each real-time application may be divided into real-time tasks that should be explicitly indicated so the hypervisor can schedule them accordingly.

- ***guest OS***. guest OSs where their tasks can be executed.

- ***Hypervisor***. The core of our virtualization proposal is carefully implemented aiming to reduce the overheads of a virtualized platform. It manages the creation and execution of VCPUs and Application Domains. Besides, the hypervisor is responsible for a scheduling scheme where the physical processing units are always aware of the next VCPU that needs to be executed, decreasing their idle time.

- ***Physical Processing Unit - CPU***. We propose the virtualization of a MIPS-based platform. We expect this model to be used in multiprocessed embedded systems connected through a bus. Although we are aware of the limitations of bus-based architectures, we intend to provide nodes for future use in cluster-based multiprocessed embedded systems [AdMH11], as

discussed earlier. The quantity of physical nodes can be limited by the bus implementation's constraints.

The overall model presented in Figure 4.13 has a large dependence on the real implementation to be worthy and we present our strategies regarding the processor later in this section. We adopted the concept of VCPUs and CPUs as it allows more flexible mapping strategies.

Initially, each virtual domain has a given task-set, associated with its VCPUs. However, from the VCPUs point of view, a single subset of the entire domain's task-set is available and is managed by the domain's guest OS. This subset can be considered as the VCPU's task array.

From the entire system point of view, we have the possibility of many VCPUs per domain, as if in a matrix arrangement. Each matrix element is independently mapped onto the CPUs. Since we are providing a bus-based virtualization node, the CPUs can be represented as an array of physical processors available in the system.

Thus, the separation provided by the virtualization model we propose can ease the dynamic mapping of tasks among VCPUs (if supported by the guest OS), VCPUs among CPUs and even tasks among CPUs. Figure 4.14 depicts this flexible mapping model for virtualized architectures. However, further investigation of the possibilities of this strategy are not in the scope of this research.

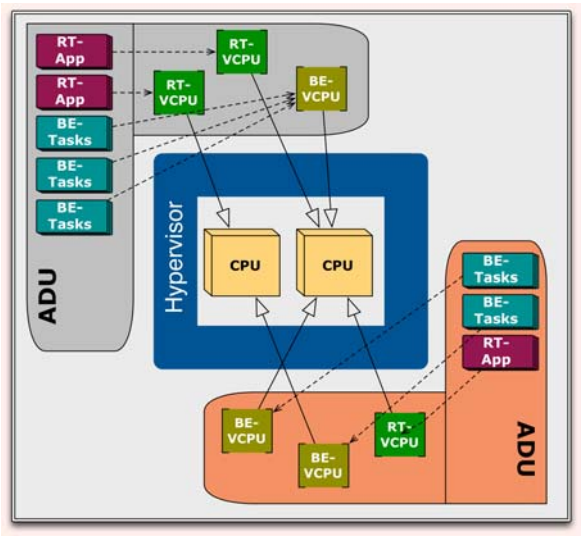


Figure 4.14 – Flexible Mapping model for multiprocessed embedded systems

### 4.2.2 Hardware platform

The virtualization model was firstly described as a hardware platform in VHDL language and the main hardware modules are showed in Figure 4.15. We use a Plasma MIPS CPU, which is a small synthesizable 32-bit RISC microprocessor that supports an interrupt controller, UART, SRAM or DDR SDRAM controller, and an Ethernet controller obtained from Opencores.org [Cor13]. The

Plasma CPU executes all MIPS I$^{TM}$ user mode instructions[2] except for *unaligned load* and *store* operations. In addition, the virtualization capabilities added to the Plasma MIPS core resulted in the vPlasma MIPS, to be detailed later in this section. We adopted the Plasma processor since its VHDL description is freely available at OpenCores.org, which allows us to modify and prototype new versions of it. Also, it occupies small area, consumes low-energy and does not contain extra features, which makes it efficient to add only the structures needed to perform virtualization. Moreover, besides a licensing scheme that allows us to modify, Plasma has a software toolchain that can also be adapted when needed.

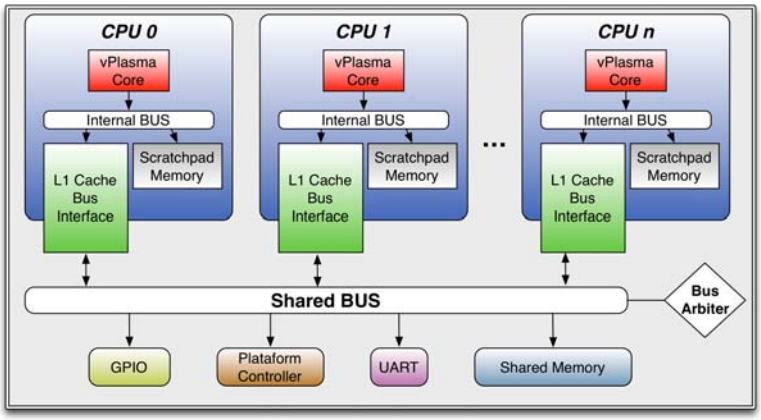


Figure 4.15 – Virtualization Hardware Platform main modules.

We use a multiprocessed platform in which several CPUs are interconnected through a *Shared BUS* that relies on a *Bus Arbiter*. Each CPU has a local memory (*Scratchpad Memory*) and an *L1 Cache*. CPUs have full visibility of all peripherals available at the *Shared BUS*, including the *Shared Memory*, where the ADUs are allocated. The hypervisor implements access policies to peripherals and manages the MMU for each processor, as described later. The *GPIO* device is useful for external communication. The *UART* is the main platform's communication device and implements a typical 16550 UART device. Finally, the *Platform Controller* is used to generate interrupts among processors allowing preemptive communication calls. Besides, it is responsible for either enabling or disabling cores. The platform has a hardwired-enabled core responsible for booting the system and powering the other cores up through the Platform Controller.

For inter CPU communication we use a 32-bit wide bus, which is word addressable, byte writeable, single-cycle arbitrated, half-duplex, and supports multiple masters (CPUs) where just one can communicate over the bus at a time. Decisions regarding the bus-owner at a given time are made by the Bus Arbiter. When not in use, both data and address bus' signals remain at high impedance levels. The address bus is always fed by the current master while the data bus is bidirectional, and fed by the master on writes and by the slave on reads. Writes can be performed at the granularity of bytes, half word and words in any possible endianness on the same bus, allowing mixed endian cores to execute without conflicts. A read is always performed on full words. When multiple devices

[2]This means that some kernel mode instructions, such as *rfe*, are not implemented in the core.

request the bus at the same time, the arbiter executes a master selection algorithm in a round-robin fashion, giving access to the next requesting device when the bus is free.

The L1 cache is a direct-mapped 1k-words cache organized in two banks of 64x8 entries each. It is implemented using two 512x32 BRAMS to hold data and two 64x22 LUTRAMS to hold the tags. Moreover, the cache implements the atomicity of the LL/SC instruction pair resulting in a synchronization mechanism between the processors.

### 4.2.3 Hardware virtualization support

Originally, the Plasma MIPS processor counts on only a single full-privileged execution mode and does not count on memory management, which is important for security and isolation. *Co-processor 0* (CP0) is responsible for controlling interruptions and timer, and can only be accessed through two special instructions: *mfc0* and *mtc0*. Such instructions are called *sensitive* [PG74] because their execution can change the processor behaviour. Thus, a mechanism is used to detect and trap such sensitive instructions and, to do that, three key features are implemented in the CPU core: (i) Memory Management Unit (MMU); (ii) Privileged Execution Mode, and; (iii) ISA's modification.

**MMU**. Our implementation is based on a 16-entry *Translation Lookaside Buffer* (TLB) and Figure 4.16 shows the MMU's block diagram.

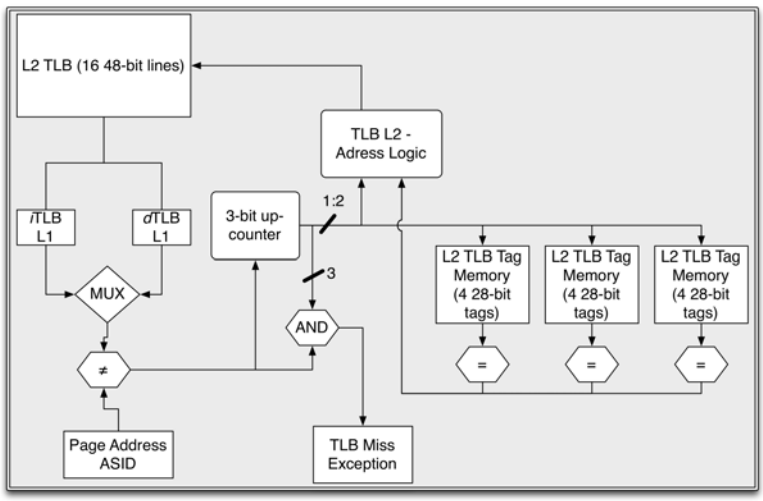


Figure 4.16 – MMU block diagram.

When *enabled*, the MMU works by keeping the line of the last successful translation for both instruction fetch and data access into two 48-bit registers, named *iTLB* and *dTLB* for instruction and data access purposes, respectively. Then, in a scheme called *L1 TLB*, a comparator is used in order to validate the translation. The *L2 TLB* unifies both instruction and data and uses one 16x48 LUTRAM and three 4x28 LUTRAMs for its line tags. A successful search is performed in up to *4* cycles: a *3*-bit up-counter, enabled by a *L1 TLB* miss, has its *2* lower bits used to

address the row and the tag memories, which outputs are compared with the current *ASID* and *virtual address* in order to determinate the translation. The combined *4*-bit address is used to address the 16x48 LUTRAM that feeds back the *L1 TLB*. The *L2 TLB* is responsible for generating a *TLB miss exception* when none of the proposed translations are valid. In this case, the hypervisor feeds the *TLB L2* using the *tlbwi* and *tlbwr* privileged instructions, specially implemented to the vPlasma MIPS core. The MMU is *disabled* either at reset or whenever a software trap is performed. When desirable the MMU can be *enabled* using a special instruction called *jump register into virtual address* (JR.V), which enables the MMU and jumps to the requested address.

**Privileged Execution Mode**. In order to accomplish Popek and Goldberg's virtualization requirements [PG74] the core execution is divided in two distinct modes: *kernel* and *user*. Sensitive instructions (*mtc0*, *mfc0*, *tlbwi*, *tlbwr*) can be executed only in *kernel mode* (privileged). If an attempt to execute these instructions in *user mode* occurs, an exception is generated and the CPU enters into *kernel mode*, passing the control over to the hypervisor. In *kernel mode*, the MMU is automatically disabled so the hypevisor has full visibility of the *Shared Memory*. The *rfe* instruction is used to return the processor to its correct the state, that is, before the exception occurred.

**ISA's modification**. We added new instructions to the Plasma MIPS core ISA, in addition to *tlbwi* and *tlbwr*, presented earlier. Figure 4.17 brings a sequence diagram that shows software and hardware actions during an instruction emulation.

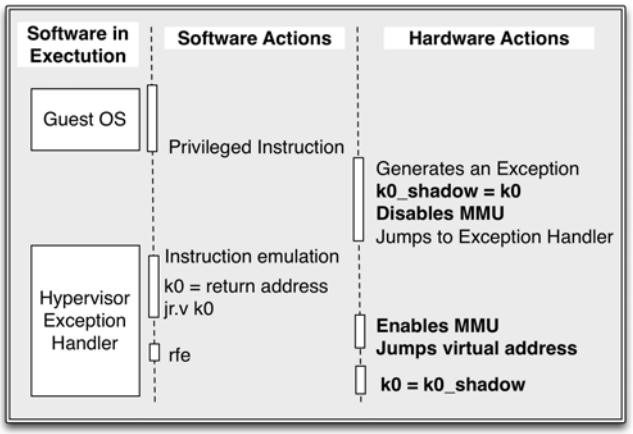


Figure 4.17 – guest OS privileged instruction execution.

Originally, MIPS I ISA contains a instruction pair to return from exceptions: *jr* and *rfe*: *jr* jumps to address contained in the specified register whilst *rfe* is used in the branch delay-slot to return the processor to its prior state. However, this behaviour prevents a correct virtualization from functioning in MIPS I processors, since a register must be used to jump to the exception return address, forcing the modification of its original value, thus disrupting the normal operation of the ADU.

To solve this problem we implemented a modified version of the *rfe* instruction and added a new one to jump to a virtual address, named *jr.v*. Also, a new register (register #30 at CP0)

was implemented as a shadow of the existing *K0* MIPS register[3] (named *k0_shadow*). In this case, when a exception occurs the last value of *K0* is saved in the *k0_shadow* register by the hypervisor. In the exception handling routine, registers are saved by software normally and restored before the return. However, *K0* is used to indicate the return address to the *jr.v* instruction. Such instruction is responsible for enabling the MMU, meaning that the address stored at *K0* corresponds to the virtual address which indicates the exception return address of the ADU. So, the modified version of the *rfe* instruction assigns the value preserved at the *k0_shadow* into *k0*, keeping register value consistency to the ADU.

### 4.2.4 Virtualization software and real-time

Figure 4.18 depicts a general view of our hypervisor composed of the following modules: (i) *Hardware Abstraction Layer (HAL)*, used to isolate layers, such as domain and scheduler from further hardware details. It merges drivers interface, along with device drivers implementation, besides handling the VCPUs abstraction. The exception handler and other low level facilities are implemented directly in assembly aiming to obtain maximum performance; (ii) *Memory-Mapped I/O (MMIO)*, which manages the memory-mapped devices; (iii) *Application Domain Unit (ADU)*, previously described; (iv) *Real-time and Best-effort schedulers*, responsible to implement the EDF (for real-time constraints) and best-effort scheduling policies; (v) *Dispatcher*, responsible for dispatching the chosen VCPU to the physical CPU; and (vi) *Toolkit* that reunites a collection of software facilities, such as linked-list manipulation procedures.

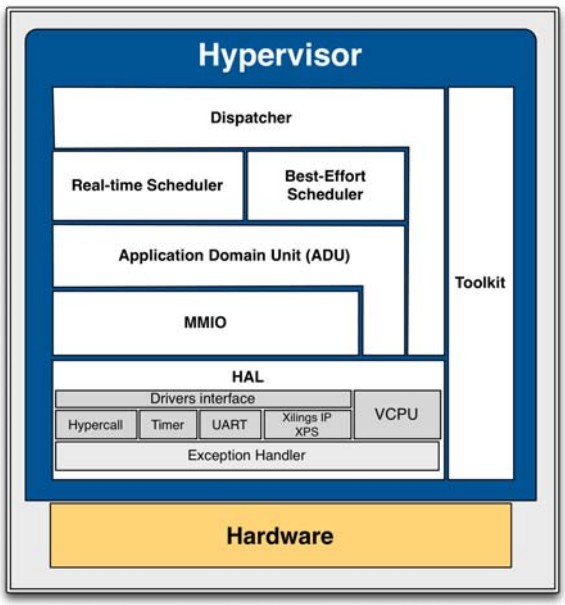


Figure 4.18 – Hypervisor block diagram

[3] *K0* register is normally dedicated to kernel use.

**Hypercalls**. The hypervisor implements the hypercall concept to allow an ADU to instantiate several VCPUs (RT- or BE-). Hypercalls are widely used in paravirtualization based approaches, but in a way so that privileged instructions of the guest OS are replaced by hypercalls. In our approach, hypervalls are only used to instantiate VCPUs to configure properly the ADU.

The system starts with a fixed number of domains configured previously by the designer, where each domain owns at least one BE-VCPU. Then, dynamically, each domain can manage its own VCPUs, as needed by the application. BE-VCPUs need to have a priority parameter define for their creation while RT-VCPUs require real-time task model parameters, such as deadline, period, and capacity to be properly handled by the hypervisor's real-time scheduler.

In practice, a hypercall consists of a write performed by the guest OS at a special memory address (*0xFFFFE800*) that causes a trap to the hypervisor. We also use this technique to emulate a shared peripheral, such as UART. The value written at *0xFFFFE8000* must reflect the address of a struct containing the new VCPU data, which contains parameters such as capacity and period (for RT-VCPUs) and priority (for BE-VCPUs).

The sequence flow that represents a guest OS's execution of a hypercall to instantiate a new RT-VCPU is depicted in the sequence diagram of Figure 4.19. Initially, the *create_rtvcpu()* system call is used as the responsible to fulfil a given struct (named *create_rtvcpu_cmd*) and write its address at *0xFFFFE800*. Then, when the hypervisor assumes the execution its first action consists of determining the hypercall type (in this case, an RT-VCPU creation). Next, the admission control algorithm is executed and, if there is enough system resources, the new RT-VCPU is accepted and assigned to a physical processor. Then, a proper return (indicating either success or failure) is sent back to the guest OS, which follows its own execution flow.

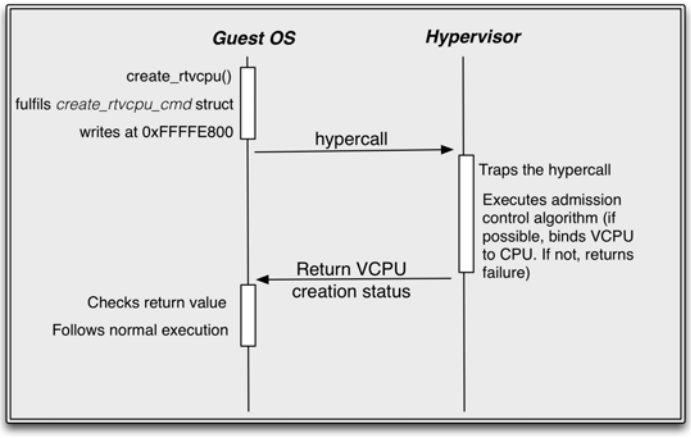


Figure 4.19 – Sequence diagram of RT-VCPU creation

**Scheduler**. The hypervisor implements both Earliest Deadline First (EDF) policy [HT94] and Best-Effort policies to deal with RT-VCPUs and BE-VCPUs, respectively. For the BE-VCPUs implementation, a single global Best-Effort queue is kept, while RT-VCPUs are kept in local individual queues per processor. The EDF is the main hypervisor's scheduler and it has a higher execution priority than the best-effort scheduler, which will not suffer from starvation since we use time

reservation for it. Figure 4.20 presents this two-level scheduling scheme, where RT-VCPUs and BE-VCPUs are placed in different positions (global and local individual queues).

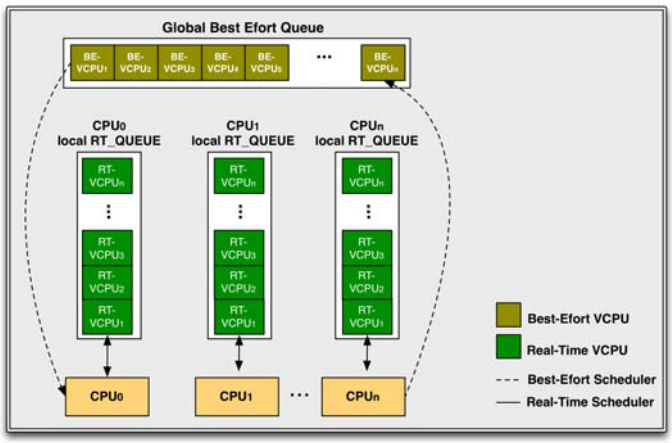


Figure 4.20 – Real-time and best-effort multiprocessed strategy

**Scheduling Sample.** In this example, we show how three VCPUs are scheduled in a system with a single processor. We have two RT-VCPUs and one BE-VCPU. For sake of simplicity, we will keep the period equal to deadline, using the EDF algorithm. We use the following real-time parameters for RT-VCPUs (*p* stands for period, *d* stands for deadline and *c* stands for capacity or worst-case execution time):

- **RT-VCPU 0**: p = 5, d = 5 and c = 3;
- **RT-VCPU 1**: p = 4, d = 4 and c = 1;

The overall system real-time utilization is determined by the $\sum \frac{c}{p}$ of all RT-VCPUs. In this case, we have a real-time utilization of 0.85 ($\frac{3}{5}+\frac{1}{4}$), it means that 85% of the system is dedicated to the RT-VCPUs. The remainder 15% is dedicated to BE-VCPUs.

Figure 4.21 shows the system scheduling when both RT-VPCUs utilize all of the capacity assigned to them. RT-VCPU *1* is scheduled first due to its nearest deadline, followed by RT-VCPU *0*. BE-VCPU only starts executing at tick time *9*, when both RT-VCPUs are waiting for their release time.

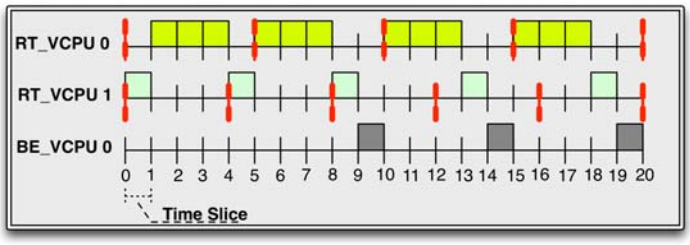


Figure 4.21 – Real-time scheduling sample

Moreover, Figure 4.22 shows the scheduler behaviour when RT-VPCU *0* releases itself from the processor, in an operation we named *Voluntary Preemption*. In this case, RT-VCPU *0*

yields the processor in the middle of time slice *3* and the best-effort scheduler is invoked, scheduling BE-VCPU *0* until the end of the time slice, when a preemption occurs. By the time slice *7*, another voluntary preemption happens, and BE-VCPU *0* is scheduled again. Finally, at time slice *9*, both RT-VCPUs *0* and *1* are waiting for their release times, allowing BE-VCPU to be scheduled again.

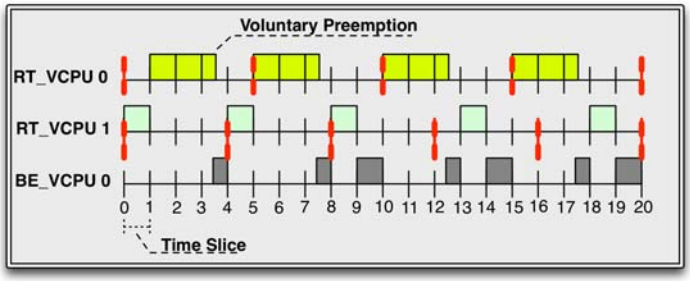


Figure 4.22 – Real-time scheduling sample with voluntary preemption

**Time Reservation.** To avoid starvation of BE-VCPUs, we adopted a time reservation strategy. In the moment of their creation, each ADU must indicate the system load capacity it desires to use. From the ADU point of view, this share of the system's entire capacity is seen as its own entire capacity. All the VCPUs (RT- and BE-) created by this ADU share its slice of the entire system's capacity. If a domain tries to allocate more than its maximum capacity, new VCPUs will fail to be created as they will not succeed through the admission control algorithm.

**Admission Control.** Whenever an application domain requests the creation of a new VCPU, the admission control algorithm checks if there are enough physical resources that can satisfy a given VCPU's requirements. In the particular case of SMP systems, even if there is enough free capacity in the entire system, this may not guarantee the new VCPU's execution since this capacity is actually fragmented among several physical CPUs. Thus, there is indeed an efficiency issue regarding the local CPU queues usage, but with the proper use of dynamic load balancing techniques this problem can be reduced in the future.

### 4.2.5 Evaluation and Results

Hardware evaluation

The VHDL description was synthesized to a Xilinx Virtex-4 XC4VLX60 FPGA Device using ISE 13.2 software. We performed three different synthesis in order to obtain the platform area occupation: the pure vPlasma MIPS core (without *schatchpad* and *L1 cache*), vPlasma processing node (with *schatchpad* and *L1 cache*) and the entire platform with four CPUs. Results are presented in Table I. The addition of the *L1 cache* and *scratchpad* represents an increase of about 1% in the total area occupation. Such area is a small price to pay considering the performance increase it brings. The entire platform with four vPlasma processors occupied 41% of area, and it is possible to synthesize up to 8 processors in the available FPGA with 78% of are occupation. It is important to

point out that each processor may contain more than one virtual processor. In this case, for many embedded applications that require high performance and therefore a high number of processors (typically applications that are implemented by NoCs), virtualization becomes an attractive alternative. Once you use less area, power consumption is lower if compared to a NoC approach, and the performance overhead does not significantly affect the entire system performance.

Table 4.2 – Synthesis results on a Xilinx Virtex-4 FPGA.

| | **Number of occupied Slices** | **FPGA occupation (%)** |
|---|---|---|
| Pure vPlasma core | 2.226 | 8 |
| vPlasma processor | 2.658 | 9 |
| Entire Platform (4 CPUs + interconnection and memory) | 11.087 | 41 |
| Entire Platform (8 CPUs + interconnection and memory) | 21.016 | 78 |

Software evaluation

We have determined the overhead of our implementation based in instruction counts for three different situations: (i) privileged instructions emulation; (ii) context switching (among virtual machines), and; (iii) device emulation (for shared devices). For the first and second cases the guest OS execution causes a trap to the hypervisor. For the third case the guest OS is preempted by the hypervisor and, if convenient, a new guest OS is scheduled.

Thus, analysing the instruction count for all different instructions we emulated, we achieved an average of **230 instructions for the emulation of a privileged instruction.** We used the same technique to determine the overhead of creating and deleting VCPUs (both BE and RT). For that, we got an average of **840 instructions for the creation of a new VCPU** and of **712 instructions for the deletion of a VCPU**. The overhead of the emulation of a shared device was determined. Our emulated device is a UART port dedicated to communication to the external world. It represents a very simple device, where reading or writing a byte from/to the external word consists in an access to the 0xFFFFE000 address. In this case, the shared memory-mapped device is not mapped to a specific guest OS, thus, a reading or writing performed in this specific address causes a trap to the hypervisor, which then emulates the device. **The average overhead detected is 245 instructions**. Although this can be considered as a very optimistic result, it is important to highlight that the more complex the device is the higher overhead it contains. Finally, we obtained the overheads of the EDF scheduler and **we detected an average of 612 instructions to preempt and schedule a new VCPU using the EDF algorithm**.

In order to validate our implementation, we elaborated an experiment where we aim to demonstrate the correct functioning of our virtualization system. The experiment consists of a pro-

ducer/consumer application, a classic example of synchronization problems. Our producer/consumer implementation consists of a producer and a pool of consumers that share a common, fixed-size ring buffer as the producer allocates data in the ring buffer at a constant rate. In order to simulate variable process time, the value allocated in the ring buffer coincides with the number of time slices the consumer needs to process it. The producer/consumer is monitored by a BE-VCPU, known as management VCPU and the one responsible to control the load of the system. The ring buffer has a maximum configurable capacity of *50* integer elements. The system starts with a domain containing the management VCPU, a producer RT-VCPU and a consumer RT-VCPU. The management VCPU is allowed to instantiate three more consumers. Figure 4.23 illustrates this scenario.

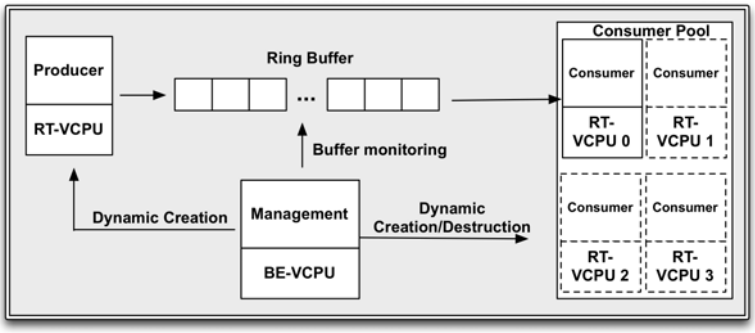


Figure 4.23 – Producer-consumer virtualization scenario

Firstly, RT-VCPUs from *1* to *3* are instantiated and destroyed dynamically by the management VCPU, which monitors the ring buffer when its occupation exceeds *80*%. If the buffer occupation reaches *100*%, the producer stops sampling. Still, the producer has a fixed CPU time reservation of *20*%. Each consumer, when executing, gets *10*% of CPU time. Thus, the system starts with a *30*% CPU usage rate, which can grow until *60*%, since *40*% of system capacity is reserved to best-effort tasks. Figure 4.24 illustrates this scenario. During start-up, the buffer occupation increases, since the producer CPU time is twice the consumers' capacity (consumers need one time slice for each buffer entry processing). Around time slice *470*, it is possible to see that the buffer reaches *80*% of its maximum capacity occupation causing the management VCPU to trigger three more consumers to avoid buffer saturation. Then, the system utilization increases up to *60*% as the buffer occupation decreases rapidly. However, around time slice *420*, the producer starts generating data that takes two time slices for the consumer to process. Thus, the system's balance is established, with an average of *33*% of buffer occupation and *60*% of system load. Still, around time slice *1700*, the producer starts generating data that takes *1* time slice for the consumer's treatment. Expectedly, the buffer occupation decreases rapidly. Then, the management VCPU reacts to the buffer occupation lower than *40*% and destroys two VCPUS. Then, the system gets balanced again at a *40*% CPU load rate.

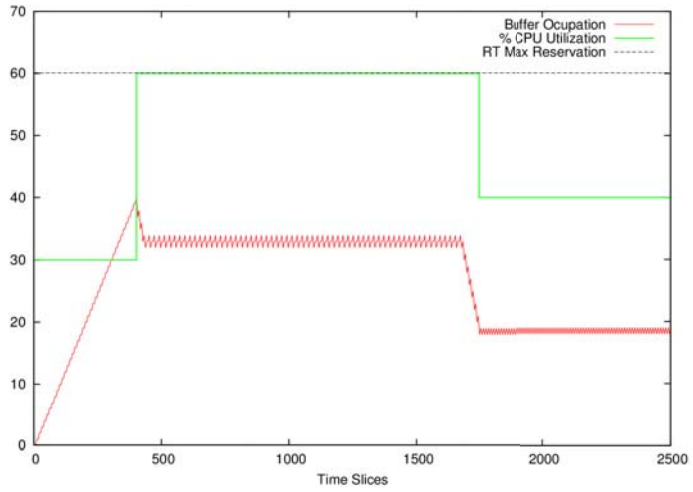


Figure 4.24 – Producer-consumer execution

### 4.2.6 Jitter measurement: Virtualized *vs.* Non-virtualized platform

We used a non-virtualized monoprocessed platform, executing on a MIPS processor running the EDF scheduling algorithm. The virtualized platform is also monoprocessed and uses an ADU for best-effort applications and another for real-time applications.

In a feasible periodic task system, each task's computation must start after its release time and be completed within its deadline. However, since other concurrent tasks exist and also compete for the processor, the tasks' execution may vary throughout the system lifetime. Thus, relatively to a task's release time, the maximum amount of time needed for the task to actually initiate its execution is defined as its jitter (desired to be small and not prevent a task from meeting its deadline). This experiment addresses jitter measurement of virtualized and non-virtualized systems. Figure 4.25 shows a case with 90% of system load usage and it is possible to see the high similarity between virtualized and non-virtualized systems.

As showed previously, the EDF algorithm executed in our hypervisor has higher priority than the best-effort policy, so similar jitter values are expected. The test used a set of three real-time tasks (correspond to three RT-VCPUs in the virtualized domain), that were generated randomly, with a uniform distribution of its period values respecting the *[20, 220]* interval and a fixed system load of *90*%. We are using the EDF scheduling algorithm and deadlines are assumed to be equals to the tasks' periods. The sets were simulated for one million time slices as we measured the jitter on each period and normalized the number of occurrences to plot a graph showing the Normalized Number of Occurrences versus the Jitter in Time Slices for each real-time task, ordered by increasing deadline. Observe that the y axis of each graph (Number of Occurrences Normalized) was plotted on a logarithmic scale to better illustrate the behaviour since most of the occurrences are close to *1*. It is important to notice that the real-time task *0* does not appear in the graph since it has the smallest deadline the EDF scheduling algorithm guarantees its jitter to be null. This experiment's

results show that the real-time tasks can meet their deadlines in a virtualized system even when sharing processing time with best-effort applications.

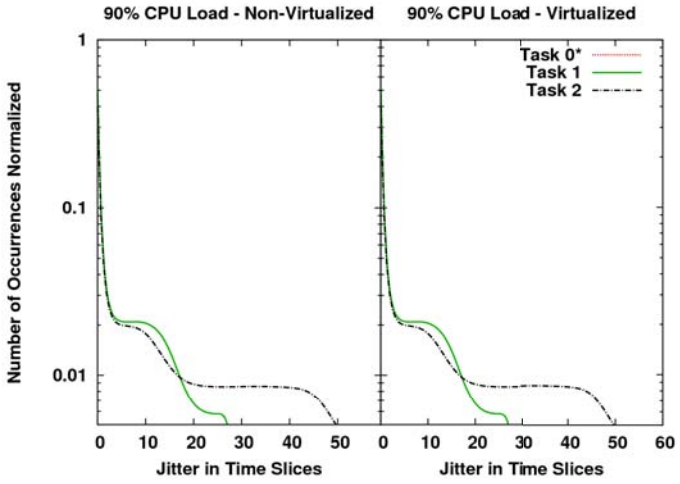


Figure 4.25 – Jitter measurement for system load of 90% in virtualized and non-virtualized platforms
*Task *0* does not appear in the graph because its jitter was null

### 4.2.7 Real-time influence on Best-effort execution: Monoprocessed *vs.* Multiprocessed

We propose a real-time virtualized model, where more than one CPU can be used. One of the greatest advantages to use a multiprocessed platform with a hypervisor is to allow load balancing for CPU usage. To demonstrate that we executed a set of synthetic best-effort applications which, in average, needed a 72133 amount of ticks to be executed (in a non-virtualized platform). Then, we executed this same set of applications in a mono- and a multi- processed virtualized platform. In both cases, the ADU (named as BE-ADU) was placed along with another ADU (named as RT-ADU). The latter instantiates RT-VCPUs in such a manner that the real-time load of the system changes increasingly. The virtualized arrangement was implemented using (i) a single CPU, and; (ii) two CPUs.

Figure 4.26 shows the average execution times of the best-effort application set and the impact it suffers from the increase of the real-time load into the system. In this figure, we show the case for both monoprocessed and multiprocessed arrangements. In the first case, it is possible to see the penalties suffered by the best effort applications regarding the use of real-time applications. Since we use a minimum of 10% of system load reserved to best-effort (to avoid starvation), we could not increase the real-time load in the monoprocessed experiment above 90%. However, while comparing to a non-virtualized approach that adopts two different processors: one for best-effort and another for real-time handling, in spite of the (expected) increased execution time for the best-effort applications, the virtualization approach allows to reduce area and energy consumption by decreasing the amount of needed CPUs in the system.

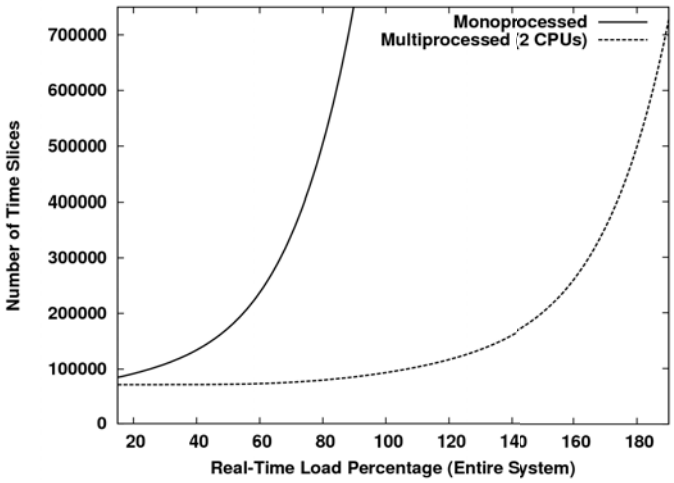


Figure 4.26 – Real-time impact onto Best-Effort execution (mono- and multi-processed virtualized platforms)

In the second case, the curve shows that, for two CPUs, while the real-time load does not exceed the maximum load of a single processor, the best-effort applications do not suffer any sort of penalties. However, as soon as the real-time needs to be spread between the processor, the best-effort applications start to be penalized. The main advantage of using virtualization in a multiprocessed fashion consists of delegating the responsibility of load balancing to the hypervisor while still maintaining high security levels among the application domains.

### 4.2.8 Case-study: porting HellfireOS to modified MIPS

HellfireOS (HFOS) [AFM+10] is a real-time, micro-kernel based, highly customizable operational system. Suitable to run on low memory constrained architectures, like typical critical embedded systems. Primarily, it was designed to run on the Plasma MIPS core. In order to support the core modifications, some effort was necessary to modify the OS to the new vPlasma MIPS core. However, once the HFOS is running on the vPlasma MIPS, virtualize it is straightforward due to our full virtualization approach.

HFOS was originally designed to run on MMU-less processors. Despite the fact that vPlasma MIPS implements a MMU, HFOS does not need to be aware on it, since the MMU is managed exclusively by the hypervisor. Besides, HFOS is not affected by the new privileged execution mode. The hypervisor always schedules a guest OS placing the processor in user mode.

Since the adaptations we performed in the Plasma processor to provide virtualization follow the MIPS R3000 Application Binary Interface (ABI) specifications, the OS modifications also follow this specification. Basically, since the *rfe* instruction was added to vPlasma's ISA, some modifications were required on the exception handler routine. Moreover, simply disabling interrupts to guarantee atomic execution of instructions is no longer enough, and a system call primitive (syscall) was implemented, allowing the OS to have similar functionality. On the OS implementation

level, modifications concern exclusively the Hardware Abstraction Layer (HAL), which is the only hardware-dependent layer of the operating system. In addition, these modifications did not impact neither the OS's code size nor the data memory usage.

### 4.2.9 Summary

This second attempt was focused on providing hardware support to vitualization. In this attempt, we offer real-time support by using bare-metal applications. Our hypervisor implements a scheduling strategy to offer support and future work include further investigation about two-level scheduling and other strategies.

We have adopted a mixed approach with some hardware facilities to virtualization but mainly we use emulation of privileged instructions. This causes some overhead and further hardware support should also be investigated. Still, we decided to try these minimum virtualization support we added in other MIPS-based processor, and analyze the main differences.

## 4.3 Third attempt - Another MIPS-processor modification

After defining the virtualization model, we had some drawbacks regarding the simulation strategy, using VHDL and RTL simulation. The third attempt consists in using another MIPS-based processor and we chose the MIPS 4Kc. This section details the modification and virtualization support we added to this architecture.

### 4.3.1 MIPS4K modification and virtualization support

The MIPS4K family is formed by three members: the 4Kc™, 4Km™, and 4Kp™ cores. The cores incorporates aspects of both MIPS Technologies' R3000® and R4000® processors although they differ mainly in the type of Multiply-Divide Unit (MDU) and the Memory Management Unit (MMU).

In this case:

- the 4Kc core contains a fully-associative Translation Lookaside Buffer (TLB)-based MMU and a pipelined MDU;
- the 4Km core contains a fixed mapping (FM) mechanism in the MMU, which is smaller and simpler than the TLB-based implementation used in the 4Kc core, and a pipelined MDU (as in the 4Kc core) is also used, and;
- the 4Kp core contains a fixed mapping (FM) mechanism in the MMU (like the 4Km core), and a smaller non-pipelined iterative MDU.

Figure 4.27 depicts the most relevant blocks on the MIPS 4Kc core: (i) Execution Core; (ii) Multiply-Divide Unit (MDU); (iii) System Control Coprocessor (CP0); (iv) Memory Management Unit (MMU) and TLB; (v) Cache Controller; (vi) Bus Interface Unit (BIU); (vii) Instruction Cache (I-Cache), and; (viii) Data Cache (D-Cache).

Among these blocks, there are a few points we need to highlight. First, the CP0 in the responsible for controlling the TLB, the cache protocols, the processor modes of operations and interruptions. Still, this CP0 contains 32 registers that differ from the 32 general-purpose registers contained in the MIPS architecture. Finally, these specific CP0 registers can only be accessed through the use of the privileged instructions *mtc0* and *mtf0*. Whenever these instructions are executed in User mode, a trap is generated.

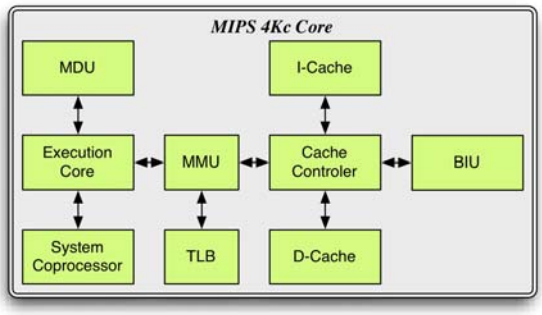


Figure 4.27 – MIPS 4K core

### 4.3.2 Memory Management

The MMU in a 4K processor core is conceived to translate any virtual address to a physical address before sending requests either to the cache controllers for tag comparison or to the bus interface unit for an external memory reference. This translation is a very useful feature for operating systems when leading the physical memory to accommodate multiple active tasks in the same memory. Other features handled by the MMU are memory areas' protection and the definition of the cache protocol.

In the 4Kc processor core, the MMU is based in a TLB that consists of three address translation buffers: (i) a 16 dual-entry fully associative Joint TLB (JTLB); (ii) a 3-entry instruction micro TLB (ITLB), and; (iii) a 3-entry data micro TLB (DTLB). Thus, when an address is translated, the appropriate micro TLB (ITLB or DTLB) is accessed first. If the translation is not found in the micro TLB, the JTLB is then accessed. If there is a miss in the JTLB, an exception is taken.

Still, all the 4K processor cores support three modes of operation: (i) **User mode**, mostly used for application programs; (ii) **Kernel mode**, typically used for handling exceptions and privileged operating system functions, including *CP0* management and I/O device accesses, and; (iii) **Debug mode**, used for software debugging usually within a software development tool. For sake of simplicity, we are not considering such mode in this study.

It is important to highlight that the address translation performed by the MMU depends on the mode in which the processor is operating. For example, Part A of Figure 4.28 depicts the differences between the memory segments that can be seen according to the active processor mode. It is possible to observe that whilst, in kernel mode, several segments are available (from *kseg0* to *kseg3*, including the *kuseg*), in user mode of operation, only the *useg* (with virtual addresses equivalent to the *kuseg* segment) is available.

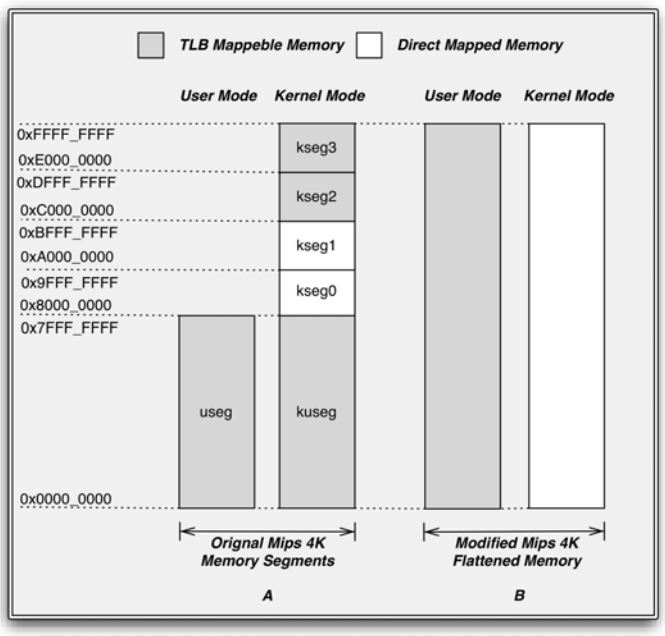


Figure 4.28 – MIPS 4K memory management for User and Kernel modes of operation

**Virtual Memory Segments**. Originally, the MIPS 4K processor contains virtual memory segments, which are differently used depending on the mode of operation, as briefly discussed. Figure 4.28 shows the segmentation for the 4 GB virtual memory space addressed by a 32-bit virtual address for both *user* and *kernel* modes of operation. Initially, the core enters into the kernel mode during reset and whenever an exception is recognized. While in Kernel mode, the software has access to the entire address space, as well as to all CP0 registers. On the other hand, User mode accesses are limited to a subset of the virtual address space (0x0000_0000 to 0x7FFF_FFFF) and can be inhibited from accessing CP0 functions. Still, while in User mode, virtual addresses 0x8000_0000 to 0xFFFF_FFFF are invalid and cause an exception whenever they are accessed.

This virtual memory segments scheme, which is adopted by MIPS 4k core, is very useful for a non-virtualized operating system. In this case, the OS can keep the user application and the kernel isolated by running them in different segments. Typically, the user applications run in User mode in a segment named *useg*, which allows the isolation of the OS software components from user applications with possible malicious behaviour. Besides, in the 4Kc core the OS can isolate the user applications from each other through the limited memory visibility for each application provided by the TLB. This is adopted so a user application with unpredictable behaviour does not influence other system applications.

Still, an important motivation to use virtual memory segments consists in allowing the OS to have privileged access in certain memory areas. For example, the Exception Vector (memory address where the beginning of handler routines for exceptions are placed) located at 0x8000_0000 coincides with the start of the *kseg0* segment, showed in Figure 4.28. Also, another interesting example is the *kseg1* where the cache is disabled to allow direct access to the registers of memory-mapped I/O devices. In this case, both segments' addresses are not eligible to be mapped by TLB, thus, they have a fixed-mapping where both segments *kseg0* and *kseg1* are mapped to the physical address 0x0000_0000.

Although the virtual memory segments scheme is strongly recommended to non-virtualized systems, since it increases software reliability, it brings undesirable restrictions to a scenario where virtualization is desired indeed. MIPS 4K core does not count on a special execution mode for hypervisors and, due to the ring de-privilege situation, the only piece of software that can be executed in privileged mode is the hypervisor itself while the guest OSs will execute in a simple User mode.

Specifically, analysing the 4K core, it means that only the first 2GB of the virtual memory will be available to the virtual machines. A guest OS running in the User mode will not be able to address virtual memory above 2GB. The second - and very critical - limitation can represent a major barrier to achieve virtualization in MIPS 4K core: the fixed-mapping of *kuseg0* and *kuseg1* segments. In this case, the hypervisor needs to register its exception routine under the Exception Vector address (at 0x8000_0000) in order to take the control of the execution of privileged instructions by the guest OS, as well as hardware interruptions, TLB misses and other system conditions.

On the other hand, a guest OS will try to register its own exception handler routine, what conflicts with the hypervisor implementation and possibly with other guest OSs. Since the Exception Vector is located at a fixed-mapped address, the hypervisor is not able to move the virtual address 0x8000_0000 to a different physical address attending the guest OSs' needs. The same scenario description can be applied to the *kseg1* segment, when the hypervisor tries to virtualize a given device.

Therefore, aiming to support full-virtualization on a MIPS 4Kc core, we propose two main modifications on the processor's core:

- removing all virtual memory segments, specially the fixed-address segments (*kseg0* e *kseg1*), and;
- disabling the TLB-Translation when the Kernel mode is active.

The removal of all virtual memory segments implies that no virtual memory address is mapped to the physical memory when the TLB does not have a valid entry. However, once the TLB translations have been turned on and a TLB flush routine has been executed, there is no way to turn the TLB off again. This imposes that a valid entry needs to be kept in the TLB in order to map the hypervisor area to the physical memory.

Such scheme is not transparent for a guest OS that tries to configure its own TLB entries. Then, to avoid such conflicts we have modified the MIPS 4Kc core so the TLB is turned off whenever

the Kernel mode is active. In this condition the modified core translates each single virtual address directly to the matching physical address, thus giving full visibility of the memory only to the hypervisor.

Finally, we extended the visibility of the virtual memory in User mode to 4GB allowing the guest OS to require addresses above 0x7FFF_FFFF. This is necessary when the guest OS tries to access either the Exception Vector or a memory mapped device[4]. The new memory map for both User and Kernel modes are depicted in Part B of Figure 4.28.

### 4.3.3 Logical Memory Organization

We implemented the hypervisor also to be responsible for controlling the memory visibility to each virtual machine using the TLB correctly. In terms of logical memory organization, Figure 4.29 depicts how we divided the implementation of our hypervisor into four different logical areas: (i) hypervisor private area; (ii) hypervisor scratchpad, which holds the hypervisor's stack; (iii) exception vector that contains the MIPS 4K exception vector, and; (iv) available memory that is where virtual machines can be allocated.

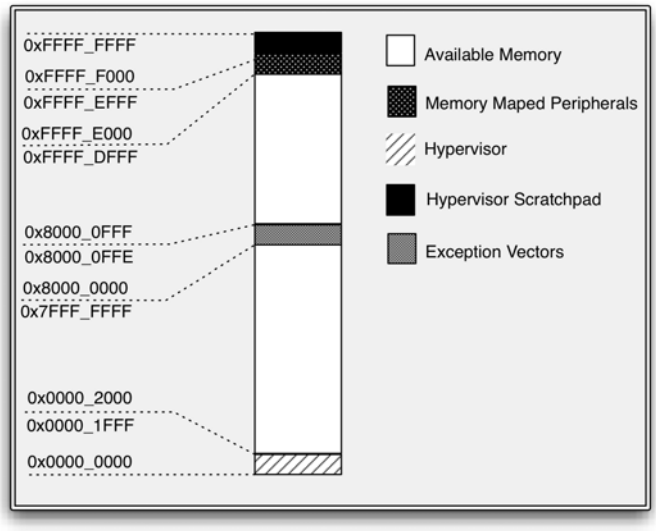


Figure 4.29 – Hypervisor memory logical organization

### 4.3.4 Exception Vector

MIPS 4K contains a fixed address designated for the Exception Vector starting at 0x8000-_0000 (when the processor is operating in any mode except for the debug mode). This causes an address conflict between the hypervisor and the guest OSs because both of them try to register their

[4]It is important to highlight that the guest OS executes in User mode.

exception routine at the same address. Thus, to solve this problem it is needed to create a virtual mapping to the guest OSs, where the physical address 0x8000_0000 is mapped to a virtual address. So, the hypervisor can register its exception routine at the physical address 0x8000_0000 while the guest OSs use a virtual address, as described in Figure 4.30.

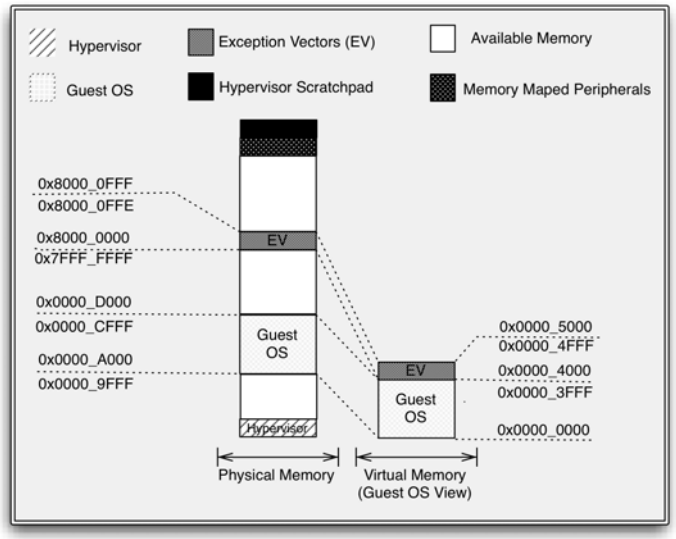


Figure 4.30 – Exception Vector modification

### 4.3.5 Exception Return

The guest OSs run in User mode, as the MIPS 4K core generates an exception whenever a privileged instruction is executed outside of its intended privilege level, enabling the hypervisor to intercept such instruction and emulate it. Then, after the software emulation of the privileged instruction occurs, the hypervisor must return the control to the guest OS. In this context, the ERET instruction (MIPS R4000) is used to return from an exception and the return address used by it is programmed at the EPC (Exception Program Counter). The EPC register at CP0 ($14) has the virtual address of the instruction that was the direct cause of the exception. The hypervisor accesses the address contained in the EPC register to find which instruction should be emulated. After this, the EPC register is incremented to the address of the next instruction and ERET instruction is performed. Figure 4.31 depicts which software and hardware actions are performed when a privileged instruction needs to be executed by the guest OS.

### 4.3.6 Timer

For timing purposes we use the internal MIPS 4K core timer. It is programmed to generate a hardware interruption which causes the control to be assumed by the hypervisor that is responsible for scheduling a new guest OS.

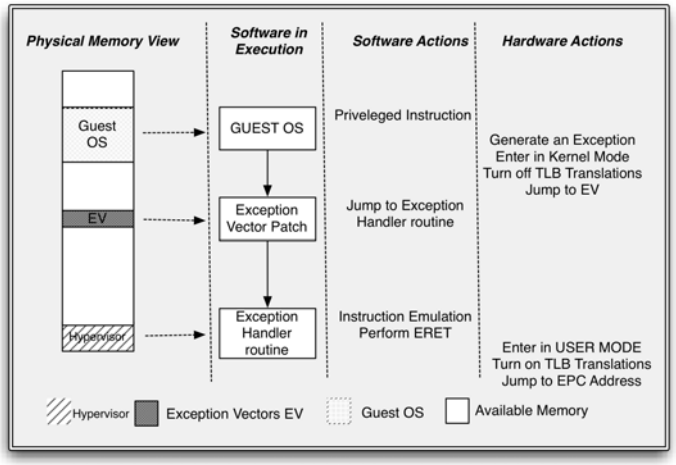


Figure 4.31 – guest OS privileged instruction execution

### 4.3.7 Memory-mapped peripherals

Currently, our hypervisor deals with memory-mapped peripherals using either direct-mapped or shared approaches. In the first case, the directed mapped peripheral technique is desirable when there is a need for high performance and/or when there are real-time constraints. The hypervisor maps the peripheral directly to a guest OS and any requests from other Guests are simply denied. In such way, no overhead is added when the guest OS accesses its directly mapped peripherals. Thus, the implementation of this technique consists in mapping the memory region where the peripheral is located to its guest OS owner, by using the TLB. This guarantees that accesses to a peripheral by its guest OS owner do not trap to the hypervisor, whereas not allowed guest OS accesses do trap to the hypervisor, generating an exception that is treated accordingly.

The shared peripheral approach is desirable for peripherals that are needed by more than one guest OS. For instance, serial ports or ethernet controllers can be considered as shared peripherals because they allow connectivity to the external world and can be used by several guest OSs. This approach requires a more complex treatment from the hypervisor point of view. A shared peripheral does not have its memory area mapped for any guest OS specifically, that is, the peripheral memory area is unmapped in User mode. An access to such area by a guest OS causes a trap to the hypervisor that can identify where the request is coming from and then emulates the peripheral. This means that the hypervisor needs to implement a device driver specifically for each shared peripheral.

A memory-mapped peripheral can be a GPIO pin, a serial port, an Ethernet controller, or even a high-speed PCI-e peripheral. The decision concerning the placement of a certain peripheral, if either shared or directly mapped, occurs at design-time. For instance, if Ethernet capabilities are desired for more than one guest OS, it might be interesting to share this device. Otherwise, if a single guest OS is the responsible for all the Ethernet communication, probably the best decision is to map it directly. Still, high-speed or real-time constrained peripherals should be directly mapped to a specific guest OS due to lower overhead and better response time rates.

### 4.3.8 Multiprocessor concerns

**Synchronization Primitive**. MIPS II provides two instructions for synchronization purposes: *Load Linked* (LL) and *Store Conditional* (SC). However, in the MIPS 4K core, these instructions are originally intended for single processor architectures. Therefore, to provide synchronization among the many processors intended by our architecture, we developed a lock-based system using a memory-mapped peripheral that guarantees atomicity for the following software layers of the system.

**Inter-domain communication**. Our proposal is flexible enough to work on monoprocessed and multiprocessed architectures. Either way, we need to provide a communication mechanism between Application Domain Units. The hypervisor is responsible for this support basically performing a copy from the sender domain's memory area into the receiver's domain memory area. For this to work, the guest OS must have a proper driver that understands our communication protocol, implemented to take advantages of the proposed architecture, reducing possible overheads.

### 4.3.9 Results

To simulate our platform we used OVP [OVP13], which is a hardware simulator written in C language, instruction-accurate, open-source and able to simulate an entire platform. OVP offers a large open-source model database, supporting several processor families (like MIPS, ARM and PowerPC) besides many peripherals. Still, it performs fast simulation aiming to deliver a virtual platform for embedded software development without the need of the real hardware platform.

In our case, the implementation of our virtualization technique requires several modifications in the processor core. Currently, we have no HDL implementation available of a MIPS 4K core.So, we are proposing a modified MIPS 4K core that allows full virtualization to be achieved. In this scenario, the OVP simulator and its open-source models allow us to implement the new processor's core behaviour and simulate our software stack.

Given the lack of a hardware implementation of the architecture and even a cycle-accurate simulator, no real performance evaluation is possible. Thus, tests were only performed using the OVP simulator, which does not model neither memory access nor cache timing correctly. However, the resulting instruction counts can still be used to get an approximate idea of the performance score, besides assuring that the implementation works as expected.

Therefore, we have determined the overhead of our implementation based in instruction counts for three different situations: (i) privileged instructions emulation; (ii) context switching (among virtual machines), and; (iii) device emulation (for shared devices). For cases (i) and (ii) the guest OS execution causes a trap to the hypervisor. For the case (iii) the guest OS is preempted by the hypervisor and, if convenient, a new guest OS is scheduled.

The instruction counts were obtained by configuring the OVP simulator to output the exact sequence of instructions executed by the core in an understandable assembly code format. Such feature increases the simulation time but is useful for a detailed analysis of the execution sequence or for debug purposes. Following, we show a sample of the privileged instruction sequence emulation:

```
1 - 0x00000070 : mtc0    t0,c0\_status
2 - 0x80000180 : sw      k0,-2048(zero)
3 - 0x80000184 : lui     k0,0x0
4 - 0x80000188 : addiu   k0,k0,228
5 - 0x8000018c : jr      k0
6 - 0x80000190 : lw      k0,-2048(zero)
7 - 0x000000e4 : sw      k0,-2024(zero)
9 - ...
10 - 0x00000228 : lw      sp,116(k0)
11 - 0x0000022c : lw      k0,104(k0)
12 - 0x00000230 : eret
13 - 0x00000074 : mtc0    zero,c0\_cause
14 - 0x80000180 : sw      k0,-2048(zero)
```

Code *line 1* contains a privileged instruction which is being executed by a guest OS[5]. The processor core switches to Kernel mode and jumps to the exception vector at the physical address 0x80000180 (exposed in code *line 2*). The exception vector routine jumps to the hypervisor specific handler routine (*line 5*) at the physical address 0x000000e4 (*line 7*). The specific handler routine code was resumed among *lines 7* and *11* for sake of simplicity. Then, *line 12* shows the hypervisor returning the control to the guest OS using the ERET MIPS R4000 instruction explained previously. This instruction jumps to the address configured in the EPC register at CP0 and switches the core to User mode. The next instruction in the guest OS is another privileged instruction (*line 13*) at virtual address 0x00000074. This causes a new exception trapped by the hypervisor and a repetition of this sequence.

Thus, analyzing the instruction count for all different instructions we emulated, we achieved an average of **220 instructions for the emulation of a privileged instruction.** We used the same technique to determine the overhead of a context switch among virtual machines. Context switches between applications on the same guest OS will not trap to the hypervisor and there is no overhead. For the sake of simplicity, at the present time, we consider the round-robin scheduling algorithm running in the hypervisor. **We detected an average of 420 instructions to preempt and schedule a new VM**.

Finally, the overhead of the emulation of a shared device was determined. Our emulated device is a UART port dedicated to communication to the external world. It represents a very simple device, where reading or writing a byte from/to the external word consists in an access to the 0xFFFFE000 address. As discussed previously, a shared memory-mapped device is not mapped to a specific guest OS, thus, a reading or writing performed in this specific address causes a trap to the hypervisor, which then emulates the device. **The average overhead detected is 260 instructions**.

[5]the guest OS is being executed in User mode, since the address 0x000000070 is a virtual address

Although this can be considered as a very optimistic result, it is important to highlight that the more complex the device is, the higher overhead it contains.

Finally, we estimated the number of Lines of Code (LoC) needed by the entire hypervisor: around 2KLoC written in both C and Assembly languages. In this case, around 500 lines are described in Assembly language and represent the Hardware Abstraction Layer (HAL). The rest, entirely written in C Language, is mainly divided in around 150 LoC dedicated to the round-robin schedule algorithm, about 600 lines to implement the VCPU concept, Timer and IRQ emulation as the rest is responsible for other routines.

### 4.3.10 Evaluation methodology

To perform our evaluation we've implemented a peripheral responsible only for measuring time in microseconds. This peripheral is placed in the shared bus and each virtual machine accesses it through emulation, with a 600-instruction overhead. This induces some extra timing to each access that is not significant when compared to the algorithm's results. We use HellfireOS as a guest OS.. Other configuration details are described at each case study, as needed.

### 4.3.11 Processing Overhead Measurement

This test demonstrates the processing overhead of our proposal by comparing it to a non-virtualized solution. We have implemented a CPU-bounded application that implements the classic Hanoi Tower problem [Hay77] resolution. Results were measured from the average execution time of one hundred iterations using a 16-piece configuration. Figure 4.32 shows the virtualization overhead compared to the non-virtualized platform. We varied the amount of physical cores (CPUs), even in the native execution. Then, for the virtualized platform, besides varying the amount of CPUs we varied the amount virtual cores (VCPUs) per physical core.

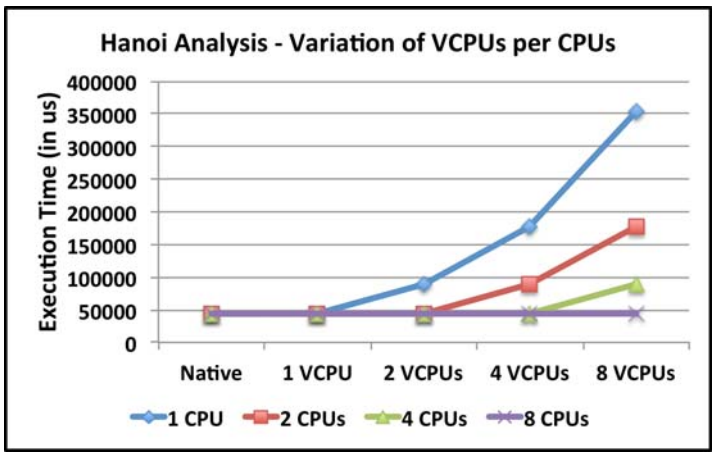


Figure 4.32 – Virtualization execution overhead for the Hanoi algorithm

Best-case scenario can be considered when we have one VCPU per CPU. In this case, the average overhead is not significant (around 0.32%). However, in the worst case comparison, when we have a single CPU and 8 VCPUs, the overhead grows dramatically to more than 700% since we need to share a single physical resource between a bigger amount of VCPUs. Still, there are cases when the total amount of VCPUs is inferior to the amount of CPUs. In these cases the average overhead is around 0.33%, meaning that there is no significant interference of the scheduling scheme in the overall system performance. The compromise of the VCPU per CPU ratio must be carefully analysed by the designer in order to balance the benefits and the cost induced by the platform.

We also analysed the execution and emulation of shared peripherals. To stimulate such problem, we analysed the relationship between the virtualization overhead growth and the amount of UART accesses (per 100000 instructions), depicted in Figure 4.33. It is possible to see that the more UART accesses a program performs, the more its execution overhead grows. However, we believe that depending on the application's behavior our virtualization overhead can still be acceptable. It is also important to highlight that, if a given virtual machine uses extensively a given peripheral, it could be considered to use the direct-mapping strategy, which eliminates the emulation overhead.

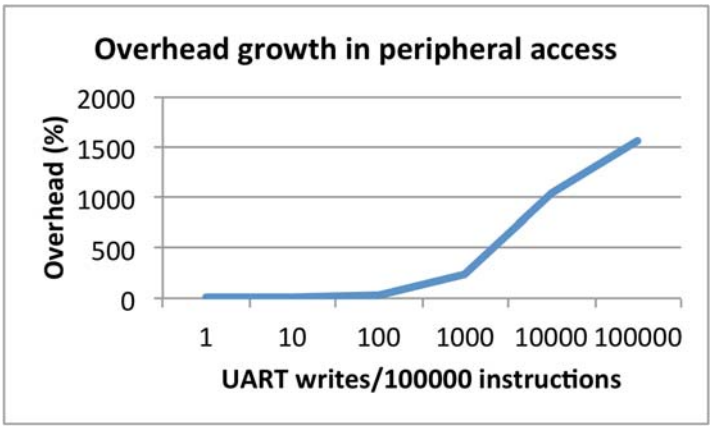


Figure 4.33 – Virtualization overhead at UART accesses

### 4.3.12 Communication Overhead Measurement

Figure 4.34 shows the results of two communication-bounded application. Firstly, the Ping Test application is responsible only for performing message exchanges. Secondly, the Bitcount application contains a master processor that is responsible for counting the amount of set bits in a given array of bits using a given number of slaves to help complete the task. Both applications are executed in virtualized and non-virtualized (native) scenarios. Since these are communicating applications, native execution requires at least two physical cores, varying from 2, 4 and 8 CPUs. In the virtualized environment, we've performed an analysis based in the VCPU per CPU ratio (VCPU:CPU), varying from 1:1, 2:1, 3:1 and 4:1. Still, for the ping test we varied the size of each message, since the larger the message size, the more packets are needed to send it through the

network. Results were taken as an average of execution time needed to send one thousand messages for each test set. For the Bitcount test we use an array size of 4096 and each VCPU as a slave and one thousand executions for the average results. Since this test involves a lot of privileged

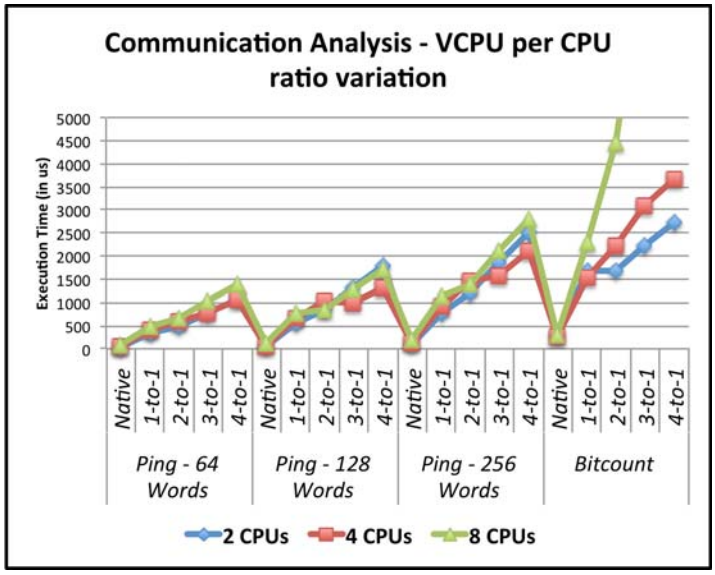


Figure 4.34 – Communication Test with Synthetic Application (Ping Test)

instructions to perform the communication, we observed that the overhead is equally high. Even in the best cases, when the VCPU per CPU ratio was set in 1:1, the communication overhead was in average 800%, due to the trap-and-emulate strategy we use. This occurs mainly due to our trap-and-emulate strategy. We believe that some refinement in the way full virtualization is provided, such as adding different execution modes (as similar to what Intel, AMD, and ARM have done) would definitively decrease the overall execution overhead.

### 4.3.13 Mixed Scenario Measurement

Finally, the third scenario mixes the processing and communication bounded applications to allow an analysis of the interference they cause in each other. We use the Hanoi as the CPU-bounded application and the Bitcount communicating algorithm. Figure 4.35 depicts the execution times in two cases: in the graph on the right side of the figure, we present the Bitcount execution times. On the left side of the figure, we present the Hanoi execution times. For both tests, each VCPU contains one test (1 Hanoi or 1 Bitcount) as we varied the amount of Hanois and Bitcounts per test, also varying the amount of CPUs. Native scenario only explores one CPU per Hanoi or Bitcount employed.

It is possible to see that the Bitcount execution suffers more interference from the amount of CPUs employed, with a single CPU being the worst case scenario and eight CPUs being the best-case scenario. The Hanoi execution time is also clearly affected by the increase of Bitcounts in the system, although the processing time itself is affected by the amount of Hanois in the system

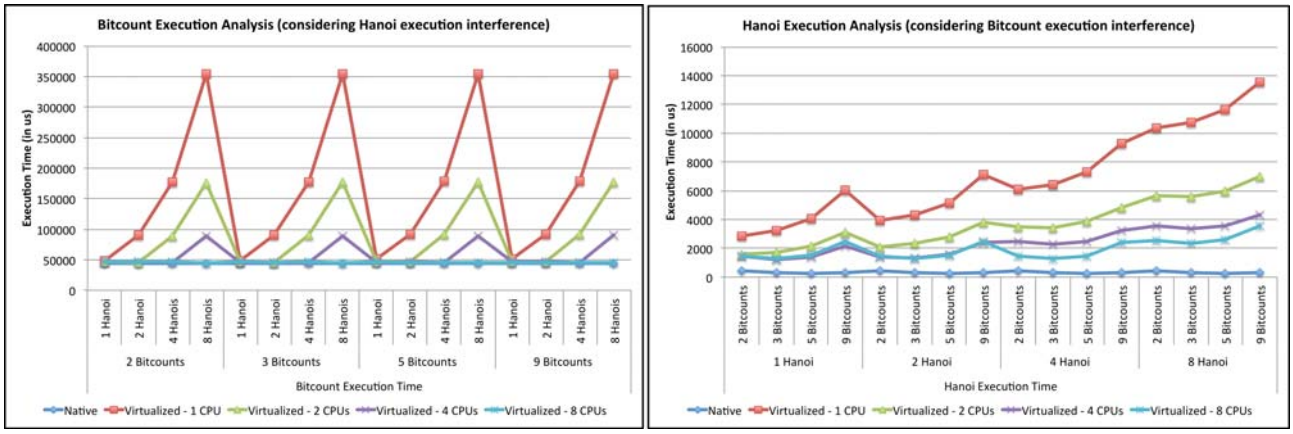


Figure 4.35 – Mixed Scenario With Bitcount and Hanoi application's interference in each other's execution time

(more VCPUs per CPU ratio) and by the time spent by the hypervisor treating the privileged communication.

### 4.3.14 Discussion

These results were taken aiming to investigate the behavior of the proposed platform. We use a hardware-assisted virtualization based in the trap-and-emulate technique. Therefore, we were able to achieve low execution overheads depending on the VCPU per CPU ratio. Obviously, if we are using a single CPU to execute two vCPUs, we'd expect the execution time of the application running on the vCPU to double, at least. However, the virtualized approach would need half the area occupied by the non-virtualized approach, and that's a tradeoff that needs to be analysed. Also, if we are talking about applications that could share a physical CPU (non-prohibitive overhead) but could not coexist due to security reasons, the strong separation between Application Domains could be a solution.

Our platform allows inter-domain communication both in mono- and multi-processed environments. For that, each Application Domain uses a network peripheral, which accesses need to be treated by the hypervisor. The hypervisor is the module responsible for performing the actual communication, by accessing the memory and copying the desired data. Therefore, tests with communication present higher overhead. We believe that adding extra hardware support, such as extra execution modes (like virtualization mode, proposed by Intel), duplicating certain structures (such as the register bank, to decrease context-switch overhead), and an improving lock system (to provide synchronization) can dramatically decrease these overheads.

Finally, by analysing these results we believe our platform is suitable for monoprocessed and small multiprocessed environments with less than 8 CPUs (due to the shared memory strategy), where applications with high idle time coexist with medium-intensive processing applications. This

scenario would benefit from the consolidation offered by virtualization with acceptable penalty due to the low amount of emulations.

# 5. FINAL CONSIDERATIONS

Virtualization has been usually adopted by enterprise market to better enjoy the multiprocessors' computing power. Meanwhile, embedded systems used to be extremely restricted systems. However, their current multiple functionalities lead to a non-linear growth of the software complexity. In this context, many solutions are being studied, like virtualization. Mainly, the advantages of embedded virtualization are: (i) to allow several OSs (RTOS and user OSs) to run in the same processor; (ii) to reduce the manufacturing cost; (iii) to improve security and reliability; and (iv) to decrease ES software development complexity.

However, despite these several advantages, the implementation of the virtualization technique can bring undesired overheads. To cope with that, most embedded systems that counted on virtualization facilities opted for using a paravirtualization approach. Paravirtualization uses the concept of hypercalls that must be inserted in each guest OS so that no unnecessary traps are performed.

Indeed, paravirtualization has the ability of decreasing virtualization overheads: at the cost of performing modification in the guest OS source code. These modification are not always simple or trivial. They imply in greater engineering costs and can affect the the product's time to market.

Considering that, this research has proposed the use of paravirtualization in the first attempt of the virtualization model. In that first attempt we also proposed the use of virtualization to provide cluster-based MPSoCs with application specialization and less area consumption. Nevertheless, analyzing the potential drawbacks of paravirtualization, we decided to investigate approaches based in virtualization with hardware support.

In the second virtualization model attempt we decided to adopt a hardware virtualization support. Thus, we decided to modify a MIPS-based processor to allow some virtualization support. Our approach is a mixed version of hardware support and trap-and-emulate. We are aware of the limitations trap-and-emulate can bring as we discuss them later. The first processor we modified to support virtualization was a Plasma core, a MIPS I processor. We managed to successfully implement virtualization and focused on multiprocessor and real-time support.

It is important to state that our multiprocessor support was guided by the cluster-based MPSoC approach proposed in the first attempt. So, we use a bus-based communication system that can be integrated in a NoC for cluster-based architectures in the future. Still, the real-time support is a simple one, where real-time applications are delegated directly to the hypervisor that uses a mixed scheduling approach of priority-based and EDF algorithms.

After defining a virtualization model we implemented the hardware support in a MIPS II processor. By that time, we changed the simulation strategy by using a higher level simulation tool. We added virtualization support to the MIPS 4Kc core and simulated other applications thanks to the new simulation strategy. This architecture was also implemented to support multiprocessed systems.

The evolution of the virtualization model and this research itself investigated the possibility of providing hardware support to virtualization in simpler cores. Multiprocessor and real-time support were also main points of concern. However, during the final stages of this research, MIPS has officially announced the guidelines for hardware support to full virtualization in its architectures. This support has some similarities with ours especially regarding to memory division and management. The main differences is that the guidelines are focused on MIPS V processors and that it focuses on full virtualization where no emulations of privileged instructions are needed.

Therefore, our research has demonstrated its validity since MIPS itself defined hardware support to its architecture, enabling better virtualization use. It is important to notice that, by the time this dissertation was written, there were no commercial solutions implementing the virtualization guidelines for MIPS processors.

Finally, this research has achieved its main goals by providing successfully an investigation concerning the use of virtualization techniques in multiprocessed architectures of embedded systems. There were three main attempts of a virtualization model, as from the first one we used mainly the cluster-based idea, that guided the following models. The second attempt introduced the support to real-time and the third attempt was able to reunite these characteristics in a more complex processor than those used before.

## 5.1 Revisiting Research Questions

In the beginning of this dissertation we proposed two main questions to be addressed throughout the research development. The first question, *What is the impact of the architecture diversity typically found in ESs in employing the virtualization technique?* could be answered by exploring related virtualization support in embedded architectures. It was possible to see some similarities among each architecture's strategies as well as some differences, since it must be adapted to each architecture profile.

The second question, *Considering the existing virtualization implementation approaches in general-purpose computers, which one is more suitable for a given embedded environment?* we concluded that hardware-based approaches tend to be more successful, since they offer greater performance and flexibility when represent an extension of a given architecture. Still, paravirtualization is a suitable solution where there is no possibility of adapting the hardware itself.

## 5.2 Research publications

This research has successfully contributed in several aspects with the field of virtualization in embedded systems. We proposed multiprocessor and real-time support for MIPS-based approaches and successfully adapted two different cores for virtualization.

Still, since this research is inserted in a broader research group, we had two main kinds of publications: direct-related, which discuss about virtualization techniques, and; indirect-related, which discuss improvements of the Hellfire framework and other design strategies.

List of direct-related (D) and indirect-related (I) publications[1]:

1. (D) Aguiar, A.; Hessel, F., "Embedded systems' virtualization: The next challenge?," *Rapid System Prototyping (RSP), 2010 21st IEEE International Symposium on* , vol., no., pp.1,7, 8-11 June 2010 doi: 10.1109/RSP.2010.5656430

2. (I) Aguiar, A.; Filho, S.J.; Magalhaes, F.G.; Casagrande, T.D.; Hessel, F., *"Hellfire: A design framework for critical embedded systems' applications," Quality Electronic Design (ISQED)*, 2010 11th International Symposium on , vol., no., pp.730,737, 22-24 March 2010 doi: 10.1109/ISQED.2010.5450495

3. (D) Aguiar, A. C. P. ; Hessel, Fabiano . Virtualização em sistemas embarcados: é o futuro?. *In: I Workshop de Sistemas Embarcados - WSE, 2010, Gramado - RS. I Workshop de Sistemas Embarcados*. Porto Alegre, 2010. p. 17-28.

4. (D) Aguiar, A. C. P. ; Hessel, Fabiano . Adapting Embedded Systems' Framework to Provide Virtualization: the Hellfire Case Study. In: *Conferência Brasileira de Sistemas Embarcados Críticos*, 2011, São Carlos - SP. 1a Conferência Brasileira de Sistemas Embarcados Críticos, 2011.

5. (D) Aguiar, A.; de Magalhaes, F.G.; Hessel, F., "Embedded virtualization for the next generation of cluster-based MPSoCs," *Rapid System Prototyping (RSP), 2011 22nd IEEE International Symposium on* , vol., no., pp.113,119, 24-27 May 2011 doi: 10.1109/RSP.2011.5929984

6. (I) Aguiar, A. ; Johann Filho, Sergio ; Magalhaes, F. ; Hessel, F. . Introdução ao Desenvolvimento de Software Embarcado. In: Alberto Souza, Wagner Meira. (Org.). *Jornadas de Atualizacao em Informatica*. 1ed.Rio de Janeiro: PUC Rio, 2011, v. 1, p. 109-158.

7. (D) Aguiar, A.; Hessel, F., "Virtual Hellfire Hypervisor: Extending Hellfire Framework for embedded virtualization support," *Quality Electronic Design (ISQED), 2011 12th International Symposium on* , vol., no., pp.1,8, 14-16 March 2011 doi: 10.1109/ISQED.2011.5770725

8. (D) Aguiar, A.; Hessel, F., Tutorial on Embedded Virtualization: Embedded systems? Virtualization: Concepts, Issues and Challenges, *Quality Electronic Design (ISQED), 2012 13th International Symposium on* , 2012.

9. (I) Antunes, E.; Soares, M.; Aguiar, A.; Johann, F.S.; Sartori, M.; Hessel, F.; Marcon, C., "Partitioning and dynamic mapping evaluation for energy consumption minimization on NoC-based MPSoC," *Quality Electronic Design (ISQED), 2012 13th International Symposium on* , vol., no., pp.451,457, 19-21 March 2012 doi: 10.1109/ISQED.2012.6187532

[1] Currently there are two conference papers in review and one article in minor review.

10. (I) Magalhaes, F.G.; Longhi, O.; Filho, S.J.; Aguiar, A.; Hessel, F., "NoC-based platform for embedded software design: An extension of the Hellfire Framework," *Quality Electronic Design (ISQED), 2012 13th International Symposium on* , vol., no., pp.97,102, 19-21 March 2012 doi: 10.1109/ISQED.2012.6187480

11. (I) Aguiar, A.; Hessel, F., "Exploring embedded software concepts using the hellfire platform in an undergraduate course," *Interdisciplinary Engineering Design Education Conference (IEDEC),* 2012 2nd , vol., no., pp.96,99, 19-19 March 2012 doi: 10.1109/IEDEC.2012.6186931

12. (D) Alexandra Aguiar and Fabiano Hessel. 2012. Current Techniques and Future Trends in ES's Virtualization. *Software Practice and Experience* 42, 7 (July 2012), 917-944. DOI = 10.1002 / spe.1156 http://dx.doi.org/10.1002/spe.1156

13. (D) Aguiar, A.; Moratelli, C.; Sartori, M.L.L.; Hessel, F., "Hardware-assisted virtualization targeting MIPS-based SoCs," *Rapid System Prototyping (RSP), 2012 23rd IEEE International Symposium on* , vol., no., pp.2,8, 11-12 Oct. 2012 doi: 10.1109/RSP.2012.6380683

14. (I) Filho, S.J.; Aguiar, A.; de Magalhaes, F.G.; Longhi, O.; Hessel, F., "Task model suitable for dynamic load balancing of real-time applications in NoC-based MPSoCs," *Computer Design (ICCD), 2012 IEEE 30th International Conference on* , vol., no., pp.49,54, Sept. 30 2012-Oct. 3 2012 doi: 10.1109/ICCD.2012.6378616

15. (D) Aguiar, A.; Moratelli, C.; Sartori, M.L.L.; Hessel, F., "A virtualization approach for MIPS-based MPSoCs," *Quality Electronic Design (ISQED), 2013 14th International Symposium on* , vol., no., pp.611,618, 4-6 March 2013 doi: 10.1109/ISQED.2013.6523674

16. (I) Aguiar, A.; Filho, S.J.; Magalhaes, F.G..; Hessel, F., "Customizable RTOS to support communication infrastructures and to improve design space exploration in MPSoCs" to appear in *Rapid System Prototyping (RSP), 2013 24rd IEEE International Symposium on* , 2013

17. (I) Alexandra Aguiar, Sergio Johann Filho, Felipe Magalhaes, and Fabiano Hessel. 2013. Communication support at the OS level to enhance design space exploration in multiprocessed embedded systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*. ACM, New York, NY, USA, 1555-1556. DOI=10.1145/2480362.2480652 http://doi.acm.org/10.1145/2480362.2480652

## 5.3 Limitations, ongoing and future work

Our approach has some limitations. The trap-and-emulate strategy was used since the hardware modification we made were mainly focused on memory separation of virtual machines to provide security. We did not add a hypervisor execution mode and therefore privileged instructions needed to be emulated. The modifications needed to add another execution mode vary with each architecture.

Currently, an ongoing work of the research group consists in studying the MIPS virtualization support guidelines so that we can implement them on a processor and, subsequently, implement a hypervisor that offers full virtualization without trapping privileged instructions from the guest OSs.

Also, we concluded that one of the bottlenecks is the hypervisor initialization. Thus, some efforts are being spent to optimize it. Still, we are working to provide dynamic memory allocation for the domains, since they are currently set statically.

Regarding real-time support, we intend to use the model's flexibility and create separate domains and VCPUs with specific optimizations. Our currently approach combines full virtualization and native execution but specific paravirtualization-based domains are still considered to provide real-time. Still, more real-time algorithms must be implemented, since we now have a round-robin based scheduling combined with EDF. Currently, only monoprocessed virtual domains are available due to limitations regarding the guest OS we are using (Hellfire Lite OS [AFM+10]).

Finally, concerning multiprocessed environments we intend to implement the cluster-based architecture with virtualization and compare it to another cluster-based platform that does not use virtualization. This other platform, without virtualization, has already been developed in the research group and will be used for further comparisons.

# BIBLIOGRAPHY

[AA06] Adams, K.; Agesen, O. "A comparison of software and hardware techniques for x86 virtualization". In: ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, 2006, pp. 2–13.

[ABK09] Acharya, A.; Buford, J.; Krishnaswamy, V. "Phone virtualization using a microkemel hypervisor". In: Proceedings of the 3rd IEEE international conference on Internet multimedia services architecture and applications, 2009, pp. 205–210.

[AdMH11] Aguiar, A.; de Magalhaes, F.; Hessel, F. "Embedded virtualization for the next generation of cluster-based mpsocs". In: Rapid System Prototyping (RSP), 2011 22nd IEEE International Symposium on, 2011, pp. 113–119.

[AFM+10] Aguiar, A.; Filho, S.; Magalhaes, F.; Casagrande, T.; Hessel, F. "Hellfire: A design framework for critical embedded systems' applications". In: Quality Electronic Design (ISQED), 2010 11th International Symposium on, 2010, pp. 730 –737.

[AFNK11] Asberg, M.; Forsberg, N.; Nolte, T.; Kato, S. "Towards real-time scheduling of virtual machines without kernel modifications". In: Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on, 2011, pp. 1–4.

[AG09] Armand, F.; Gien, M. "A Practical Look at Micro-Kernels and Virtual Machine Monitors". In: Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE, 2009, pp. 1–7.

[Arc13] Architecture, A. "ARM Virtualization Extensions Architecture Specification". Captured in: http://www.arm.com/products/processors/technologies/virtualization-extensions.php, July 2013.

[BABP06] Bertozzi, S.; Acquaviva, A.; Bertozzi, D.; Poggiali, A. "Supporting task migration in multi-processor systems-on-chip: A feasibility study". In: Design, Automation and Test in Europe, 2006. DATE '06. Proceedings, 2006, pp. 1–6.

[BDB+08] Brakensiek, J.; Dröge, A.; Botteck, M.; Härtig, H.; Lackorzynski, A. "Virtualization as an enabler for security in mobile devices". In: Proceedings of the 1st workshop on Isolation and integration in embedded systems, 2008, pp. 17–22.

[BGDB10] Bertin, C.; Guillon, C.; De Bosschere, K. "Compilation and virtualization in the hipeac vision". In: Design Automation Conference (DAC), 2010 47th ACM/IEEE, 2010, pp. 96–101.

[CB09] Cereia, M.; Bertolotti, I. C. "Virtual machines for distributed real-time systems", *Computer Standards & Interfaces*, vol. 31–1, jan 2009, pp. 30–39.

[CG05] Choudhuri, S.; Givargis, T. "Software virtual memory management for MMU-less embedded systems", Technical Report, University of California, 2005.

[Cor13] Cores, O. "Plasma most MIPS I(TM) opcodes". Captured in: http://www.opencores.org.uk/projects.cgi/web/mips/, August 2013.

[CR10] Cohen, A.; Rohou, E. "Processor virtualization and split compilation for heterogeneous multicore embedded systems". In: Design Automation Conference (DAC), 2010 47th ACM/IEEE, 2010, pp. 102–107.

[CRM10] 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.

[EHMAZAR05] El-Haj-Mahmoud, A.; AL-Zawawi, A. S.; Anantaraman, A.; Rotenberg, E. "Virtual multiprocessor: an analyzable, high-performance architecture for real-time computing". In: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems, 2005, pp. 213–224.

[For10] Fornaeus, J. "Device hypervisors". In: Proceedings of the 47th Design Automation Conference, DAC, 2010, pp. 114–119.

[GIZG+09] Geng, L.-F.; li Zhang, D.; Gao, M.-L.; Chen, Y.-C.; Du, G.-M. "Prototype design of cluster-based homogeneous multiprocessor system-on-chip". In: Anti-counterfeiting, Security, and Identification in Communication, 2009. ASID 2009. 3rd International Conference on, 2009, pp. 311–315.

[Hay77] Hayes, P. "A note on the towers of hanoi problem", *The Computer Journal*, vol. 20–1, 1977, pp. 282–285.

[HbSH+08] Hwang, J.-Y.; bum Suh, S.; Heo, S.-K.; Park, C.-J.; Ryu, J.-M.; Park, S.-Y.; Kim, C.-R. "Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones". In: Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE, 2008, pp. 257–261.

[Hei08] Heiser, G. "The role of virtualization in embedded systems". In: Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems, 2008, pp. 11–16.

[Hei09] Heiser, G. "Hypervisors for consumer electronics". In: Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference, 2009, pp. 614–618.

[Hei11] Heiser, G. "Virtualizing embedded systems: why bother?" In: Proceedings of the 48th Design Automation Conference, 2011, pp. 901–905.

[Her09] Hermeling, M. "The key differences between enterprise and embedded virtualisation". Captured in: http://ece-news.stc-d.de/custom/enews220709.htm, Feb 2009.

[HNN05] Hansson, H.; Nolin, M.; Nolte, T. "Real-time in embedded systems". In: *Embedded Systems Handbook*, Zurawski, R. (Editor), CRC press, 2005, chap. 2.

[HPHS04] Hohmuth, M.; Peter, M.; Härtig, H.; Shapiro, J. S. "Reducing tcb size by using untrusted components: small kernels versus virtual-machine monitors". In: EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop, 2004, pp. 22.

[HT94] Hesselink, W. H.; Tol, R. M. "Formal feasibility conditions for earliest deadline first scheduling", Technical Report, University of California, 1994.

[Int11] Intel Architecture Group. "Virtualization Technology Specification". Captured in: http://www.intel.com/p/en_US/embedded/hwsw/technology/virtualization, Dec 2011.

[IO07] Ito, M.; Oikawa, S. "Mesovirtualization: lightweight virtualization technique for embedded systems". In: Proceedings of the 5th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems, 2007, pp. 496–505.

[ISE08] Inoue, H.; Sakai, J.; Edahiro, M. "Processor virtualization for secure mobile terminals", *Transactions on Design Automation of Electronic Systems (TODAES*, vol. 13–3, 2008.

[JSZ10] Jin, X.; Song, Y.; Zhang, D. "FPGA prototype design of the computation nodes in a cluster based MPSoC". In: Anti-Counterfeiting Security and Identification in Communication (ASID), 2010 International Conference on, 2010, pp. 71–74.

[JTW05] Jerraya, A.; Tenhunen, H.; Wolf, W. "Multiprocessor systems-on-chips", *Computer*, vol. 38–Issue 7, July 2005, pp. 36– 40.

[KW08] Kleidermacher, D.; Wolf, M. "Mils virtualization for integrated modular avionics". In: Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th, 2008, pp. 3–8.

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

[LbSC10] Lee, S.-M.; bum Suh, S.; Choi, J.-D. "Fine-grained I/O access control based on xen virtualization for 3G/4G mobile devices". In: Design Automation Conference (DAC), 2010 47th ACM/IEEE, 2010, pp. 108–113.

[LHC11] Lee, Y.-C.; Hsueh, C.-W.; Chang, R.-G. "Inline emulation for paravirtualization environment on embedded systems". In: Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on, 2011, pp. 388–392.

[LKM+12] Li, N.; Kinebuchi, Y.; Mitake, H.; Shimada, H.; Lin, T.-H.; Nakajima, T. "A light-weighted virtualization layer for multicore processor-based rich functional embedded systems". In: Proceedings of the 2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, 2012, pp. 144–153.

[LP05] Lavagno, L.; Passerone, C. "Design of embedded systems". In: *Embedded Systems Handbook*, Zurawski, R. (Editor), CRC press, 2005, chap. 3, pp. 52–102.

[MBC+09] Moore, R. W.; Baiocchi, J. A.; Childers, B. R.; Davidson, J. W.; Hiser, J. D. "Addressing the challenges of dbt for the arm architecture". In: Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, 2009, pp. 147–156.

[MCM+04] Moraes, F.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. "Hermes: An infrastructure for low area overhead packet-switching networks on chip", *Integr. VLSI J.*, vol. 38–1, Oct 2004, pp. 69–93.

[Moy13] Moyer, B. "Real World Multicore Embedded Systems". Newnes, 2013, 648p.

[NAMV05] Nollet, V.; Avasare, P.; Mignolet, J.-Y.; Verkest, D. "Low cost task migration initiation in a heterogeneous mp-soc". In: DATE '05: Proceedings of the conference on Design, Automation and Test in Europe, 2005, pp. 252–253.

[NKS+11] Nakajima, T.; Kinebuchi, Y.; Shimada, H.; Courbot, A.; Lin, T.-H. "Temporal and spatial isolation in a virtualization layer for multi-core processor based information appliances". In: Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific, 2011, pp. 645–652.

[Noe05] Noergaard, T. "Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers". Newnes, 2005, 656p.

[org12] org, X. "Embedded Xen Project." Captured in: http://www.xen.org/community/projects.html, Aug 2012.

[OVI+12] Ost, L.; Varyani, S.; Indrusiak, L. S.; Mandelli, M.; Almeida, G. M.; Wachter, E.; Moraes, F.; Sassatelli, G. "Enabling adaptive techniques in heterogeneous mpsocs based on virtualization", *ACM Trans. Reconfigurable Technol. Syst.*, vol. 5–3, Oct 2012, pp. 17:1–17:11.

[OVP13] OVP, O. "Open virtual platforms". Captured in: http://www.ovpworld.org/, June 2013.

[Pa09] Park, M.; al, e. "Real-time Operating System Virtualization for Xen-Arm". In: International Symposium on Embedded Systems, 2009, pp. 60–66.

[PG74] Popek, G. J.; Goldberg, R. P. "Formal requirements for virtualizable third generation architectures", *Communications ACM*, vol. 17–7, 1974, pp. 412–421.

[Pow12] Power Organization Group. "PowerISA v2.06 - Virtualization support". Captured in: https://www.power.org/resources/downloads/virtualization_for_Embedded_Power_Architecture.pdf, Feb 2012.

[RAN10] Rivas, R.; Arefin, A.; Nahrstedt, K. "Janus: A cross-layer soft real-time architecture for virtualization". In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, 2010, pp. 676–683.

[Riv13] River, W. "Wind River". Captured in: http://www.windriver.com/, Aug 2013.

[RKKE10] Ryu, E.; Kim, I.; Kim, J.; Eom, Y. I. "Myav: An all-round virtual machine monitor for mobile environments". In: Industrial Informatics (INDIN), 2010 8th IEEE International Conference on, 2010, pp. 657–662.

[Ros04] Rosenblum, M. "The reincarnation of virtual machines", *Queue*, vol. 2–5, 2004, pp. 34–40.

[Rud09] Rudolph, L. "A virtualization infrastructure that supports pervasive computing", *Pervasive Computing, IEEE*, vol. 8–4, oct 2009, pp. 8 –13.

[SCH+09] Su, D.; Chen, W.; Huang, W.; Shan, H.; Jiang, Y. "Smartvisor: Towards an efficient and compatible virtualization platform for embedded system". In: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems, 2009, pp. 37–41.

[SK10] Steinberg, U.; Kauer, B. "Nova: A microhypervisor-based secure virtualization architecture". In: Proceedings of the 5th European Conference on Computer Systems, 2010, pp. 209–222.

[SLB07] Stoess, J.; Lang, C.; Bellosa, F. "Energy management for hypervisor-based virtual machines". In: ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference, 2007, pp. 1–14.

[SP09] Shen, H.; Petrot, F. "Novel task migration framework on configurable heterogeneous mpsoc platforms". In: Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific, 2009, pp. 733–738.

[SPKG10] Srinivasan, V.; Parihar, N.; Khurana, V.; Gavrilovska, A. "A split driver approach to soc virtualization - challenges and opportunities". In: Parallel Processing Workshops (ICPPW), 2010 39th International Conference on, 2010, pp. 50–57.

[Sub09] Subar, S. "Virtualisation to enable next billion devices". Captured in: http://www.embeddeddesignindia.co.in/ART_8800576093_2800003_TA_7cb7532e.HTM, Feb 2009.

[Tra13] Trango. "Trango Hypervisor". Captured in: http://www.trango.com/, Aug 2013.

[VLX13] VLX, V. "Virtuallogic Project." Captured in: http://http://www.virtuallogix.com, Aug 2013.

[Wal02] Waldspurger, C. A. "Memory resource management in VMware ESX server", *SIGOPS Oper. Syst. Rev.*, vol. 36–SI, Dec 2002, pp. 181–194.

[WCC+08] Walters, J. P.; Chaudhary, V.; Cha, M.; Jr., S. G.; Gallo, S. "A comparison of virtualization technologies for HPC". In: Proceedings of the 22nd International Conference on Advanced Information Networking and Applications, 2008, pp. 861–868.

[Wol03] Wolf, W. "A decade of hardware/software codesign", *Computer*, vol. 36–4, April 2003, pp. 38–43.

[WZS+10] Wang, Y.; Zhang, J.; Shang, L.; Long, X.; Jin, H. "Research of real-time task in xen virtualization environment". In: Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on, 2010, pp. 496–500.

[XWLG11] Xi, S.; Wilson, J.; Lu, C.; Gill, C. "Rt-xen: Towards real-time hypervisor scheduling in xen". In: Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on, 2011, pp. 39–48.

[YKP+11] Yang, J.; Kim, H.; Park, S.; Hong, C.; Shin, I. "Implementation of compositional scheduling framework on virtualization", *SIGBED Review*, vol. 8–1, 2011.

[YKY10] Yoo, S.; Kim, Y.-P.; Yoo, C. "Real-time scheduling in a virtualized ce device". In: Consumer Electronics (ICCE), 2010 Digest of Technical Papers International Conference on, 2010, pp. 261–262.

[YLH+08] Yoo, S.; Liu, Y.; Hong, C.-H.; Yoo, C.; Zhang, Y. "Mobivmm: A virtual machine monitor for mobile phones". In: Proceedings of the First Workshop on Virtualization in Mobile Computing, 2008, pp. 1–5.

[YNK01] Yuan, W.; Nahrstedt, K.; Kim, K. “R-edf: A reservation-based edf scheduling algorithm for multiple multimedia task classes”. In: Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS ’01), 2001, pp. 149–200.

[ZM00] Zorian, Y.; Marinissen, E. “System chip test: how will it impact your design?” In: Design Automation Conference, 2000, DAC, 2000, pp. 136–141.