Evaluation of a multiple criticality real-time virtual machine system and configuration of an RTOS's resources allocation techniques

Mohamed El Mehdi Aichouch

Thèse

THESE INSA Rennes
sous le sceau de l'Université européenne de Bretagne
pour obtenir le titre de
DOCTEUR DE L'INSA DE RENNES
Spécialité : Electronique et Télécommunication

présentée par
Mohamed El Mehdi Aichouch
ECOLE DOCTORALE : Matisse
LABORATOIRE : IETR

Evaluation of a Multiple Criticality Rea-Time Virtual Machine System and Configuration of an RTOS's Resources Allocation Techniques

Thèse soutenue le 28.05.2014
devant le jury composé de :

Isabelle Puaut
Professeur à l'université de Rennes 1 / *Président*
Laurent Pautet
Professeur à Télécom Paris-Tech / *rapporteur*
François Verdier
Professeur à l'université de Nice-Sophia Antipolis / *rapporteur*
Jean-Luc Béchennec
Chargé de recherche à l'IRRCyN CNRS UMR 6597 à Nantes / *examinateur*
Jean-Christophe Prévotet
Maître de conférence à l'INSA de Rennes / Co-encadrant de thèse
Fabienne Nouvel
Maître de conférence à l'INSA de Rennes / Directeur de thèse

Evaluation of a Multiple-Criticality Real-Time Virtual Machine System and Configuration of an RTOS's Resources Allocation Techniques.

Mohamed El Mehdi Aichouch

A dissertation submitted to the faculty of the INSA de Rennes in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electronic and Telecommunication.

INSA de Rennes
2014

Approved by:

Isabelle Puaut

Jean-Luc Béchennec

Laurent Pautet

François Verdier

Jean-Christophe Prévotet

Fabienne Nouvel

ABSTRACT

Mohamed El Mehdi Aichouch
Evaluation of a Multiple-Criticality Real-Time Virtual Machine System
and Configuration of an RTOS's Resources Allocation Techniques.

In the domain of server and mainframe systems, virtualizing a computing system's physical resources to achieve improved sharing and utilization has been well established for decades. Full virtualization of all system resources, including processor, memory and I/O devices makes it possible to run multiple operating systems on a single physical platform. Recently, the availability of full virtualization on physical platforms that target embedded systems creates new uses cases in the domain of real-time embedded systems. In a non virtualized system, a single OS controls all hardware platform resources. A virtualized system includes a new layer of software, the virtual machine monitor (VMM). The VMM's principal role is to arbitrate accesses to the underlying physical host platform's resources so that multiple operating systems can share them. The VMM presents to each OS a set of virtual platform interfaces that constitute a virtual machine.

Given the existence of a multitude of VMMs that have been proved efficient in the domain of server and mainframe systems, there is a trend to reuse the existing work. However, there is a difference in the performance metric required by these two domains.

In this dissertation we use an existing VMM to evaluate the performance of a real-time operating system. We observed that the virtual machine monitor affects the internal overheads and latencies of the guest operating system. This observation led us to conduct further investigation in order to answer the following question: what are the hardware mechanisms and software implementations that could prevent the system from meeting its deadlines and guaranteeing its real-time constraints?

Our analysis revealed that hardware mechanisms that allow a VMM to provide an efficient way to virtualize the memory management unit, and the device interrupts, are necessary to limit the overhead of the virtualization on real-time systems. More importantly, the scheduling of virtual

machines by the VMM is essential to guarantee the temporal constraints of the system and have to be configured carefully.

In a second work, and starting from a previous project aiming at helping a system designer to explore a software-hardware co-design of a solution using high-level simulation models, we proposed a methodology that allow the transformation of a simulation model into an executable program on a real hardware. The idea is to provide the system designer with the necessary tools to rapidly explore the design space and validate it, and then to generate a configuration that could be used directly on top of a real hardware.

We used a model-driven engineering approach to perform a model-to-model transformation to convert the simulation model into an executable model. And we used a middleware able to support a variety of resources allocation techniques in order to implement the configuration previously selected by the system designer during the simulation phase. We proposed a prototype that implements our methodology and validate our concepts. The results of the experiments confirmed the viability of the approach.

To my parents, Abdelhamid and Kalthoum.

ACKNOWLEDGEMENTS

I would like to thank my advisors, Fabienne Nouvel and Jean-Christophe Prévotet for their unwavering support, for their help, and their precious advices.

I would like to express my thanks and appreciation to all the members of my committe, Professor Isabelle Puaut, Professor François Verdier, Professor Laurent Pautet, and Doctor Jean-Luc Béchennec for their guidance and advice. Your acceptance to be present in my committe is great honor for me.

I would also like to express my thanks and appreciation to all my colleagues. Very special thanks to Yaset Oliva, Yvan Kokar, Thierry Dubois, Tony Makdissy, Nicolas Cornillet, Jordan Lorandel, Philippe Tanguy, Simon Mener, Vincent Callec, Abdallah Hamini, Ahmed Jaban, Saber Dakhli, Imen Ben Trad, Rida El Chall, Hiba Bawab, Hua Fu, Jean-Christophe Sibel, Ming Liu, Hui Ji, Linning Peng, Tian Xia, Ali Cheaito, Mohamed Maaz, Roua Youssef, Hussein Kudoh, Hanna Farhat, Bachir Habib, Georges Da Silva, Sofiane Chaabane, Morad Larbi, and Abdul Fall. — thank y'all for your kindness and the great moment we shared in our beautiful work place.

I am deeply thankful to my colleague and friend Yaset Oliva for reviewing my research papers and giving me many precious adivces. Thank you very much for the interesting discussions, and for your help when I came to Rennes.

I would like also to thank all the professors, research scientists, and technical staff at the Institut d'Électronique et de Télécommunication de Rennes. I would like also to thank all the SRC department staff at the INSA de Rennes.

I am deeply thankful to my professor Benoît Miramond, thank you for inviting me to join your research team and for your encouragement that motivated me to continue in this scientific research field.

I am deeply thankful to my friend Mac Mollisson, from the real-time system group at the University of North Carolina at Chappel Hill, thank you very much for your support regarding the library, and thanks a lot for the great discussions and feedback regarding my research work.

Foremost, I am greatly indebted to my parents Abdelhamid and Kalthoum for their unwavering support, understanding, and encouragement, both during my research study and before.

I am also greatly indebted to my sister Nesrine and my brother-in-law Aimed, for their love and continuous support, for their help, and the wonderful times we spent together. I am also grateful to all my family members and friends. Thank you very much for your generosity and friendship. I could not finished this without you — thank y'all for your trust that this was indeed the right way.

TABLE OF CONTENTS

LIST OF TABLES xiii

LIST OF FIGURES xiv

LIST OF ABBREVIATIONS xvii

1 Introduction 1

2 Related Work 3

2.1 Real-Time OS alongside General-Purpose OS 4

2.1.1 Dual-Kernel Design 5

2.1.2 Native Real-Time Linux 6

2.2 Virtual Machine Systems 7

2.3 Real-Time Virtual Machine Systems 9

2.3.1 Linux Kernel-based Virtual Machine 9

2.3.2 Microkernel Support for Virtualization 13

2.3.2.1 OKL^4 microvisor 14

2.3.2.2 Nova microhypervisor 15

2.3.2.3 L^{4}Fiasco microkernel 16

2.3.3 Xen 19

2.3.4 RT-Xen 23

2.3.5 Real-Time Xen-ARM 30

2.3.6 Virtualization for safety-critical system 32

2.4 RTOS Configuration 40

2.4.1 Composite 40

2.4.2 ExSched 42
2.4.3 LITMUS^{RT} 44
2.4.4 Microkernel 46
2.4.5 OveRSoC RTOS Model 47

3 Virtualization and Real-Time Systems 50
3.1 Hardware-Assisted Virtualization 50
3.1.1 Resource Virtualization - Processors 51
3.1.1.1 Conditions for ISA Virtualization 51
3.1.1.2 Intel Virtualization Extension 52
3.1.1.3 ARM Virtualization Extension 53
3.2 Linux Kernel Virtual Machine 54
3.2.1 Qemu 55
3.2.2 Virtual Machine Process 55
3.3 Scheduling Latency Evaluation 57
3.4 Fine-Grained Overheads and Latencies Evaluation 62
3.4.1 Overheads and Latencies 62
3.4.2 Hardware platform 63
3.4.3 LITMUS^{RT} and Feather Trace toolkit 64
3.4.4 Synthetic Workloads 66
3.5 Results 67
3.6 Emulation of the I/O interrupts 74
3.6.1 Comparison with ARM I/O virtualization 75
3.6.2 Comparison with Custom ARM Hardware Architecture 77
3.7 Summary 77

4 Real-Time Scheduling of Virtual Machines 79
4.1 Real-Time Task Model 79
4.1.1 Temporal Correctness 80

4.1.2 Schedulability Test 81
4.2 Real-Time Scheduling 81
4.2.1 Fixed-Priority Scheduling 82
4.2.2 Dynamic-Priority Scheduling 84
4.3 Algorithmic Analysis 85
4.4 Computing of the Efficient Scheduling Parameters 87
4.4.1 Execution Length of a Virtual Machine 88
4.4.2 Schedulability Condition on a VM 89
4.4.3 Computing of the Highest-Priority VM's Parameters 92
4.5 Overhead-aware Schedulability Analysis 96
4.6 Empirical Evaluation 98
4.7 Summary 103
5 RTOS Models Transformation and Configuration 105
5.1 Software/Hardware Co-design Process 105
5.2 OveRSoC Methodology 106
5.2.1 From Simulation Models to Executable Models 111
5.3 Model Driven Engineering 112
5.3.1 Model Driven Architecture 113
5.3.2 Domain Specific Language 114
5.4 RTOS-specific Modeling Language 115
5.4.1 RTOS Meta-Model 115
5.4.2 Concrete Syntax 117
5.4.3 Model-to-Code Transformation 118
5.4.4 Test of the Transformation 120
5.4.5 Model-To-Model Transformation 120
5.4.6 Limitation of the Approach 121
5.5 Summary 122

6 RTOS Configuration using User-Level Library 123
6.1 User-Level Scheduling on top of Microkernel 125
6.2 Tasks Model and Thread Mechanisms 126
6.2.1 Sporadic Task Model 126
6.2.2 Thread library 126
6.3 Nova Microkernel and Runtime Environment 128
6.4 Library Implementation 129
6.5 Experiments 132
6.5.1 Overheads and Latencies 132
6.5.2 Experiment Setup 132
6.5.3 Experimental Workloads and Execution Trace 132
6.5.4 Measured Results 134
6.5.5 Comparison with Similar Approaches 136
6.6 Summary 138
7 Conclusions 139
7.1 Open Question and Future Works 141
8 Résumé de la thèse 142
8.1 Etat de l'art sur la virtualisation 143
8.1.1 Linux Kernel Virtual Machine 144
8.1.2 Virtualisation basée sur le Micro-noyau 145
8.1.3 Xen 146
8.1.4 Virtualisation pour les systèmes critiques 149
8.2 Etat de l'art sur la configuration des systèmes d'exploitation 150
8.3 Impact de la Virtualisation sur les Systèmes Temps-Réel 152
8.4 Ordonnancement Temps-Réel des Machines Virtuelles 152
8.5 Transformation d'un modèle d'OS Temps-Réel 153
8.6 Utilisation d'une Libraire pour la Configuration d'OS 153

8.7 Conclusion et futurs travaux 155

BIBLIOGRAPHY 156

LIST OF TABLES

4.1 Example of real-time task set schedulable under RM scheduling. 82

4.2 Task set of simple real-time automotive application 94

4.3 Simplified real-time applications. 99

4.4 Real-Time Virtual Machine System configuration. 99

6.1 Example of real-time task set schedulable under RM scheduling. 133

6.2 Overheads comparison. 137

LIST OF FIGURES

2.1 The native and the dual-kernel design of real-time Linux. 4

2.2 Native and Hosted VM Systems. 8

2.3 kvm software architecture. 10

2.4 Comparison of the TCB of three different virtual machine systems. 15

2.5 Nova software architecture. 16

2.6 Hierarchical Scheduling Framework concept. 17

2.7 Xen software architecture. 20

2.8 Request bound function. 27

2.9 Schematic of the overall architecture of Hijack$^{\mathrm{COS}}_{\mathrm{Linux}}$. 41

2.10 Schematic of the overall architecture of ExSched. 43

2.11 Schematic of the overall architecture of LITMUS$^{\mathrm{RT}}$. 45

3.1 Virtual Machine System Concept. 50

3.2 Intel ISA's operation modes and privilege levels. 53

3.3 ARM ISA's operation modes and privilege levels. 54

3.4 Linux Kernel Virtual Machine and Qemu. 55

3.5 Scheduling of the kvm and Qemu threads 56

3.6 The relationship between the scheduling of the kvm threads and the ksoftirq thread . . 56

3.7 Scheduling latency of a real-time Linux running natively on an Intel hardware. 59

3.8 Scheduling latency of a real-time Linux running on a virtual machine. 59

3.9 Scheduling latency of a real-time Linux measured on a recent Intel core i7 hardware. 61

3.10 Timeline illustrating release delay under fixed-priority scheduling 63

3.11 Architecture of the native and the virtual platform used in the experiments. 64

3.12 Feather trace toolkit. 65

3.13 Scheduling overhead 68

3.14 Event latency 69

3.15 Context-switch overhead 70
3.16 Release overhead 71
3.17 Distribution of release latency at n equals 14 tasks per processor, in the virtual case. . 72
3.18 Distribution of the context-switch overhead for n equals 10 tasks per processor. 72
3.19 Distribution of the context-switch overhead for n equals 20 tasks per processor 73
3.20 Scheduling overhead at n equals 5 tasks per processor measured in the virtual case... 73
3.21 Scheduling of virtual machines according to the fixed-priority algorithm. 75

4.1 Illustration of the temporal properties of a periodic task. 80
4.2 Scheduling of the task set from Table 3.1 according to the RM algorithm. 83
4.3 Scheduling of the task set from Table 3.1 according to the EDF policy. 85
4.4 Scheduling of virtual machines according to SCHED_FIFO scheduling algorithm. .. 86
4.5 Scheduling of virtual machines according to the RM algorithm. 87
4.6 V_l execution length. 88
4.7 Schedulability condition for a task T_i at a time $t + \cdot\Pi_i$ 90
4.8 Schedulability condition for a task T_i at a time $t + 2 \cdot \Pi_i$ 91
4.9 Schedulability condition of task T_i executed on the virtual machine V_l 91
4.10 Overhead related to release event 96
4.11 Scheduling of virtual machines using the periodic resource model 99
4.12 Real-Time application executed on two VMs scheduled by SCHED_DEADLINE ... 102
4.13 Scheduling of virtual machines VM^1 and VM^2 using the same priority. 103

5.1 System-level design flow for SoCs. 106
5.2 OveRSoC Development Tool. 108
5.3 OveRSoC's Library and design Tool. 109
5.4 OveRSoC component designer. 110
5.5 Model-Driven software development process. 113
5.6 Hierarchical Modeling Levels. 114
5.7 Meta Object Facility language. 116

5.8 RTOS structure meta-model. 116
5.9 RTOS-specific language tool. 117
5.10 Extended RTOS meta-model. 119
5.11 Model-to-Model transformation process. 121

6.1 Xilinx Zinq 7000. 124
6.2 Many-to-Many model. 127
6.3 Set of registers that constitute the CPU user-context. 128
6.4 Schematic architecture of the user-level library 130
6.5 Scheduling trace of a task set according to the RM algorithm. 133
6.6 Context-Switch overhead of the FP scheduler. 134
6.7 Scheduling overhead of the FP scheduler. 135
6.8 Scheduling overhead of the EDF scheduler. 135

8.1 Architecture logicielle des systèmes supportant des machines virtuelles. 143

LIST OF ABBREVIATIONS

ABD	All But Dissertation
BW	Bandwidth
CBS	Constant Bandwidth Server
DMR	Deadline Miss Ratio
DS	Deferable Server
EDF	Earliest Deadline First
FP	Fixed-Priority
G-EDF	Global Earliest Deadline First
G-FP	Global Fixed Priority
HRT	Hard Real-Time
HSF	Hierarchical Scheduling Framework
I/O	Input/Output
IPC	Inter-Process Communication
IPI	Inter-Processor Interrupt
KVM	Kernel Virtual Machine
LoC	Lines of Code
PRM	Periodic Resource Model
PS	Periodic Server
RBF	Request Bound Function
RM	Rate-Monotonic
SBF	Supply Bound Function
SEDF	Simple Earliest Deadline First
SMI	System Management Interrupts
SoC	System-on-Chip
SRT	Soft Real-Time
TLB	Translation Lookaside Buffer
TSC	Timestamp Counter
VM	Virtual Machine

VMM	Virtual Machine Monitor
WCET	Worst-Case Execution Time
WSS	Working Set Size

CHAPTER 1

Introduction

Multicore chips enabled with hardware-assisted virtualization mechanisms are commonly encountered in servers and personal computers. Such platforms offer a considerable processing capacity while reducing the space required to deploy the system. As a result, these platforms are also considered to be deployed in real-time embedded systems.

A key requirement when building safety-critical applications is isolation, *i.e.* the failure of one component should not crash the whole system. The easiest and historically most-commonly used way to ensure isolation is to employ a dedicated processor for each functionality. However, this approach has led to an increasingly unmanageable proliferation of such systems, to the effect that some modern cars may contain more than one hundred Electronic Control Unit. For example, the number of ECUs in the car has grown to the level where the complexity of the electrical and electronic system is difficult to manage. Every embedded system requires wiring and cooling, adds weight, requires space, drains power, and must be purchased, transported, tested, and documented, *etc.* Thus, instead of embedding one hundred networked, slow uniprocessors throughout a car, it would be desirable to use only ten (or fewer) shared, but ten-times as powerful, multicore processors that are highly utilized.

While in one case the migration of legacy applications from uniprocessors to multicore platforms requires the use of one operating system to manage all the applications, in other case, the use of several operating systems is necessary. For example, in the automotive domain, one real-time operating system (RTOS) will be used for real-time tasks, and a general-purpose operating system will be used to support in-vehicle infotainment applications. Each operating system will be executed in a separate and secure *virtual machine*. A virtual machine (VM) is a hardware and software technique that ”gives the impression” to the operating system that it is running on the real hardware while in

reality it is not. Thus, multiple virtual machines could be deployed on a physical machine, and are controlled by a *Virtual Machine Monitor* (VMM).

In addition to dependability requirements, the correctness of real-time systems is dependent on the system's ability to meet application temporal constraints. Expressed in terms of tasks deadlines, applications' resources requirements define the service levels required from the system. To behave in a predictable manner and support the correctness of these applications, the system must contain resource management policies capable of dealing with specific applications temporal constraints.

The goal of this dissertation is, first, to determine how to securely deploy multiple operating systems on a same hardware platform while preserving the temporal correctness of the system. Second, how to easily configure the real-time operating systems in order to adapt it to the applications requirements.

This dissertation is organized as follows: in Chapter 2 we review the studies related to our work on the use of virtualization in real-time systems, and the configuration of the resources allocation techniques of an RTOS. In Chapter 3 we evaluate the overheads and latencies of an RTOS that is deployed on a virtual machine. Then, in Chapter 4 we analyze the role of scheduling in maintaining the performance of real-time virtual machine system. Next, in Chapter 5 we present a transformation of an RTOS simulation model into an executable programs on a real hardware, then we define in Chapter 6 a method to preserve its configurability feature. We conclude this dissertation in Chapter 7.

CHAPTER 2

Related Work

There are multiple applications of a software architecture able to co-locate a real-time operating system and a general-purpose operating system. For example, in the automotive domain, an AUTOSAR compliant RTOS and a Linux Genivi operating system that support in-vehicle infotainment application, could be co-located on the same electronic control unit (ECU) (Heiser, 2011).

Multicore chips enabled with instruction set architecture (ISA) that support virtualization offer an efficient solution to fulfill such a requirement. New processors extended with this virtualization feature allow to execute multiple virtual machines on the same real hardware. Thus, it is possible to execute multiple unmodified operating systems at the same speed rate as on the native hardware.

This dissertation addresses two fundamental questions to the design of a real-time virtual machine system: what is the order of magnitude of the overheads and latencies of an RTOS running in a Virtual Machine, and how these overheads and latencies impact the temporal characteristics of a real-time system.

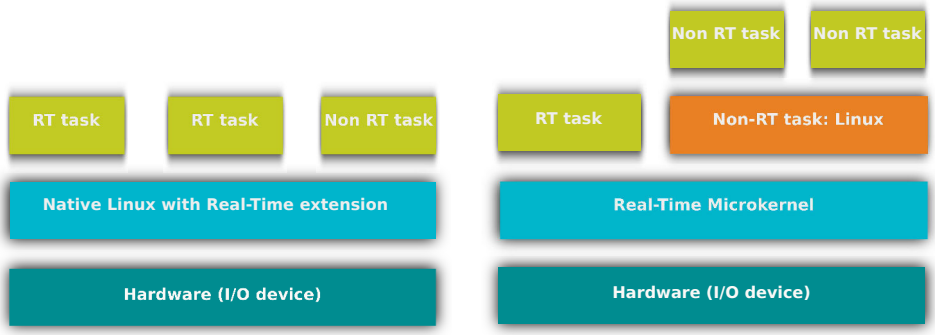
The third question addressed in this thesis is: how to transform a component-based RTOS model from its simulation form into an executable program, while preserving the configurability property offered by its design? By configurability of the RTOS we mean changing for instance its internal resources allocation policy and adapting the operating system by selecting the appropriate services required by the application.

In this chapter, we first present some approaches that investigated the combination of an RTOS and general-purpose OS without using virtualization, then we review the research studies that investigated the use of virtualization in real-time systems. Second, we discuss different proposed solutions to enable the configuration of a real-time operating system.

2.1 Real-Time OS alongside General-Purpose OS

Running a real-time OS alongside a general-purpose OS could be achieved by ”re-tailoring” an existing general-purpose OS to acquire the desired real-time features, for example, a real-time scheduler, temporal isolation, or low interrupt latency. One advantage of this approach is that it greatly reduces the development costs since basic OS functionality such as memory management, device drivers, and process abstraction, do not have to be re-implemented.

Due to its open source nature, Linux, is the most frequently chosen operating system when combining a general-purpose and a real-time operating system. There are two variants of solution based on Linux. In a *native design*, the Linux kernel is the only kernel responsible for meeting the real-time requirements. The real-time tasks are regular Linux processes as indicated in Figure 2.1(a). In contrast, in a *dual-kernel design*, a specialized hard real-time-capable kernel is inserted between the Linux and the hardware. Such an implementation follows the classical *microkernel* design in which Linux is an OS server and is scheduled as a background, non-real-time thread by the microkernel. Real-Time tasks are not Linux processes, they are specialized threads managed directly by the *small kernel* besides the Linux kernel as depicted in Figure 2.1(b). In the next two sections we first review the dual-kernel variant, then we discuss the native-kernel variant.



(a) Native Design (b) Dual Kernel Design

Figure 2.1: In a native real-time Linux, real-time tasks are processes of the Linux kernel. While in dual kernel design the real-time tasks are managed by a special real-time kernel isolated from Linux. Linux is executed as a non real-time task of that small kernel.

2.1.1 Dual-Kernel Design

In the early versions of Linux, all system calls and interrupts handling were executed as one non preemptive section. This greatly simplified synchronization requirements on a uniprocessor. However, it could lead to excessively long non preemptive section in the case of a real-time system. For example, when a high priority real-time job is released, the corresponding real-time process could be delayed if the kernel were executing on behalf of a lower-priority task. As a result, a non preemptive kernel with long code paths may cause real-time tasks to incur unacceptable delay in the worst case.

Consequently, real-time Linux variants, in particular those focused on hard real-time applications, chose to work around the Linux kernel and its internal limitations using a dual-kernel design approach. In this approach, Linux is executed as a real-time background task of a small real-time kernel. In this case, the Linux kernel is not in full control of the hardware, does not have the right to disable interrupts, and thus can be preempted at any time.

There are two key advantages to such dual-kernel design. First, low interrupt latencies can be guaranteed to real-time tasks regardless of any deficiencies in the Linux kernel. Second, only relatively small changes to the Linux kernel are required, which means that integrating improvements made in newer Linux versions is relatively easy.

A disadvantage of the dual-kernel design is that real-time tasks execute directly on top of the *small kernel* and cannot make use of the Linux services such as device drivers, POSIX IPC, synchronization primitives, file-systems, *etc.* This limitation is fundamental since a dual-kernel does not improve Linux's real-time capabilities, rather, it enables real-time tasks to safely co-exist with the Linux kernel.

There are two main classes of dual-kernel Linux. L^4Linux (Lackorzynski, 2014) is an example of a dual-kernel system where both Linux and real-time tasks are executed in private address spaces and thus isolated from each other. Also several commercial RTOSs offer Linux dual-kernel support as well, among them Green Hills's INTEGRITY, Sysgo AG's PikeOS, and LynxWorks's Lynx OS. In contrast, real-time tasks execute in kernel mode in RT-Linux (Zijlstra, 2008) and are thus not isolated from the Linux kernel. Besides RT-Linux, two other well-known real-time Linux based on the dual-kernel design approach that omit isolation are the Real-Time Application Inter-

face (RTAI) (Cloutier et al., 2008), which targets industrial applications, and the *Xenomai* project, which targets similar use cases but also focuses on providing RTOS compatibility layers (so-called *skins*) to support legacy applications (Gerum, 2008).

2.1.2 Native Real-Time Linux

In a native design, one kernel is present and in full control of the hardware platform. Only the Linux kernel is modified in order to enhance its real-time capabilities. This design is preferred to the dual-kernel design in the vast majority of applications if timing constraints can be met, that is, if Linux's limitation such as high interrupt latencies can be addressed. In the case of applications with very stringent constraints (*e.g.* engine control software), a dual-kernel approach may be the only feasible design.

Multiple works attempted to integrate a real-time infrastructure to the Linux kernel. For example, the Kansas University Real-Time Linux (KURT Linux), developed by Srinivasan et al. (1998) provided the high-resolution (software) timers based on hardware timers operating in one-shot mode ("UTIME" patch). Later, this design was re-implemented in a POSIX-compliant way and merged into a standard Linux under the name *hrtimers* (Gleixner and Niehaus, 2006).

Linux versions higher than 3.0 are suited for use in real-time systems, and the current native real-time Linux design focus on scheduling and locking algorithmic changes. Linux 3.0 gained several improvements over the course of several versions that greatly improved its viability as an RTOS, namely high-resolution timers, priority inheritance, mostly preemptable kernel execution, much-shortened non-preemptive sections, and an improved lower-overhead fixed-priority scheduler. Mainline Linux is now almost POSIX-compliant and supports fixed-priority scheduling (SCHED_FIFO and SCHED_RR) with 100 distinct priorities, processor affinity masks, and the *priority inheritance protocol*.

However, the Linux kernel still contains some limitations in terms of non-preemptive code paths that are long in the context of real-time systems and architectural design choices that were made with throughput in mind. For example, interrupts are, by default, not serviced using split interrupt handling; rather, Interrupt Service Routines (ISRs) are typically executed immediately when an interrupt is raised and are not subject to scheduling. Executing ISRs right away benefits network and disk bandwidth, but can also delay real-time tasks. Thus, while API-compatible, current mainline

Linux is not yet comparable to purpose-built RTOSs such as VxWorks or QNX Neutrino in terms of predictability and interrupt latency.

Moving beyond this limitations is the goal of the PREEMPT_RT patch, which is the *de facto* real-time standard variant of Linux. It changes the Linux core infrastructure significantly by reducing the number and the length of non-preemptive critical sections, converting most spin-locks in the kernel to semaphores, and further enables the *priority inheritance protocol* by default for all semaphores in the kernel. One important feature introduced by the patch is to force split interrupt handling for all ISRs except timers. The PREEMPT_RT patch is under active development, and besides serving as a staging ground for real-time features that are intended to be incorporated into mainline Linux at a later point, it is also widely used in industrial projects.

Summary. While real-time Linux variants offer a good approach to co-locate a real-time OS and a general purpose OS, such a design could be problematic in the case where a legacy application has already been developed and certified on a existing RTOS. In this situation, adopting a design based on a real-time Linux variant would require the porting of the application using a new API, and going through a new certification process if the application is used in a safety-critical system. This extra work increases the development cost and the already tight time-to-market.

An alternative solution could be provided by the use of the virtual machine concept. By using a Virtual Machine System, it is possible to run the existing RTOS and the application in a virtual machine alongside a general-purpose operating system on the same hardware. We study this approach in the next section.

2.2 Virtual Machine Systems

A Virtual Machine System is a concept intended to host multiple operating systems simultaneously on a single hardware platform. Each *guest* operating system is executed in a separate and secure Virtual Machine (VM). The virtual machine is the abstraction of the hardware resources provided to the guest operating systems, and managed by a low-complexity kernel referred to as a Virtual Machine Monitor (VMM).

The virtual machine monitor must ensure that a temporal or local fault in one virtual machine (*e.g.*, an infinite loop, out-of-bounds array access, exhaustion of assigned resources) does not af-

fect the operation of other correct virtual machines. This requirement is referred to as *logical* and *temporal isolation* in the real-time community, and as *space* and *time partitioning* in the RTOS industry.

Virtualization can be implemented in different ways. The classic approach to design a virtual machine system is to place the VMM on bare hardware whereas the virtual machine fits on top. The VMM runs in the most highly *privileged level*[1], while all guest operating systems run with lesser privileges, as shown in Figure 2.2. Then, in a completely transparent way, the VMM can intercept and implement all the guest OS's actions that interact with the hardware resources.

An alternative implementation builds the VMM on top of an existing host operating system, resulting in what is called a *hosted VM* as shown in Figure 2.2c and Figure 2.2d. In this configuration, the installation process is similar to installing a typical application program.

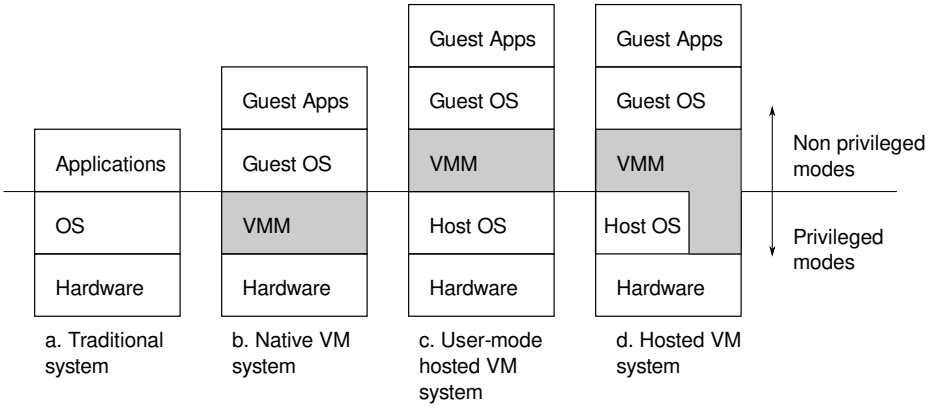


Figure 2.2: Native and Hosted VM Systems.

Executing multiple guest operating systems by a VMM is similar to the execution of multiple user processes by an operating system in a conventional time-sharing system. The VMM moves the entire guest registers' contents into the host's registers after saving the registers of the previous guest into memory. Then, the execution can proceed at the same speed rate as on a machine running the guest natively. Once the VMM gives the resources to a guest virtual machine, it is important that the VMM could get them back so they can be later assigned to a different VM. Again, this step

[1]This level (reserved for the most privileged code, data, and stacks) is used for the segments containing the critical software, usually the kernel of an operating system. The other privilege levels are used for less critical software. For instance, the x86 Intel architecture has 4 privilege levels. Linux on the x86 architecture uses the highest privilege level, and the applications use the lowest one, the other intermediate levels are not used.

is similar to an operating system that regains control of all the hardware resources when it executes multiple user jobs concurrently on a machine.

Virtual Machine Systems have been widely deployed in the domain of enterprise server. Given this success, these systems are also increasingly deployed in the embedded real-time systems. A considerable research effort has been spent in adapting existing virtual machine systems used in the server domain to the real-time embedded system domain.

With regards to the questions of this thesis, we are interested in investigating the capability of the existing systems that were initially designed for the server domain to support the real-time embedded systems demands. While reviewing the existing work in this direction, our approach is to understand the limitation of the initial implementation targeting the server domain and how it was adapted to fulfil the requirements of the real-time systems.

2.3 Real-Time Virtual Machine Systems

In this section, we review how existing Virtual Machine Systems have been adapted to real-time systems. For each solution, we first describe its design and implementation, then we discuss its real-time performance.

2.3.1 Linux Kernel-based Virtual Machine

Linux Kernel Virtual Machine (shortened as kvm) is a *hosted VM* system (Kivity et al., 2007). Its main task is to manage unprivileged access to hardware features that can only be used directly by the privileged kernel. Its tremendous success is in large part due to its relative simplicity compared to other approaches. This simplicity is achieved by leveraging the functionality already provided by the Linux kernel, and relying on hardware-assisted virtualization, which allows it to be ported to wide range of architectures such as x86, PowerPC, ARM and IBM s390.

Under kvm, when a virtual machine is created, a data structure is instantiated to hold in memory the CPU registers used by the guest operating system, and acts as a **virtual CPU** (vCPU). A virtual CPU is associated to a regular Linux process and scheduled by the Linux kernel scheduler alongside the other processes. The spatial isolation between virtual machines relies on the Linux's virtual memory management, for each virtual machine a separate memory address space is created. Each

guest operating system has its own memory separated from the other guests. Figure 2.3 illustrates the integration of kvm into the Linux kernel.

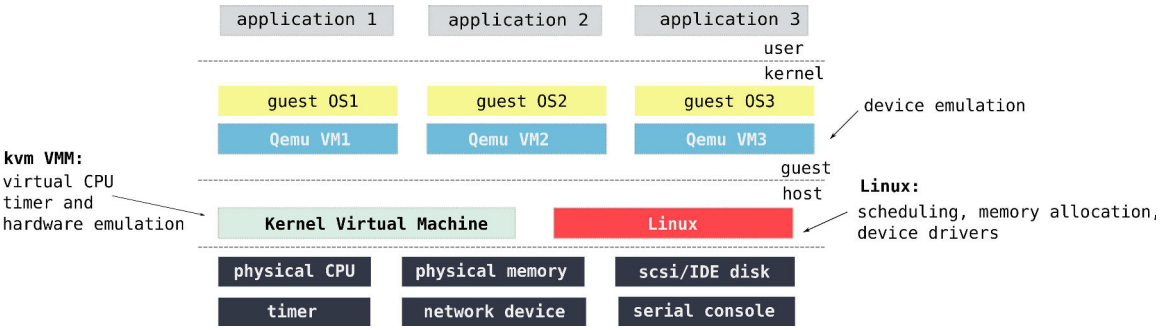


Figure 2.3: kvm software architecture.

kvm is a hosted-VM system that requires two components: a VMM-native (VMM-n) and a VMM-user (VMM-u) components (see Figure 2.2d):

VMM-n (native). This component runs natively on the hardware and has characteristics similar to the VMM on a *native VM* system. It is the component that intercepts traps due to the privileged instructions executed by a guest operating system running in a virtual machine.

VMM-u (user). This component runs as a user-level process on the host operating system. It makes resource requests to the host OS, in particular, memory and I/O requests, on behalf of the native mode VMM using system library functions supplied by the host operating system.

An important task of kvm is to provide fast virtualization for frequently accessed guest devices. Specially, the interrupt and timer controllers are provided by the VMM-n component. The advantage is that no consultation of the VMM-u component is required if a guest accesses any of these devices, which reduces the virtualization overhead.

As mentioned earlier, kvm creates for each virtual machine a virtual CPU to hold the guest CPU state. When the guest operating system executes a *privileged instruction*[2], the hardware virtualization mechanism triggers an "exit " from the virtual CPU execution context to the VMM. If the privileged instruction could not be handled by the VMM-n component, the exit event is propagated to the VMM-u component.

[2]Some of the system instructions called "privileged instructions" are protected from use by application programs. They control system functions (such as the loading of system registers). They can be executed only at the most privileged level. If one of these instructions is executed at lower privilege level, a general-protection exception is generated. The x86 WRMSR instruction (write model specific registers) is an example.

The VMM-u component attach to the virtual CPU's execution context one of its threads. And when the Linux kernel schedules the thread, it results in running the guest code. Thus, any modification applied to the Linux scheduler influences also the scheduling of the virtual machines.

Consequently, a virtual machine process could also be preempted by the host interrupts and processes which represents a problem in the case where a real-time application is executed in the virtual machine. Because, if the thread associated to the virtual machine is preempted by another process or an interrupt running in the host, the response time of the currently running real-time task in the guest OS could be affected. A simple solution to this problem is to raise the priority of the virtual machine thread and to configure it as a real-time thread[3].

The evaluation of the real-time capability of kvm has been investigated from an implementation perspective. Multiple studies (Bing, 2010; Forsberg, 2011; Åsberg et al., 2011; Kiszka, 2010; Ramachandran, 2013; Zhang et al., 2010b,a; Zuo et al., 2010) measured the *scheduling latency* of the guest operating system. The measurement of this latency usually used the cyclictest benchmark from the rt-test project (Molnar, 2004). Concretely, it repeats the measurement of the sleep( ) system call latency during a specified duration, for example 15 minutes, or one hour. Then, the results of the minimum, the maximum and the average latency are reported at the end of the experiment.

Testing this benchmark on an operating system that is running on a real hardware and on a virtual machine allow to observe the effect of virtualization mechanism on the operating system performance, and whether the raise of the priority of the virtual machine's process results in an improved performance or not.

The comparison of the measurements of the latency from a native OS vs. an unprioritized guest OS, and vs. a prioritized guest OS, showed that the probability of the multiple milliseconds latency was much higher in the unprioritized guest than on the native OS. The maximum latency exceeded the $100ms$ (Zhang et al., 2010b). However, the prioritization of the guest OS, that is, configuring the virtual machine thread as a real-time thread and raising its priority, significantly decreased the average latency.

[3] In Linux, the threads that are configured to be scheduled under the SCHED_FIFO or SCHED_RR scheduling classes are considered real-time as threads and treated prior to the regular non-real-time processes.

Being integrated to the Linux kernel, kvm benefits from the real-time properties of the Linux kernel. Any improvements to the real-time capability of Linux through scheduling algorithms, synchronization, preemption, low latency, or drivers will bring better performance to kvm.

One promising solution in this direction is the use of the PREEMPT_RT real-time patch to configure the host Linux kernel. In this configuration, the host system is enabled with real-time capability which improves the response time of the virtual machine thread.

The repeating of the precedent experiment (the cyclictest benchmark) using the configured host OS (Linux configured with the real-time PREEMPT_RT patch) revealed that the application of the PREEMPT_RT patch reduced the average-case latency to less than $1ms$, and removed the $100ms$ maximum latency observed in the non-prioritized guest OS.

Cucinotta et al. (2009a) evaluated the real-time capability of kvm from a theoretical perspective. The authors investigated the problem of guaranteeing temporal isolation among multiple VMs managed by Linux and kvm, and experimented two test cases. First, they executed two real-time tasks, $T_1 = (30ms, 150ms)$ and $T_2 = (50ms, 200ms)$, on a real hardware. Second, they executed the same task set on a virtual machine using kvm. Then, they measured the response time of all the jobs of the two tasks. The observation of the results showed that when the task set is executed inside a virtual machine most of the deadlines were easily missed.

The authors attributed this result due to the general-purpose scheduling used to allocate the host CPU resources to the virtual machines. They stated that simple solution based on a proportional-fair share algorithm may fail to guarantee a sufficient degree of isolation, and do not generally provide enough control over the granularity of the CPU allocation to the various VMs. The authors also declared that using a *fixed-priority* algorithm would create a second problem because in the case where a higher priority VM consumes more CPU time than expected it could prevent a lower priority VM from running.

The alternative solution proposed by Cucinotta et al. (2009a) is to use the well established real-time scheduling techniques, in particular the *resource reservation* to schedule the virtual machines. Such a technique associates to each VM a reservation tuple (Θ, Π), where Θ is the processor time reserved for a VM every Π time units.

The proposed approach suits the needs of concurrently running VMs, because it allows to control the amount of time required by each VM, and guarantees the respect of deadlines for the tasks

running inside the VM. This approach relies on the hard reservation variant[4] of the Constant Bandwidth Server (CBS) scheduling policy (Abeni and Buttazzo, 1998). It was implemented using kvm and the AQuoSA framework (Cucinotta et al., 2009b) for Linux kernel.

Given a task set $\tau = \{T_1, ..., T_n\}$, a virtual machine VM^k that is allocated a resource reservation (Θ, Π), and using a CBS algorithm to schedule the virtual machine and fixed-priority algorithm to schedule the the task set τ, it is possible to guarantee the *schedulability* of the task set τ if and only if:

$$\forall i \exists t \in P^k : e_i^k + \sum_{j<i} \left\lceil \frac{t}{T_j^k} \right\rceil \cdot e_j^k \leq Z^k(t), \tag{2.1}$$

where e_i^k is the *worst case execution time* of a task τ_i^k in a virtual machine VM^k. $Z^k(t)$ is a *characteristic function* indicating the amount of time allocated to the VM^k by the root scheduler, and P^k is a set of appropriate scheduling points.

To evaluate this approach two experiments were conducted. In the first experiment two simple real-time task sets were used in order to easily understand the behavior of the system, and in the second experiment multiple web servers were executed in virtual machines to demonstrate the effectiveness of the approach in a real-world *service oriented architecture* scenario.

In the first test, two task set were used, $\tau_a = \{T_1(30ms, 150ms), T_2(50ms, 200ms)\}$ was executed on the virtual machine VM^a, and $\tau_b = \{T_3(30ms, 120ms), T_4(40ms, 240ms)\}$ on VM^b.

The results of the tests showed that when the two VMs were executed without CBS the deadlines were easily missed. However, by allocating a resource reservation $(a = (28ms, 50ms))$ for VM^a and $(b = (52ms, 120ms))$ for VM^b, all deadlines have been respected. However, the article does not demonstrate how the parameters (Θ, Π) of each resource reservation were calculated.

2.3.2 Microkernel Support for Virtualization

Similar to monolithic operating system, microkernel-based operating systems were also extended to provide the virtual machine monitor functionality. OKL[4] microvisor from Open Kernel Labs, L[4]Fiasco and Nova microhypervisor from the Technische Universitaet Dresden are examples of

[4]In the particular case of a *hard* reservation, the VM is not allowed to execute more than Θ time units every Π.

microkernel-based operating systems that are derived from the L4 microkernel family (Liedtke, 1996).

2.3.2.1 OKL^4 microvisor

The OKL^4 microvisor is considered as the first commercial VMM deployed on a mobile phone (the Motorola QA4). A prototype based on OKL^4 (Varanasi and Heiser, 2011) was recently ported to the ARM Cortex A15, in order to benefit from the support of the new hardware instruction set that allows the execution of unmodified guest operating system binaries inside a virtual machine. The developed microkernel was evaluated on the ARM *Fast Models* simulator due to the unavailability of hardware implementation of the architecture that supports the virtualization extension.

The evaluation of the VMM implementation on the CPU simulator (not cycle-accurate) allowed to estimate its low-level performances. Using a number of micro-benchmarks the execution time measured in CPU cycles of the VMM routines were calculated based on the instructions count from collected traces, and the weighting of the instructions by their known latencies in cycles from the ARM Cortex A9 processor and an equivalent memory system.

For example, the IRQ (interrupt request) entry, which is the entry to the VMM IRQ routine upon the arrival of an interrupt, is estimated to 239 instructions, which is approximated to 700 cycles. As a comparison to x86 architecture, the same operation measured using the Nova microhypervisor, would costs 4,000 cycles. Switching between virtual machines contexts was done efficiently using the ARM's multi-register operations to save and restore state. Part of this state is kept in co-processor (MMU) and external core such as virtual interrupt and devices registers which are more expensive to access than internal registers. As a result, the overhead of switching between VMs was estimated to 2842 instructions which is equal to 7555 cycles.

The estimated performances of this VMM prototype, and its approximated 6,000 lines of code for its fully-functional version, allowed to take the decision of turning this prototype into a commercial product. However, due to the fact that this implementation is at prototype level, no real-time performance evaluation were conducted yet.

2.3.2.2 Nova microhypervisor

The Nova microhypervisor (Steinberg and Kauer, 2010) is the third generation of the L4 microkernel that supports virtualization since its design phase, and not as an extension of existing L4 microkernel versions such as L^{4}Fiasco.

Like all the variants of L4 microkernel, Nova is designed based on the principle of small *trusted computing base* (TCB). These systems take an extreme approach to the principle of least privilege by using small kernel that implements only a minimal set of abstractions. Liedtke (1996) recommended three key abstractions that should be provided by a microkernel: address spaces, threads, and interprocess communication. The other functionality should be implemented at user-level.

The main characteristic of Nova is its TCB size. The TCB is the part of the software that runs at the highest privilege level, and must be trusted. Comparing to Linux-kvm and Xen, the TCB of Nova is at least an order of magnitude smaller than these systems. Figure 2.4 summarizes the comparison of the total sizes of Nova, kvm and Xen.

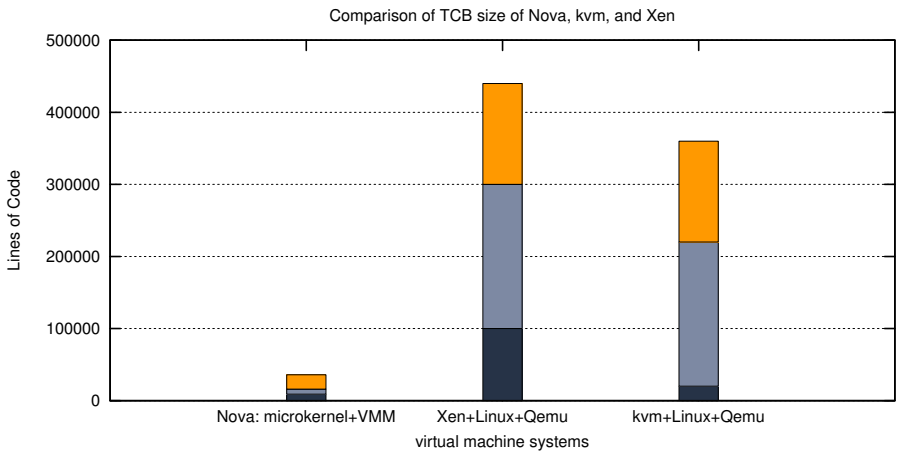


Figure 2.4: Comparison of the TCB of three different virtual machine systems.

The size of the Nova's TCB is 9,000 lines of code (LoC), whereas the Xen's TCB is equal to 300,000 LoC, and the kvm is 220,000 LoC. This is because Xen VMM is about 100,000 LoC, and uses a privileged domain[5], called *dom0*, which is a Linux kernel (200,000 LoC) and all its device drivers. In order to emulate devices, the Qemu hardware emulator is executed as a user application on top of Linux. kvm however is part of the Linux kernel, thus its TCB size is equal to the sum of

[5]In the context of Xen, a domain is equivalent to a virtual machine.

Linux kernel source code plus the required file-system, plus device drivers, and the source code of kvm itself (20,000 LoC). In total it is estimated to be 220,000 LoC.

Small TCB is an important security requirement of safety-critical systems. The VMM is responsible for controlling the platform, and if an adversary manages to compromise it, subverting the security of all hosted operating systems would be easy. Reducing the TCB will reduce the attack surface significantly, and thereby improves the security of the system.

To achieve such a feature, the Nova VMM was designed as a decomposed virtualization architecture that minimizes the amount of code in the privileged VMM as illustrated in Figure 2.5. By implementing the part of the VMM that emulates the instructions at user-level, it was possible to trade improved security for a slight decrease in performance.

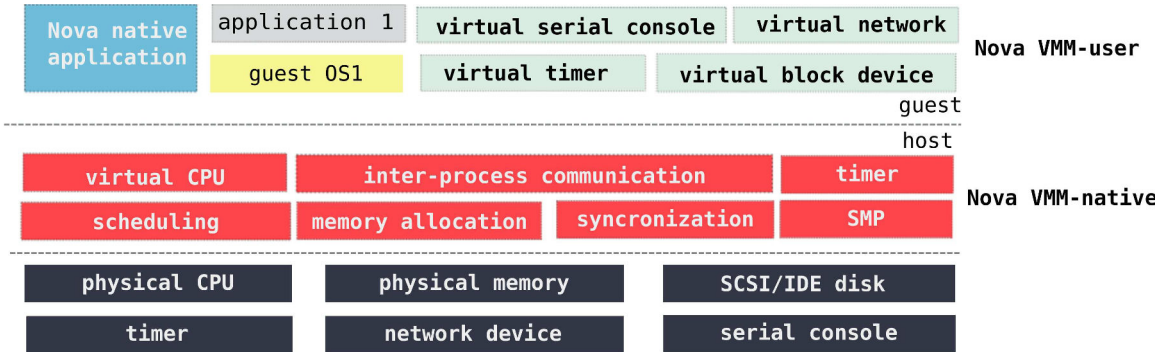


Figure 2.5: Nova software architecture.

Regarding the real-time characteristics, Nova implements a fair share scheduling using a preemptive priority-driven round-robin policy with one run-queue per CPU. When invoked, the scheduler selects the highest-priority thread from the run-queue and dispatches it. Once dispatched, the thread can run until its time quantum is depleted or until it is preempted by the release of a higher-priority scheduling context.

2.3.2.3 L^{4}Fiasco microkernel

The problem of using a microkernel in a real-time virtual machine system has been explored by Yang et al. (2011). The authors used the L^{4}Fiasco microkernel as a VMM and the *paravirtualized*[6]

[6]Paravirtualization is a technique for reducing the performance overhead of virtualization by making a guest operating system aware of the virtualization environment. It replaces *privileged instructions* in the guest OS with *hyper-calls* to the virtual machine monitor.

L^{4}Linux, a modified version of Linux kernel, in which the HAL (hardware abstraction layer) in Linux have been replaced by a set of calls using the microkernel API (application programming interface). The L^{4}Linux is considered by the microkernel as a user-level thread. The Linux kernel uses the set of *hyper-calls* provided by L^{4}Fiasco to request the privileged operations that it is not able to perform due to its unprivileged status.

The authors argued that a two-level Hierarchical Scheduling Framework (HSF) is naturally suited to build a real-time virtual machine system. In such a design, the root scheduling level is the microkernel scheduler and the second level scheduler is located at the L^{4}Linux scheduler as illustrated in Figure 2.6.

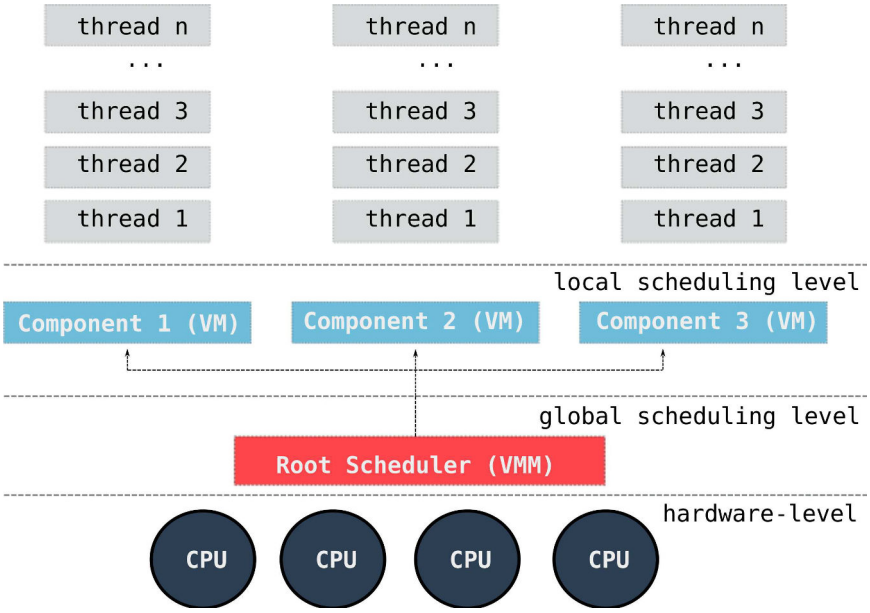


Figure 2.6: Hierarchical Scheduling Framework concept.

The root scheduler in the microkernel schedules the L^{4}Linux server[7] using a *periodic resource model* (PRM), denoted by the tuple $\Gamma = (\Theta, \Pi)$. And the scheduler of L^{4}Linux schedules the real-time tasks. Both scheduling levels employ the *fixed-priority rate-monotonic* policy.

The L^{4}Linux server is composed by several L^{4}Fiasco threads such as the Linux kernel thread, the timer interrupt thread, and an idle thread. For each real-time task created by L^{4}Linux, an L^{4}Fiasco *shadow* thread is created and attached to it. Releasing a real-time task in L^{4}Linux releases a shadow thread in L^{4}Fiasco that executes the user-code on behalf of the task.

[7] In the context of hierarchical scheduling theory a server is synonym of component.

In the implementation, the Linux kernel thread and the L^{4}Fiasco shadow threads are considered as one scheduling group, and scheduled together using the same execution budget from their associated PRM. However the corresponding interrupt timer thread is treated independently and given a higher priority to ensure that it is scheduled as soon as an interrupt is triggered.

To preserve the execution budget associated to each PRM, a *real-time timeout* has been created and set equal to Θ to prevent that the current L^{4}Linux kernel thread and subsequent shadow L^{4}Fiasco threads from being disturbed by other VM's threads during this amount of time, except by other interrupt threads.

The PRM associated to each virtual machine is calculated dynamically by the L^{4}Linux each time a new real-time task is created. The PRM is then given to the VMM which will take into account the new value at the next scheduling period. In the implementation the Π was fixed to $500ms$.

To calculate the PRM, the authors fixed the period Π and used the *periodic capacity bound for rate-monotonic* scheduling as defined by Theorem 2.1 to determine Θ.

Theorem 2.1 (Shin and Lee (2003))**.** *For a given workload W, a period* Π*, and under the fixed-priority rate-monotonic policy, the execution time* Θ *is:*

$$\Theta = \max_{\forall T_i \in W} \left(\frac{-(p_i - 2\Pi) + \sqrt{(p_i - 2\Pi) + 8\Pi \cdot I_i}}{4} \right), \tag{2.2}$$

where,

$$I_i = e_i + \sum_{T_k \in high-priority(W,T_i)} \left\lceil \frac{p_i}{p_k} \right\rceil \cdot e_k, \tag{2.3}$$

and p_i, e_i are the period and the execution time of a task T_i respectively. The PRM is calculated at runtime by the L^{4}Linux server and given to the L^{4}Fiasco through the l4-rt-change-timeslice( ) hypercall.

The evaluation of the HSF implementation and its comparison with the round-robin scheduling policy and the RM scheduling policy already implemented in L^{4}Fiasco using two virtual machine showed that the HSF was able to avoid any deadline miss of the real-time tasks running in the VMs. Two scenarios have been evaluated, first, the task sets $\tau_a = \{T_1(1, 0.2), T_2(1.2, 0.2), T_3(1.5, 0.2)\}$ and $\tau_b = \{T_4(20, 2), T_5(30, 2)\}$ were executed in VM^a and VM^b respectively. Second, the task sets

$\tau_a = \{T_1(8, 1.5), T_2(10, 2)\}$ and $\tau_b = \{T_3(2, 0.1), T_4(3, 0.1)\}$ were executed in VM^a and VM^b respectively.

In the first scenario, the tasks T_2 and T_3 miss some deadlines under the round-robin scheduling and this could be explained by the fact that if all tasks are released at the same time, and the CPU time is shared fairly among the two VMs, the execution of VM^b delayed the execution of the tasks in VM^a.

In the second scenario the task T_3 and T_4 incur some deadline miss under the RM scheduling. The reason for this is because VM^a has given a higher priority than VM^b, because VM^a's CPU utilization = 0.39 and VM^b's CPU utilization = 0.28, and VM^a retains the CPU for 3.5 $second$ which delays the execution of T_3 and T_4 jobs.

With regards to overheads, three operations have been measured, the *selecting of a next thread in the ready queue*, the *setting of the real-time timeout*, and the *calculation of the periodic resource model interface* on a dual-core Intel 2.0GHz machine. The setting of the timer is done every $500ms$ and is estimated to $500\mu s$ when two VMs are running. Setting a real-time timeout prevent other VM's tasks from disturbing the execution of the current selected VM. The overhead of selecting the next ready task is less than $25\mu s$ when two VMs are running. This overhead and the overhead of calculating the PRM depend linearly on the number of running VMs. The most expensive operation is the calculation of PRM due to IPC communication, however the authors argued that this is reasonable because it occurs only when a new task is spawned.

2.3.3 Xen

Xen is a *native VM* system (Barham et al., 2003) that was initially designed to host multiple commodity operating system instances on a modern server. The Xen VMM is responsible for the CPU scheduling and the memory allocation. Xen uses a special guest operating system called *driver domain* containing the device drivers to provide access to the actual hardware I/O devices. Xen VMM grants the driver domain direct access to the devices and does not allow the other guest domains to access them directly. Therefore, all I/O requests must pass through the driver domain. Figure 2.7 illustrates the software architecture of Xen.

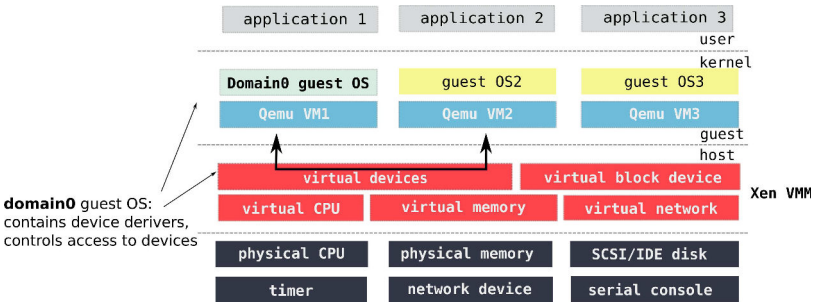


Figure 2.7: Xen software architecture.

The Xen VMM protects the guest domains from each other and shares I/O resources through the driver domain. This enables each guest operating system to behave as if it was running directly on the hardware without worrying about protection and fairness.

In the Xen terminology a virtual machine is also called a *domain*. The default scheduler in Xen is the *Credit* Scheduler. The domains in Xen are scheduled according to their state. Each domain could be in the UNDER state or in the OVER state. In the UNDER state, domains still have a remaining credits, and in the OVER state domains have gone over their credit allocation. Credits are periodically debited every $10ms$. When a scheduler interrupt occurs the currently running domain is debited 100 credits. The domains' credits are replenished when the sum of the credits of all domains in the system goes negative. When making scheduling decisions, domains in the UNDER state are prioritized over the domains in the OVER state. If there is no domains in the UNDER state and the processors would be idle, the domains in the OVER state could be executed.

The Credit scheduler selects a domain to run depending on its state. It does not considers the absolute number of credits that remain for a domain. Rather, domains in the same state are selected according to the *first-in first-out* policy. Domains are always inserted at the end of the run queue after the domains in the same state. The scheduler selects the domain at the head of the run queue. A selected domains is allowed to run for $30ms$ as long as its credit allows.

Xen provides a real-time scheduler called the *simple earliest deadline first* (SEDF). It schedules domains according to two parameters: the *slice* and the *period*. Runnable domains are allowed to execute periodically for an amount of time units given by their slice. The SEDF scheduler maintains for each domains a *deadline*, the time at which the current domain's period ends, and the amount of

processing time the domain is due before the deadline passes. The domains are ordered in the run queue according to their deadlines.

According to the SEDF scheduler, a domain can only be activated once if it blocks during its period, independently of whether it has used its whole slice or not. This could represent a problem with regards to the *worst-case execution time*. For example, in the case of a driver domain that is reacting to multiple networking packets, if it finishes processing all pending packets and blocks itself for the remainder of its current period, then if packets arrive while driver domain is blocked, they will be delayed until the next activation of the driver domain by the Xen scheduler irrespective of whether the driver domain has used its complete slice in the current period or not.

Masrur et al. (2010) proposed an improvement of the SEDF by allowing the driver domain to utilize its complete slice within its period independently of its current state (waiting or blocked). Moreover, the critical domains are given higher fixed-priority than the other domains which improve their response time and avoid any deadline miss.

They experimented their implementation by running in each real time domain only one task because they used a standard operating system available for Xen which does not have real-time capabilities. Thus, by running one real-time task per domain, they can still guarantee correct timing behavior, because the real-time task will be scheduled whenever the corresponding domain is scheduled by Xen independently of the scheduler used by the OS in the domain. Using one task per domain allows for a higher CPU utilization, because it is possible to select the slice and the period of each VM to fit the specific requirements of the only task running on it.

In a second work (Masrur et al., 2011), the authors proposed a two-level hierarchical scheduling architecture in Xen. The domains were scheduled under the *rate-monotonic* policy (RM), and the tasks in each domain were scheduled under the *deadline-monotonic* policy (DM), resulting in a DM over RM hierarchical scheduling.

They proposed a method to calculate the optimal time-slice and period for each domain in order to provide a *schedulability condition* for a set of real-time tasks running in a set of domains to meet their deadlines. The period of a domain is specified by the minimum deadline that has to be scheduled on that domain. And the selection of an efficient time slice requires an iterative procedure. Using the minimum requirements for a VM and the schedulability condition of a task running on that VM, they compute the initial domain's time slice. Then, this value is improved towards the

optimum in a reduced number of subsequent steps. In their experimental setup they compared the case where one real-time task was executed per domain to the case where all application's tasks were executed in one domain. They observed that the average response time improves when only one task is executed in one domain.

By providing isolation and managing the access to the hardware resources, a virtual machine monitor allows multiple virtual machines to share the same physical machine safely and fairly. The scheduler within the VMM is responsible for maintaining the overall fairness and performance characteristics of the virtual machine system. Traditionally, maintaining the fair share of the processor resources among the domains was the main focus of the VMM scheduler, and the scheduling of I/O resources was left as a secondary concern. This could result in poor and unpredictable I/O performance, making the virtual machine system less desirable for application whose performance is critically dependent on I/O latency and bandwidth.

Ongaro et al. (2008) explored the relationship between domains scheduling in a VMM and I/O performance. To verify the correctness of their assumptions, the Xen scheduler was used. They examined a number of new and existing extensions to Xen's Credit scheduler targeted at improving the I/O performance.

They analyzed the impact of VMM scheduling on I/O performance using multiple guest domains concurrently running different types of applications. They concurrently tested processor-intensive, latency-intensive, and bandwidth-intensive applications to quantify the impacts of different scheduler configurations on processor and I/O performance. Their tests revealed several insights into the key problems in VMM scheduling.

For instance, they observed that both the Credit scheduler and the SEDF scheduler within Xen achieve a good performance of fairly sharing processor resources among compute-intensive domains. However, the schedulers do not achieve the same performance when bandwidth-intensive and latency-intensive domains are executed.

The Credit scheduler in Xen uses the credit/debit system to fairly share the processor resources. It is invoked whenever an I/O event is sent and boosts the priority of an idle domain receiving that I/O event. However, the domains are not sorted in a run queue according to their remaining credits. To improve the I/O performance, two key optimizations were proposed. First, avoid preempting the driver domain while it is de-multiplexing I/O packets. Second, sort the run queue using the

domains remaining credits. These optimizations come from the observation that the I/O-intensive domains will often consume less credits than the compute-intensive domains. In fact, I/O-intensive domains are not debited any credit if they happen to block before the occurring of the scheduler interrupt. When they become runnable later, their remaining credit do not influence their order in the run queue, it only determines their state (UNDER or OVER) as mentioned earlier. A domain is always enqueued after the last domain in the same state. In the case where there is multiple compute-intensive domains that are inserted in the run queue before the I/O-intensive domain, it will wait for all the preceding domains to finish before it can run, which could increase its response time. However, by sorting the run queue based on remaining credits allows infrequently running, but latency-sensitive domains to run sooner. These two optimizations have a positive effect on the I/O performance of the virtual machine system.

2.3.4 RT-Xen

RT-Xen (Xi et al., 2011) integrates the fixed-priority hierarchical real-time scheduling theory into the Xen VMM. The first reason for adopting the hierarchical scheduling theory is because the two-level scheduling model proposed by this theory could be easily applied to the scheduling framework of a virtual machine system (see Figure 2.6). In the case of RT-Xen, the root level refers to the scheduling of virtual machines by the virtual machine monitor, and the second level corresponds to the scheduling of the tasks by the guest operating system. The second reason is the availability of a rich body of schedulability analysis tools that allow the formal verification of the scheduling parameters at design phase.

Four different scheduling algorithms have been implemented within Xen's scheduler, specifically, the *deferrable server*, *periodic server*, *polling server* and *sporadic server*. An empirical evaluation have been conducted and showed that the deferrable servers outperforms Xen's default Credit Scheduler. Recall that the Credit scheduler is the default scheduler in Xen, and provides a form of fair share scheduling.

In the hierarchical scheduling theory, the scheduling abstraction is called a *server*. And in the case of RT-Xen, it corresponds to a virtual CPU (vCPU). Each vCPU is characterized by a *budget*, a *period*, and a *priority*. These parameters are specified by the developer at design phase. The four server algorithms implemented by RT-Xen differ in the way the *budget* is consumed and replenished.

But all four algorithms uses the preemptive fixed-priority scheduling algorithm to select the eligible vCPU.

In the case of the deferrable server algorithm, the vCPU executes the ready tasks until either the tasks complete or the budget is exhausted. If the vCPU is idle, its budget is preserved until the next period, when it is replenished. In the periodic server, if the vCPU has no task to run its idle budget is consumed, as if it had an idle task that consumed its budget. The polling server differs from the periodic server in the way that it discards its remaining budget immediately when it has no tasks to run. And finally, the sporadic servers differ from the other in the way it is invoked. While all the other servers are invoked periodically, the sporadic server is not invoked with a fixed period, but rather, its budget is continuously replenished as it is used.

Four different measurements have been conducted to evaluate the performance of RT-XEN. The *deadline miss ratio* (DMR) was used as a metric to evaluate the performance. The DMR is equal to the total number of jobs that missed their deadlines, divided by the total number of jobs executed by one guest OS.

Three different *scheduling quantum* were used to find the most appropriate value that leads to an acceptable overhead. The scheduling quantum is the time interval at which the scheduling of a virtual machine is triggered. The values of $1ms$, $100\mu s$, and $10\mu s$, were used in the experimentation. Recall that the default scheduling quantum in Xen is $10ms$. It should be noted that while finer grained quantum results in the more precise scheduling, it also incurs a larger overhead. The results of the experiments showed that at $1\mu s$ the guest OS cannot even be booted while the $1ms$ gives a better DMR than the $100\mu s$.

In a second experimentation, the overhead of the context-switch between the virtual machines, and the scheduling latency of VMs were evaluated. The four different fixed-priority scheduling servers, the Xen Credit scheduler and the SEDF scheduler were evaluated. The results showed that the scheduling latency in the four scheduling servers is higher than the Credit and SEDF ones. The reason of this is attributed to the management of the three different queues, notably the *run queue*, *ready queue*, and *replenishment queue*, used to implement the hierarchical scheduling framework. However, the tests showed that the scheduling overhead ranges from 0.21% to 0.23% of the CPU time which demonstrates the feasibility of supporting the fixed-priority servers in a VMM. Among

the four scheduling servers, the sporadic server algorithm has the higher overhead due to its complex budget management functionality.

In a third experimentation, the impact of an overloaded domain was evaluated. The goal of this measurement is to evaluate the capability of the RT-Xen VMM to guarantee the partitioning of the CPU time between the guest operating systems. In the experiment, five guest domains were run on top of RT-Xen. One of the five domains was configured as an *overloaded domain*. The results showed that only under the Credit scheduler, the first guest domain misses almost all deadlines. The first reason for this is due to the fact that all five domains were treated equally in a round-robin fashion by the Credit Scheduler, causing the first domain to miss deadlines. Meanwhile under the fixed-priority schedulers, that domain has the the highest priority, was scheduled prior to the other, and able to meet all deadlines. The second reason is due to the fact that the first domain has the smallest period, and its tasks have the tightest deadlines, which makes it more susceptible to deadline misses. The authors argued that this observation illustrates the inability of the Credit Scheduler to provide a finer grained scheduling, and confirmed its inability to deliver real-time performance.

Another set of experiments were conducted to compare the *soft real-time* performance of different servers. In this case, five domains were configured to run with fixed budget and priority, but periods varied according to three different policies: *decreasing, even, and increasing* share ($= \frac{budget}{period}$). In the decreasing case, the first domain has the largest share and highest priority, for instance the following values: ($\frac{1}{2}, \frac{1}{5}, \frac{1}{8}, \frac{1}{10}, \frac{1}{20}$) are the shares of domains 1 to 5 respectively. In the even case, all domains have the same shares. And the increasing is the opposite of the decreasing case. The total system load was varied from 30% to 100% in a step of 5, and for each value, five real-time task sets were generated randomly and distributed among the five domains.

Again the metric of the *deadline miss ratio* was used to evaluate the performance of the different configurations. The results of the experiments showed that the Credit scheduler incurs the highest *capacity loss*. The SEDF scheduler delivers a good performance in the most cases. And the deferrable server delivers the best performance among all the RT-Xen servers. In contrast, when the system is overloaded, that is, when total system load reaches 100%, the periodic server delivers the worst performance. This is due to the fact that in a the periodic server, a VCPU continue to execute an IDLE task until its budget is completely depleted. While at the same time, lower priority domains

with positive budget could execute a waiting tasks but they are not allowed to until the IDLE vCPU exhausts its budget.

Based on the RT-Xen work, Lee et al. (2011) built the Compositional Scheduling Architecture (CSA) which is an implementation of the Compositional Scheduling Framework (CSF). In the CSF, a system consists of a set of components, where each component is composed of either a set of subcomponents or a set of tasks. A component is defined by a tuple $C = (W, \Gamma, A)$, where W (Workload) is the set of tasks, Γ is a resource interface, and A is a scheduling algorithm used to schedule W. A periodic task T_i in a component is defined by (e_i, p_i), where e_i is the worst-case execution time, and p_i is the period and deadline. The resource interface Γ is defined by the *periodic resource model* (PRM), $\Gamma = (\Theta, \Pi)$, where Π is the resource period and Θ the execution budget. The Compositional Scheduling Framework is similar to the Hierarchical Scheduling Framework. In fact, this second work complements the initial work of RT-Xen by providing two new work-conserving periodic server algorithms to improve the soft real-time performance. Moreover, it proposes a new method to select the optimal periodic resource model parameter for a given scheduling quantum for the *rate monotonic* scheduling.

A periodic resource model Γ is *optimal* for a workload W if it has the smallest *bandwidth* among all PRMs that can *feasibly* schedule W. The bandwidth of Γ is given by $\mathsf{bw}(\Gamma) = \frac{\Theta}{\Pi}$.

The minimum resource guaranteed by a PRM Γ is given by the *supply bound function* (sbf_Γ), which gives the minimum number of execution units provided by Γ over any time interval of length t, for all $t \geq 0$.

$$\mathsf{sbf}_\Gamma(t) = \begin{cases} y \cdot \Theta + max(0, t - x - y \cdot \Pi), & \text{if } t \geq \Pi - \Theta \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

where x and y are given by:

- $x = (\Pi - \Theta)$ and $y = \lfloor \frac{t}{\Pi} \rfloor$, if W is harmonic; and
- $x = 2 \cdot (\Pi - \Theta)$ and $y = \lfloor \frac{t - \Pi - \Theta}{\Pi} \rfloor$, otherwise.

Note that a workload is *harmonic* if its tasks are pairwise divisible. The resource demand of a component $C = (W, \Gamma, A)$ with $W = (T_1, T_2, \cdots, T_n)$, and A is a *rate-monotonic* algorithm, is calculated using the *request bound function* ($\mathsf{rbf}_{W,i}$) of W, given by:

$$\mathsf{rbf}_{W,i}(t) = \sum_{k \leq 1} \left\lceil \frac{t}{p_i} \right\rceil \cdot e_i, \text{ for all } 1 \leq i \leq n. \tag{2.5}$$

Figure 2.8 illustrates the shape of the $\mathsf{rbf}_{W,i}$ for a task τ_i during an interval of time $t = 5 \cdot p_i$. Between the interval $[0, p_i]$ the $\mathsf{rbf}_{W,i} = 0$, because there is no job of task τ_i that arrived. Between the interval $[p_i, 2 \cdot p_i]$ the $\mathsf{rbf}_{W,i} = e_i$ because only one job of task τ_i has arrived and requires and an execution time equals e_i. And between the interval $[2p_i, 3p_i]$ the $\mathsf{rbf}_{W,i}$ is equal to $2 \cdot e_i$ because two jobs of task τ_i have arrived and require an execution time equals to $2 \cdot e_i$. Similarly, the rest of the graph could be interpreted.

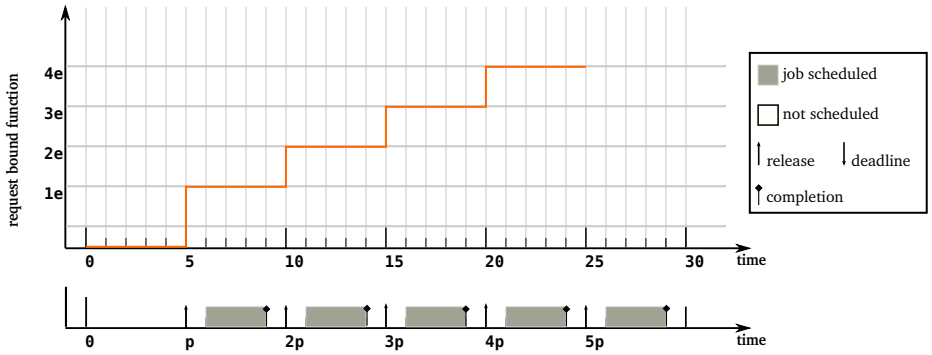


Figure 2.8: Request bound function.

Using the functions sbf_Γ and $\mathsf{rbf}_{W,i}$, the Lemma 2.1 defines the schedulability condition:

Lemma 2.1. *Given a component $C = (W, \Gamma, A)$ with $W = (T_1, T_2, \cdots, T_n)$ and $T_i = (e_i, p_i)$ for all $1 \leq i \leq n$. Then, C is schedulable if and only if,*

$$\forall 1 \leq i \leq, \exists t \in [0, p_i], \mathsf{sbf}_\Gamma(t) \geq \mathsf{rbf}_{W,i}(t)$$

Then, the necessary schedulability condition for C is $\mathsf{bw}(\Gamma) \geq \mathsf{bw}(W)$, where $\mathsf{bw}(\Gamma) = \frac{\Theta}{\Pi}$, and $\mathsf{bw}(W) = \sum_{i=1}^{n} \frac{e_i}{p_i}$. The difference, $\mathsf{bw}(\Gamma) - \mathsf{bw}(W)$, is the *resource interface overhead* of C. And, Γ is optimal for W if and only if it has the smallest interface overhead compared to all interfaces that can feasibly schedule W.

The periodic server policy is used to implement the periodic resource model ($\Gamma = (\Theta, \Pi)$) of a component. Recall that the periodic server policy used in the initial work on RT-Xen is a non-conserving work policy, that is, when a higher priority component is IDLE, it continues executing until its budget is consumed, while a lower-priority component is ready to execute a workload. In RT-Xen this is realized by running the IDLE virtual CPU while the higher priority domain idles away its budget.

To enhance the non-work-conserving aspect of the periodic server (PS), two variants have been proposed. The first is the *work-conserving periodic server* (WCPS) and the second is the *capacity reclaiming periodic server* (CRPS).

The WCPS allows a lower priority non-idle component to run if the currently running component still has a budget and is IDLE. In this case, both the budget of higher priority component and the lower priority component are consumed. And the lower priority component continue executing until, its budget is consumed, or the budget of the higher priority component is consumed, or new tasks are ready to execute in the higher priority component. In term of schedulability, this does not have any negative effects because it is only the idle budget time that is given to an another component.

Similar to the WCPS, CRPS is a work-conserving policy. The budget of a component is replenished to full capacity every period. The idle time of a component is given to another component independently from its priority, lower or higher. However, only the given budget is consumed and not the budget of the receiver like in the WCPS. In this way, a component could benefit from an extra budget, that could be used to improve its tasks' response time.

To guarantee the real-time constraints in RT-Xen, a schedulability analysis based on the Compositional Scheduling Framework theory is used. Given the period and budget of each component, a system is schedulable if and only if, all components are feasibly scheduled by the VMM's scheduler. In theory, the calculation of the budget and period of each component assume that these parameters could be assigned real values. These values are computed by iterating over the resource period from 1 to a manually chosen value, while assuming rational values for the budget. But in practice, a real system such as Xen must deal with *quantized* scheduling. For instance, the scheduling quantum in RT-Xen is $1ms$. Thus, to use this approach in RT-Xen, and given the time granularity of RT-Xen, the resource budget needs to be scaled to a multiple of the time unit, here equals to $1ms$. For exam-

ple, if the optimal PRM of a component is (0.6, 1), then after rounding up the budget we get (1, 1). However, this PRM is not optimal because, the bandwidth is equal to ($1 = \frac{1}{1}$), if the PRM is (1,1), while a PRM of (3, 4) gives a more optimal bandwidth ($0.75 = \frac{3}{4}$).

To address this issue, a method to calculate the optimal bandwidth interface of a component has been developed. It extends the compositional scheduling framework theory by fixing an upper bound on the period of the optimal PRM of a component. Theorem 2.2 computes this upper bound based on an initial feasible PRM, Γ_c, for a workload $W = \{(e_i, p_i); 1 \leq i \leq n\}$ under RM policy.

Theorem 2.2 (Lee et al. (2011)). *Suppose $\Gamma_c = (\Theta_c, \Pi_c)$ is the minimum bandwidth PRM among all PRMs that can feasibly schedule a workload W and whose period is at most Π_c. Then, the optimal PRM, $\Gamma_{opt} = (\Theta_{opt}, \Pi_{opt})$ for W, satisfies $\Pi_c \leq \Pi_{opt} \leq$ MaxResourcePeriod$(\mathcal{K}, W)$, where $\mathcal{K} = \frac{\Theta_{opt}}{\Pi_{opt}}$ and:*

$$\textit{MaxResourcePeriod}(\mathcal{K}, W) = \min_{1 \leq i \leq n} \left(\max_{t \in \textit{CrT}_{W,i}(t)} \frac{\mathcal{K} \cdot t - \textit{rbf}_{W,i}(t)}{\mathcal{K}(1 - \mathcal{K})} \right)$$

The $\mathsf{CrT}_{W,i}$ denotes the set of time-coordinates of the critical points. A critical point is a meeting point between the *upper supply bound functions* and a step-point of $\mathsf{rbf}_{W,i}$. The *upper supply bound functions* (usbf_Γ) is the minimum-slopped linear function that upper bounds sbf_Γ. The usbf_Γ of a PRM, $\Gamma = (\Theta, \Pi)$, is defined by:

$$\forall t \geq 0, \mathsf{usbf}_\Gamma(t) = max(\frac{\Theta}{\Pi}(t - (\Pi - \Theta)), 0).$$

A set of experiments allowed to compare the soft real-time performance of the three periodic server policies. These three algorithms differ in the way the idle budget is reused in the system. A set of synthetic workload has been generated and distributed among five different domains. The total system workload was varied from 0.7 to 1.0 in a step of 0.1. And three different period interval have been used, [350 ms, 650 ms], [550 ms, 650 ms] and [100 ms, 1100 ms]. The optimal periodic resource model of each domain was computed using the condition of Theorem 2.2. The *deadline miss ratio* (DMR) and the *responsiveness* ($= \frac{\text{job's response time}}{\text{job's deadline}}$) were the metrics used to evaluate the run-time performance of the tasks executed by the system.

The results of the experiments showed that the CRPS achieved the better performance comparing to PS and WCPS in terms of deadline miss ratio. This is explained by the fact that the CRPS benefits from its dual work conserving and capacity reclaiming strategy in improving the performance of low priority domains.

To validate these results, a set experiments have been conduct using real world workload from the avionics domains. The results showed that the CRPS and WCPS outperforms the PS in terms of deadline miss ratio. In terms of responsiveness, the CRPS achieves better performance over WCPS and PS in particular when the total interface overhead is high, that is, the difference between the periodic resource model given to a domain U_{RM}, and the total workload of a domain U_W, $(U_{RM} - U_W)$, is high. Conversely, if this difference is low then there is no much improvement in the responsiveness. This is explained by the fact that if the interface overhead is high means that the domain was given more resources than it really utilizes, resulting in a more available idle time. And as mentioned before, the WCPS and CRPS take their advantage from the reuse of this idle time. In the case where there is no idle time, all the policies behaves similarly.

2.3.5 Real-Time Xen-ARM

Hwang et al. (2008) ported the Xen virtual machine monitor to the ARM architecture. Xen-ARM has been adapted to be suitable for usage in mobile smart phones subject to real-time constraints (Yoo and Yoo, 2013). One of the limitation of the Xen-ARM is the size of the scheduling tick and its integer value. In the case of mobile phones, the tick size affects the overall response time, context-switch overhead and battery lifetime. Small tick size is harmful because tick interferes with CPUs power saving mode, and it incurs considerable TLB and cache flush overheads. In contrast, large scheduling tick size is bad for responsiveness because it implies longer scheduling latency. The default scheduling tick in Xen-ARM is an integer value equals to $10ms$.

The real-time scheduler in Xen SEDF (*simple earliest deadline first*) requires that each virtual machine VM_k have to be assigned a period Π_k and a budget Θ_k. The value of these parameters have to be presented with the integer number of ticks. However, existing compositional scheduling studies assume that the execution time, Θ_k, is a real number, which could not be used in Xen. Thus, the budget of a virtual machine have to be rounded up in order to be used in Xen. For example, if a VM has a budget equals to $\Theta = 17.6$, then a new $\Theta' = \lceil \Theta \rceil = 18$ must be used instead. This

additional amount of CPU bandwidth given to a VM is identified as the *quantization overhead*, and considered as a CPU wastage.

In compositional scheduling, the ratio ($\frac{\Theta}{\Pi}$) is the CPU bandwidth. The quantization overhead could be defined as follows:

$$\Delta(\Pi) = \frac{\Theta'}{\Pi} - \frac{\Theta}{\Pi} = \frac{\Theta' - \Theta}{\Pi} \tag{2.6}$$

The task set executed by a guest OS is denoted, $\tau_k = \{T_i(p_i, e_i, d_i)\}$, where each task $T_i(e_i, p_i, d_i)$ consists of an execution time, e_i, a period, p_i, and deadline, d_i. The total workload of this task set is calculated by :

$$U_W = \sum_i \frac{e_i}{p_i} \tag{2.7}$$

To ensure the *intra-VM schedulability*, the virtual machine must be allocated a periodic resource model $\Gamma_k = (\Theta_k, \Pi_k)$, which the ratio $\frac{\Theta_k}{\Pi_k}$ must be greater than the U_W:

$$\frac{\Theta_k}{\Pi_k} \geq U_W \tag{2.8}$$

The difference between the periodic resource model Γ_k reserved to a VM (VM_k), and the total workload executed by VM_k, is defined as the *abstraction overhead*, Ψ, and derived as follows:

$$\Psi(\Pi_k) = \frac{\Theta_k}{\Pi_k} - U_W \tag{2.9}$$

To find an optimal scheduling period that satisfies the two constraints (Equation (2.6) and Equation (2.9)), Seehwan Yoo and Chuk Yoo proposed a new algorithm called *SH-Quantization*. The algorithm has three input parameters, the total utilization of workload in the guest OS, U_W, the minimum period of tasks in the guest OS, P_{min}, and the intra-VM scheduling algorithm, $\mathcal{A}$. It returns a pair (Θ, Π) that is optimal with regards to the two constraints.

The idea of the algorithm is first, to calculate a set of pair (Θ, Π) that meets the intra-VM schedulability. Second, for each pair in this set, it finds the scheduling parameter Θ that has the minimum quantization overhead. Note that the algorithm searches the scheduling parameter only when the lower bound of abstraction overhead is smaller than the upper bound of the quantization overhead.

To validate the algorithm, an implementation has been proposed in Xen-ARM VMM. A new system call was introduced into Xen-ARM programming interface to allow the guest OSs to request from Xen-ARM the computation of their resource interface (Θ, Π) using the SH-Quantization algorithm at run-time. To use this system call, the real-time OS, μcOS-II, has been modified to request the service from Xen-ARM each time the input parameters $(U_W, P_{min}, \mathcal{A})$ are changed.

The implementation has been evaluated on a Freescale's imx21-ADS ARM hardware platform, containing a 266MHz ARM9 processor and 64MB memory size.

A set of experiments was conducted to validate the viability of the *SH-Quantization* algorithm with multiple virtual machines. A test with two virtual machines and a second test with three VMs were conducted, and the results showed that no deadline miss is observed in both cases.

Also a set of experiments using a real world workload from the avionics domain were conducted. The results revealed that in some cases the quantization overhead could be larger than the abstraction overhead. A second interesting remark from the results is that, depending on the workload, the gap between the total workload utilization U_W and the total CPU bandwidth that take into account the abstraction overhead and the quantization overhead could reach 45.7%. Which means that the two overheads have significant impact on actual CPU bandwidth allocation.

2.3.6 Virtualization for safety-critical system

XtartuM is a *native-VM* system designed to meet safety-critical real-time application requirements in the aerospace domain. Initially implemented on the x86 architecture, it has been ported to the LEON2 (Masmano et al., 2009), LEON3 (Masmano et al., 2010) and LEON4 (Carrascosa et al., 2013) 32-bit processors compliant with the Sparc V8 ISA (instruction set architecture). The port to the LEON3 processor enabled the implementation of full spatial isolation through the use of the MMU[8]. While the precedent version provided only read-only memory access to the partition[9], which reported as very problematic in the case of a trap raised by a partition when it is trying to write in write-protected memory area. This trap is received several cycles later, which complicated the emulating of the instruction and finding the offending partition, while an MMU trap is synchronous. The port to the LEON4 added the support of SMP multicore to XtartuM.

[8]Memory Management Unit.

[9]In the context of XtartuM a partition is equivalent to a virtual machine.

XtartuM uses *paravirtualization* and dedicated device techniques to allow a modified guest operating system to access the hardware platform. Paravirtualization is a technique common to many virtual machine monitors, it allows a modified guest OS to request services from the VMM through a set of special system calls, referred to as *hyper-calls*. In the case of XtartuM, *privileged instructions* of a guest OS are replaced by these hyper-calls, for instance to enable or disable the interrupts and to use a virtual timer device.

The use of paravirtualization is due to the fact that the Sparc V8 ISA provides only two privilege levels, user and supervisor mode. This prevents a guest operating system to run correctly because it is executed at user mode and not at supervisor mode in order for XtartuM to guarantee the spatial isolation. Thus the guest operating system needs to be ported on top of XtartuM.

To enforce the temporal isolation between the partitions, XtartuM implements a cyclic scheduling policy as recommended by the ARINC 653 specification. Each cycle is divided into slots and each slot is allocated to a partition. The duration of a slot given to a partition is defined statically at design time. The cycle is then repeated periodically. The VMM ensures that a partition starts at a specified time and runs for a specified amount of time slot. Each partition schedules its internal tasks using its own policy.

To improve the responsiveness of a partition XtartuM allows the direct management of interrupt of non critical devices by the partition itself. Furthermore, each partition is given a priority to allow the VMM to prioritize the events and interrupts directed to that partition. The interrupt destinated to high priority partitions are treated before the one of the low priority partitions. In addition, XtartuM assigns the system resources such as memory, I/O registers, and devices to specific partitions. In order to reduce design complexity and increase the reliability of the implementation, the VMM is designed as a monolithic, non-preemptable kernel. The authors argued that this restriction does not impact the performance of a small VMM implemented using simple and fast code.

A development board including a LEON3 50MHz with 128MB of RAM and 16MB of flash PROM has been used to evaluate the performance of XtartuM. The idea of the experiment is to measure the performance loss due to partition context-switches performed by XtartuM. This overhead is the time needed to stop the execution of a partition and to resume the next partition in the scheduling plan.

The scenario consists of several bare partitions that increase an integer counter and a (reader) partition that reads the counter values of the other partitions. This partition is executed in last slot in the cycle with enough time to print all counter values through the serial port.

To measure the duration of the context-switch one tracing-point was inserted in the XtartuM kernel before and after the context-switch. The results showed that the overhead of a context-switch was equal to $100\mu s$ in average and $116\mu s$ in maximum.

However, repeating the same scenario and increasing the number of partition context-switches did not affect the individual performance of a partition. That is, the difference between 3 context-switches and 150 context-switches for the same duration of the experiment results in only 0.8% of performance loss.

The authors argued that the support of MMU in the LEON3 is beneficial to the VMM even if it adds more CPU cycles to the overhead in comparison to the LEON2, due to the translation from virtual to physical space. And this though the presence of a TLB[10] which mitigates this effect. And the flush of the whole TLB is avoided at a context-switch due to the context tag provided by the Sparc V8 implementation. But due to the small size of LEON3's TLB, the size of the memory page must be selected carefully. For instance, as the LEON3's TLB has 32 entries, it covers only to 128KB of memory if the page size is 4KB (4KB*32), which could generate some TLB miss and therefore increase the overhead. However, the authors indicated that the overhead of a page fault is not considered since there is no page fault.

The port of XtartuM to multicore processor modified many of its properties. The major modification concerned the implementation of a fine-grained synchronization mechanism that grants exclusive access to the critical sections of XtartuM by protecting shared data structures through spin-locks to avoid the race conditions.

The second major modification concern the scheduling, which however is independent from the port to multicore version, is the use of fixed priority scheduling. This is because in cyclic scheduling, each core has its own cyclic plan, and this could cause a delay problem when handling asynchronous interrupt. Because an interrupt allocated to one partition may stay pending until the partition is scheduled.

[10]Translation Lookaside Buffer.

To evaluate the multicore implementation of XtartuM, a quad-core 32-bits LEON4 50MHz processor with (4*4KB) instruction and data L1-cache, and a shared 256KB L2-cache, an MMU, an ioMMU, and two shared FPUs has been used.

The Dhrystone and CoreMark benchmarks were used to evaluate the performance of XtartuM on the multicore. The comparison of the benchmarks executed natively on the hardware and on a partition revealed that the performance loss is about 1% for the CoreMark. In the case of Dhrystone, the performance loss was negligible because this benchmark executes only simple integer operation and do not require the intervention of the VMM.

While the CoreMark test executes a set of algorithms that generate almost 2235 stack window overflow and underflow that cause traps which are handled by the VMM. And the handling of a trap forces the VMM to perform a switch context. Note that in the experiment the partition running the benchmark was allocated 30 $second$ of time slot, which is sufficient to complete the execution of the test without preempting the partition.

A second test using the CoreMark benchmark and varying the partition time slot from 30 $second$ to $1000ms$, $500ms$, $100ms$ and to $10ms$ revealed that the performance loss increase with the decrease of the time slot duration. At $10ms$ time slot the performance loss reaches 2.5%. This is attributed to the overhead of context-switch at the end of each slot, which is estimated to $151\mu s$.

In comparison with our precedent analysis of kvm and microkernel-based virtual machine systems, the review of XtartuM clarified multiple points regarding the overhead induced by the virtualization. We believe that the simple memory management model and a set of simple test cases is essential to understand the overhead of virtualization because the degree of uncertainty regarding the source of overheads increase significantly with the complexity of the hardware, the software implementation, the guest OS and the test cases. In the following paragraph, we continue our review of another simple virtual machine system similar to XtartuM.

Tavares et al. (2012) developed a *native-VM* system compliant with the ARINC 653 standard and targeting an aerospace application. It was implemented on the PowerPC 405 processor embedded in a Xilinx FPGA. The temporal isolation between virtual machine is guaranteed through the use of a real-time scheduler that allocates to each VM a fixed time slot. Spatial isolation is realized by running the guest operating system at user-mode and the VMM at privileged mode of the proces-

sor, and through the use of the Memory Management Unit to control the virtual to physical memory mapping of the guests.

Traditionally, when an operating system is supposed to run at privileged mode because it uses a set of privileged instruction to control the hardware and access all the registers. However, by running the operating system in user-mode, will force every execution of a privileged instruction by this guest OS to generate an exception. This exception is captured by the VMM and emulated. The VMM uses the user privilege bit, the instruction address translation bit, and data address translation bit in the Machine State Register of the processor to determine what is the original processor mode of the code that generates the exception. If the code that generates the exception was supposed to run in user mode then the exception is forwarded to guest operating system. And if the code that generates the exception was supposed to run at privileged mode, that is, the exception was generated by the kernel of the guest OS then the instruction is decoded and inspected to determine the operation. The operation is then executed by the VMM on behalf of the guest OS and the result is delivered to the VM if it is still schedulable, otherwise it will be delivered the next time the VM is activated.

The VMM intercepts all address space updates and emulates them because the guest operating systems are not allowed to use the Memory Management Unit. The implementation of the VMM benefits from the PowerPC 405 software-managed and tagged TLB to separate VM's address spaces and virtualize the MMU.

The VMM reserves statically at compilation time for each VM an address space in the real physical memory. Specifically, the VMM uses the first 64KB page of the physical address space (0x00000000 - 0x0000FFFF), and then, it allocates for the first VM the first 64KB page of the second 16MB page which corresponds to the physical address space (0x01000000 - 0x0100FFFF), but the VM is allowed to access the whole 16MB page real addresses (0x01000000 - 0x01FFFFFF). Similarly, it allocates for the second VM the physical address space (0x02000000 - 0x02FFFFFF), and for the third VM the physical memory address space (0x03000000 - 0x03FFFFFF).

Then using a set of mapping operations, the VMM translates the real addresses of a VM into virtual addresses of the VMM if the VM is running in real mode, or the virtual addresses of a VM into virtual addresses of the VMM if the VM is running in virtual mode.

Using the TID bit in the TLB register to tag each TLB entry, the VMM avoids to flush all the TLB when performing VM context-switch. The VMM also provides a hypercall to the guest operating systems to request an MMU update and minimize the number of VM exit/entry to and from the VMM. However, this optimization forces the modification of the guest OS source code.

The evaluation of the VMM showed that the most expensive operation is the saving of the virtual machine CPU state, which was estimated to 2963 cycles, followed by the programmable interrupt timer which is equal to 1393 cycles. The instruction decoding overhead is equal to 351 cycles.

Researchers from the real-time system group at the Commissariat à l'Energie Atomique (CEA-List) developed PharOS (Lemerre et al., 2011) a real-time operating system for mixed-criticality system targeting the automotive domain. PharOS combines two real-time programming paradigms: the *time-triggered* and the *event-triggered* models. The time-triggered tasks are defined using the time-constrained automata model to specify their temporal requirements. Temporal isolation among time-triggered tasks is then ensured by PharOS kernel using these informations. The isolation between time-triggered and event-triggered tasks is realized by running the tasks from the two models onto separate cores of the hardware, that is, a set of cores is reserved to time-triggered tasks and the other set is dedicated to the event-triggered tasks. The spatial isolation between the tasks was ensured by associating for each task a memory context and protecting each context using a hardware memory protection unit.

PharOS was also used as a virtual machine monitor to build an automotive mixed-criticality system. The Trampoline RTOS was used as a para-virtualized guest on top of PharOS. Trampoline (Bechennec et al., 2006) is an RTOS compliant with the OSEK/VDX automotive standard for software development. Trampoline was modified in order to replace its privileged instructions by a set of hyper-calls to PharOS. The Trampoline guest was encapsulated in a time-triggered task that is executed periodically every $10ms$ and given a 10% of the CPU resource. For each release of the Trampoline task, the PharOS emulates a timer interrupt to signal the scheduling tick for Trampoline.

The prototype was evaluated using two real-time applications, a first set of six critical tasks controlling some system commands, a CAN bus communication and a sensor signal mechanisms were implemented directly on PharOS, and a second set of tasks implementing a diagnostic and aliveness monitoring function was implemented on top of Trampoline.

A first test consists of verifying the spatial isolation between Trampoline and PharOS by making a memory read/write operation from Trampoline tasks to protected memory in PharOS. The test showed that the error was detected and the Trampoline task was restarted. A second test consists of verifying the temporal isolation by inserting an infinite loop into a Trampoline task and showing that this erroneous behavior did not affect the other tasks of PharOS. However, no overheads or latencies measurement was conducted using this prototype.

Meanwhile, in a discussion with our industrial colleagues, they were interested in developing a solution for automotive application using the Trampoline as an AUTOSAR compliant RTOS and Linux-Genivi as an infotainment OS, but using POK as a virtual machine monitor. Although, they were aware that Trampoline was already para-virtualized on top of PharOS, they suggested to investigate POK as a virtual machine monitor. POK (Delange and Lec, 2011) is a real-time operating system compliant with the ARINC 653 avionic standard. This property allows it to securely co-locate multiple applications on the same processor. Note that these applications were previously deployed on separate hardware. POK already provides time and space isolation through the use of partitions. Each partition is allocated a certain amount of CPU resource as if it was running on a dedicated processor, and is given a unique memory segment that is protected from the other partitions. If we suppose that a partition contains a guest operating system and its application, POK could be considered as virtual machine monitor similar to XtratuM. Obviously, this requires that the guest OS need to be para-virtualized using POK API, or extending POK to implement the new hardware assisted virtualization mechanisms to support unmodified guests.

Summary. The review of these studies regarding the existing virtual machine system aimed to understand how these systems have been designed and adapted to respect the real-time system requirements.

From the discussed studies we observed that the real-time issues in a virtual machine system were treated from two main perspectives, first from a scheduling theory perspective, and second from an implementation perspective.

From the scheduling perspective, the researchers were aware that there is an overhead when using a virtual machine system to build a real-time system. But they were more concerned about building new theoretical tools that take into account this overhead and allow to build more efficient real-time virtual machine systems.

Specifically, the Hierarchical Scheduling Framework was a common solution adopted by multiple studies. The HSF is a design that decomposes the scheduling into two levels, one global-level for scheduling virtual machines, and a second local-level for scheduling real-time tasks inside each virtual machine. The HSF benefits from a solid theoretical background that gives the system designer a set of analytical tool to verify the correctness of the system. The experimentation of the HSF in practice showed the effectiveness of such a design in improving the real-time performance of a virtual machine system.

From an implementation perspective, the studies used simple hardware mechanisms, such as simple memory management model, and simple software implementation of the VMM, and were more focused on low-level overhead measurement without paying enough attention to real-time performance at application level. We argued that this perspective is very useful because in a virtual machine system built upon complex hardware mechanisms, and a complex software implementation, there is a lot of uncertainty about the source of the overhead.

However, we believe that both approaches are necessary and we aimed at combining these two approaches to help reducing the degree of uncertainty about the source of virtualization overhead.

In our work, we initially treated the problem of adapting a virtual machine system to real-time system from a practical perspective. We were more concerned about virtualization overhead and its impact on real-time properties. Our assumption is: if the current and the future hardware architectures allow to run a guest operating system without any modification to its source code, then we should be able to obtain the same performance as if the operating system was running on a real hardware. Because the guest operating system is supposed to run on a virtual hardware at the same speed rate as if it was running on a real hardware. And if this assumption is not verified, then our question is: what are the hardware mechanisms and the software implementation that prevent the guest operating system from achieving the required real-time performances?

The idea behind this question comes from the fact that we are considering the problem from an operating system developer perspective. As an OS developer, our main objective is to support as much applications as possible, thus, if we are able to state that the guest OS that is running on a virtual hardware present the same characteristics as on a real hardware, then we can state that the guest OS is able to support the same range of application that it is able to support as if it was running on a real hardware. Otherwise, we have to evaluate the degree of performance decrease.

Therefore, in our work we are more concerned about evaluating the fine-grained internal overheads and latencies of an RTOS. Because the application performance depends on these fine-grained overheads and latencies.

While paravirtualization could be an efficient solution to reduce the virtualization overhead, we do not consider it in our evaluation of the virtual machine system, because we neither want to depend on the availability of source code for the guest operating systems nor make the extra effort of porting operating systems to a paravirtualization interface especially when architectures such as Intel x86, ARM, and PowerPC provide hardware virtualization extensions that allow to avoid such a porting effort.

We are more interested in using overhead-aware scheduling algorithm based on solid theoretical foundation to resolve the real-time issue in a virtual machine system.

In the next chapter, we present our methodology to evaluate a virtual machine system. We first define the hardware and the software mechanisms required to build an efficient real-time virtual machine system then we present the results of the evaluation.

2.4 RTOS Configuration

In this section, we review most relevant work with respect to RTOS configuration. By configuration we mean the adaptation of the RTOS internal resource allocation techniques such as the scheduling and synchronization mechanisms to the requirements of the supported application.

2.4.1 Composite

The Composite component-based OS (Parmer, 2010) is a research operating system focused on reliability, predictability, and configuration. The configuration is enabled by the use of user-level components, where each component is independent from the others, and interacts with them through a contractually-specified interface. Each component defines one specific functionality.

Composite uses user-level components to implement scheduling policy, memory management, and synchronization. The decoupling of the component's interface from its implementation enables those policies to be changed and used in a system that provides behavior adapted towards the application's goals.

The communication between components relies on a set of optimized Inter-Process Communications. The invocation of functions in a component interface involves two system calls and switching between two protection domains[11], and back.

A prototype of Composite has been implemented using the Hijack$^{\text{COS}}_{\text{Linux}}$ (Parmer et al., 2012) module. Hijack$^{\text{COS}}_{\text{Linux}}$ is an interposition layer inserted between the hardware and each OS to multiplex the hardware between them. In such design, Linux acts as the host OS, and manages the resources that cannot be shared such as device drivers. Hijack$^{\text{COS}}_{\text{Linux}}$ intercepts the kernel entry points, and multiplexes hardware events to either Linux or Composite. Hijack$^{\text{COS}}_{\text{Linux}}$ is implemented as Linux kernel module, and Composite is executed as the highest priority task in Linux. Figure 2.9 illustrates the overall architecture of Hijack$^{\text{COS}}_{\text{Linux}}$.

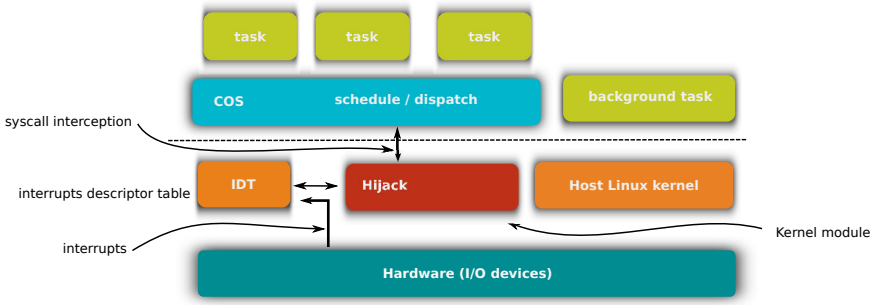


Figure 2.9: Schematic of the overall architecture of Hijack$^{\text{COS}}_{\text{Linux}}$.

Inside Hijack$^{\text{COS}}_{\text{Linux}}$ there is a Hardware Abstraction Layer responsible for the page-table and physical memory management. It also provides facilities to retrieve information about the timer interrupts (e.g. frequency), and specify the handler function. It allows Composite to notify Linux if there is no current activity.

The second major role of Hijack$^{\text{COS}}_{\text{Linux}}$ is to multiplex between Composite and Linux. In particular, Hijack intercepts the system calls, interrupts, and exceptions, then it dispatch them to the appropriate handler. Depending on the OS that a user-process that generated the event belongs to, Hijack$^{\text{COS}}_{\text{Linux}}$ decides where to redirect it. For instance, a system call executed by a Linux process is redirected to the Linux kernel, otherwise it is dispatched to the Composite kernel.

The reason of co-locating the Composite OS and the Linux is to benefit from the support for the architectures and device drivers provided by Linux. Moreover, using the configuration of Composite

[11] A protection domain is equivalent to an address space.

and co-locating it with Linux, allows to build new system requiring adaptation that could be difficult to integrate into Linux.

Dividing the system into components provides the opportunity for increased system fault isolation as each component is placed into its own hardware-provided protection domain at user-level. A fault due to malicious or erroneous code in any component is prevented from trivially propagating and corrupting the memory of other components or the trusted kernel.

One notable limitation in this approach is the expensiveness of the switching between hardware protection domains in comparison with function call. This is mainly due to overheads in crossing between protection levels, and the necessary invalidation of hardware caches. Communication between separate protection domains requires these switches and imposes significant overhead on the system. This overhead prevents some applications from meeting performance or predictability constraints, depending on the inter-component communication patterns and overheads of the system.

The Mutable Protection Domains (Parmer et al., 2012) was proposed as a solution to dynamically leverage the trade-off between the granularity of fault isolation, and the performance of the system. Mutable Protection Domains allows protection boundaries to be placed and removed dynamically as the performance bottlenecks of the system change. When there are large communications overheads between two components due to protection domain switches, the protection domain boundary is removed, if necessary. In areas of the system where protection domain boundaries have been removed, but there is little inter-component communication, boundaries are re-installed.

2.4.2 ExSched

The ExSched framework (Åsberg et al., 2012) is an approach to easily support new multicore scheduling algorithms into Linux without modifying the kernel. The authors argued that adopting an approach based on non-intrusive solution will benefit to academia and industry. Easier installation of frameworks and schedulers on various software platforms could increase the re-usability of already implemented solutions in academia. In the industry, this would make it easier to update to new kernel versions since loadable kernel-modules require much less (or no) kernel modifications compared to patches.

The ExSched framework provides a set of services to implement different schedulers as external plug-ins for different OS platforms. It also provides an API (Application Programming Interface)

for user programs. Neither scheduler plug-ins nor user programs will access OS native functions. The core component of ExSched is a kernel-space module that controls the CPU scheduler via scheduler-related functions exported by the underlying OS.

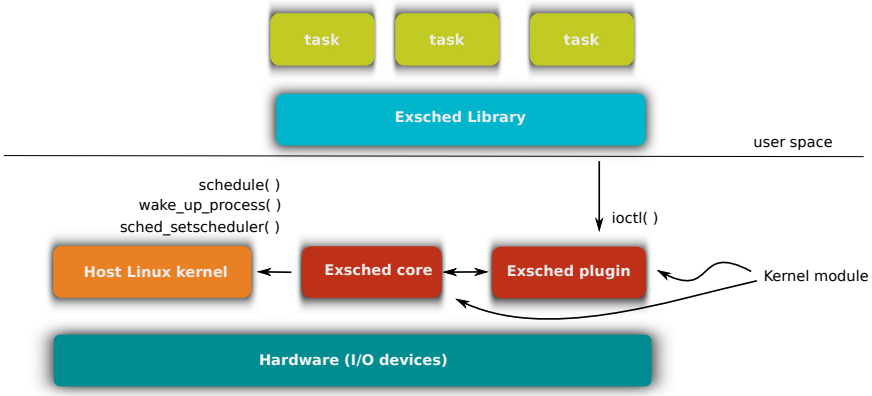


Figure 2.10: Schematic of the overall architecture of ExSched.

As an example, to switch between tasks, to migrate tasks to other cores, and to change the priorities of tasks, ExSched relies on the primitives of the underlying OS. The ExSched core is built as a character device and its installation creates an accessible device file /dev/exsched. The scheduler plug-ins request the ExSched kernel module using the ioctl( ) system call. In return, the ExSched core calls back the scheduler plug-ins using an appropriate set of functions implemented in the plug-ins. Figure 2.10 depicts the overall architecture of ExSched.

The ExSched framework has been implemented in the Linux kernel and in VxWorks. In Linux, ExSched uses a real-time scheduling class, rt_sched_class to isolate real-time tasks from non-real-time tasks. The non-real-time tasks are scheduled by the fair share scheduling class, fair_sched_class, in Linux.

The implementation of the ExSched core module relies on the primitives functions exported by the underling OS platform. In Linux, it uses the schedule( ) function to switch between the current and the highest-priority ready task. It uses also the sched_setscheduler(task, policy, prio) function to set the scheduling policy and the priority of the task, and the setup_timer(timer, func, arg) function is used to associate the timer object with a given function and its argument.

The VxWorks implementation of ExSched do not differ much from Linux except that there is no need to ioctl( ) calls because VxWorks does not support user-space mode. Furthermore, there is no need for scheduling class because all tasks in VxWorks are real-time tasks.

The ExSched framework has been used to develop six different schedulers, two are hierarchical schedulers and four of them targeted the multicore scheduling algorithms (Åsberg et al., 2012). The two hierarchical scheduling plug-ins in ExSched differ in the policy employed at the global scheduling level, one plug in uses the EDF and the second uses the FP policy.

ExSched comes with two global multicore scheduling plug-ins. The G-FP (*global fixed-priority*) scheduler selects the highest priority tasks and dispatch them in global scheduling fashion on the available cores. The FP-US classifies tasks as heavy and light tasks based on the CPU utilization factors. The heavy tasks are statically assigned the highest priorities, and light tasks are not changed. ExSched also provides two partitioned scheduling plug-ins. The FP-FF uses a *fixed-priority first-fit* heuristic to assign tasks to CPUs. And the FP-PM is a *semi-partitioned* scheduling that migrates tasks across multiple CPUs if the tasks cannot be assigned to any CPU by a first-fit allocation.

The results of the overhead measurement of the implemented multicore scheduling algorithms as ExSched plug-ins validate the theoretical assumption of the corresponding algorithms.

One limitation of the ExSched framework raises when the underlying operating system do not provide such primitives functions. For instance, the schedule( ) function or a similar primitive is not available by default in every operating system. This prevents from easily porting and reusing the ExSched framework. A second limitation of the Exsched is that core module is executed at kernel-space, this represents a risk because a fault in the module is not isolated from the rest of the kernel and could crash the whole system.

2.4.3 LITMUS$^{\mathrm{RT}}$

LITMUS$^{\mathrm{RT}}$ (Brandenburg, 2011) is a native real-time Linux kernel, developed essentially to explore the implementation of the state-of-the-art multiprocessor scheduling algorithms and synchronization protocols.

The software architecture of LITMUS$^{\mathrm{RT}}$ is almost similar to the architecture of ExSched. LITMUS$^{\mathrm{RT}}$ is composed of four parts: the core infrastructure, a set of scheduler plug-ins, the user-space interface, and the user-space library. The core infrastructure is the connection layer between scheduler plug-ins and the Linux scheduling hierarchy, as shown in Figure 2.11.

From one side, the core infrastructure is integrated in the Linux kernel by implementing the Linux scheduling interface consisting of 22 functions. From the other side, the core interface provides an interface to the scheduler plug-ins.

Whenever a scheduling decision is required, the Linux scheduler uses the set of its interface functions to request LITMUS$^{\mathrm{RT}}$ which process to dispatch, LITMUS$^{\mathrm{RT}}$ in its turn, invokes the functions of its interface with the active scheduler plug-in to select the highest priority real-time task to schedule.

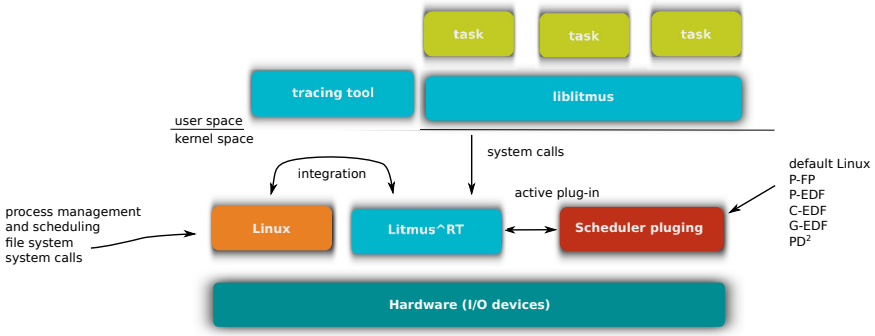


Figure 2.11: Schematic of the overall architecture of LITMUS$^{\mathrm{RT}}$.

This layer of indirection between the scheduler plug-ins and Linux is required to factor the common functionality of multiple plug-ins, such as migration mechanisms, tracing, and debugging. It also avoids to change every scheduler plug-in each time the Linux scheduling interface changes because this interface changes frequently between versions.

The user-space interface and library are useful to program real-time application and to configure the needed scheduler plug-in. Since the real-time tasks are regular Linux processes, the user-space library provides system calls to create the real-time tasks by specifying the task execution time and period. It also allows to create, lock, and unlock the real-time semaphores.

The flexibility of the LITMUS$^{\mathrm{RT}}$ infrastructure allows to implement multiple scheduling policies. Global scheduling policy such as the G-FP (*global fixed priority*) and the G-EDF (*global earliest deadline first*) have been developed. In global scheduling, the tasks are allowed to migrate between processors at runtime. The P-FP (*partitioned fixed priority*) and the P-EDF (*partitioned earliest deadline first*) scheduling algorithms have been also implemented. In the case of partitioned scheduling, tasks are statically mapped to processor and never migrate.

Moreover, *clustered* scheduling algorithm has been integrated. This hybrid scheduling policy combines global and partitioned policies. Here, a cluster is a set of CPU cores that are grouped together according to their cache hierarchy. The task set is then partitioned into multiple task subsets. Each task subset is associated to a cluster, and tasks migrate among the cores within the cluster. Nevertheless, they are not authorized to migrate to core in other clusters.

The LITMUS$^{\mathrm{RT}}$ user can simply change the scheduler at runtime using the tool provided by the user-space library. In comparison with ExSched, the LITMUS$^{\mathrm{RT}}$ core infrastructure requires the source code modification (patch) of the Linux kernel. In contrast, ExSched core infrastructure is implemented as a module, that could be integrated into Linux without any need to patch or to compile the kernel.

While LITMUS$^{\mathrm{RT}}$ offers an efficient and valuable platform to test new scheduling algorithms and synchronization protocols, it remains a research operating system.

2.4.4 Microkernel

The microkernel design principles recommend to implement in the kernel only the necessary abstractions to build a complete operating system. The necessary abstractions are, the address spaces, the threads, and the inter-process communication. The rest of the ”primitives” such as, memory management, networking, file system, device drivers, paging, and more, should be implemented by servers outside the kernel. Nowadays, commercial products based on microkernel design approach exists. Microkernel-based RTOS such as QNX Neutrino and PikeOS have been certified to be used in real-time safety-critical systems. Their wide adoption in domains such as avionics, automotive, medical devices, and military system validates the efficiency of their design approach.

Among the advantages of such a design, we can mention the flexibility and extensibility offered to the system. It is possible to easily and effectively adapt it to new hardware or new applications. Only small set of servers need to be modified or added to the system, and this without affecting the correctness of the already developed kernel and other servers. Another advantage concerns the safety of the system. Erroneous functionality in servers are isolated as normal application malfunction.

With respect to our configuration requirement, the microkernel approach offers an interesting opportunity to integrate new strategies to the operating system. For example, by implementing

new different scheduling policies in different user-level servers. It is possible to select the needed resources allocation strategy just by including in the final system the server that implements the strategy without any need to modify the kernel.

Based on the advantages that offer this design in terms of general flexibility and power, we decided to use a microkernel-based OS as an implementation platform for our second requirement, which is the configuration of the operating system without modifying its internal structure and implementation.

2.4.5 OveRSoC RTOS Model

In the OveRSoC project (Miramond et al., 2009), a real-time operating system for Reconfigurable System-on-Chip platform has been designed. The RTOS and the RSoC hardware platform have been implemented using SystemC (Accellera, 2014) system-level simulation language.

The OveRSoC component-based RTOS model offer an easy way to configure the operating system by allowing the user to change each component independently from the other components. Using the OveRSoC RTOS model, industry practitioners would benefit from the ability to select and deploy resource allocation techniques commensurate with their particular applications.

The results of the experiments we made using the OveRSoC RTOS model, lead us to propose a transformation from the simulation model into an executable model, in order to run the RTOS on a real hardware.

The specification of our work is to transform the OveRSoC RTOS model into an executable model, and to preserve the configuration characteristic offered by its design.

To implement this specification we proposed a two-step approach. First, the OveRSoC RTOS model could be transformed into executable programs on a real hardware by implementing some of its components simply using some modules from an existing real-time operating system. More precisely, by reusing functionalities such as the boot-loader, the process abstraction, the memory management service, and the drivers from an existing RTOS. Second, in order to preserve the customization property of the OveRSoC RTOS model, a middleware could be deployed on top of the RTOS in order to implement the customized functionalities from the OveRSoC OS model that are not supported natively by the RTOS.

We will detail this idea in the Chapters 5 and 6. In Chapter 5 we explain how the OveRSoC RTOS model could be transformed automatically into an executable model. Then, in the Chapter 6 we present the implementation of the middleware on top of an existing operating system.

Summary. The leader of the software team at SpaceX (Rose, 2013), a company that build space vehicles such as the Falcon, Dragon and Grasshopper vehicles used by the NASA, declared that the Linux operating system was embedded in the space vehicles to control the flight software and put the spacecraft into orbit. Linux is also used to control ground station and by the developers to build the software.

The team leader also mentioned that using Linux to build safety-critical software does not mean that they use an ”off-the-shelf distribution kernel”. Instead, they spend a lot of time evaluating a kernel for their needs, and one of the area they focus on is scheduler performance and wake up latency. For instance, they stress the network and test the scheduler performance. However, the developer declared that once a kernel is chosen ”they try not to change it”.

The leader of the PREEMPT_RT project (Gleixner, 2013) announced that in the future there will be two options for the real-time Linux patch, whether the 100% of the patch gets integrated into the mainline Linux kernel, or to decide that the 95% of the real-time work already upstream is sufficient and to drop further efforts. Unfortunately, the later option is a serious problem for the future of a real-time Linux. The reason for this decision is attributed to the fact that updating the 5% of the patch for each new kernel release is no longer acceptable, and making the rest of the code ready for mainline requires more of an effort from a wider group than is currently involved.

The PREEMPT_RT was started by RedHat and IBM after obtaining a contract from the US Navy. Today, the patch is essentially maintained by permanent engineer paid by RedHat and some developers from the community. According to the project leader, this is not sufficient and the whole problem is the lack of permanent developer and contribution from companies such as Wind River and Intel, that uses the real-time patch.

Researchers at the real-time system group at the Scuola Superiore Sant’Anna in Piza and engineers from the Evidence Company developed SCHED_DEADLINE (Faggioli et al., 2009), a patch to the Linux kernel that implements the EDF real-time scheduling algorithm. The research work started in 2009, was then maintained by researchers (Lelli et al., 2011) until it has been recently

merged into the mainline Linux kernel (LWN, 2014). This success rewards the tremendous research effort spent in developing this Linux patch.

These examples illustrate the main problems encountered in practice when new real-time functionality and features need to be integrated into an operating system kernel. In one case, modifying a kernel to integrate a new scheduling algorithm or any other techniques is not considered as a good idea from an industrial perspective. In a second case, maintaining a patch updated with each new release of the kernel could be, at long term, not a good plan. In a third case, the longevity of an effort to mainline a single scheduling algorithm demonstrated the difficulty of integrating the advance in the real-time research theory into practice.

We believe that using a middleware to deploy a new scheduling and synchronization techniques on a real-time operating system is more effective in overcoming the problem of integrating the real-time theory advances in practice. Through the use of a middleware, it is possible to implement new resource allocation techniques at user-level, thus avoiding the modification of the kernel, and the problems related to the kernel-level approach.

In Chapter 5, we will present a model-driven engineering technique to automatically transform an RTOS model into a source code that is executable on a real hardware. Next in Chapter 6, we present a prototype to demonstrate the feasibility of the configuration of an operating system without the modification of its kernel through the use of middleware.

CHAPTER 3

Virtualization and Real-Time Systems

In this chapter, we evaluate the ability of a virtual machine system to co-locate a real-time operating system and a general-purpose operating system. This will allow us to answer the question of: what is the overhead of virtualization on a guest RTOS?

Before presenting the evaluation, we define the mechanisms provided by the hardware architecture to build a virtual machine system, and discuss how these elements are involved in the virtualization overhead. First we measure the overall overhead, then we decompose this overall overhead into a set of finer-grained overheads and latencies. Next, we analyze the results of the evaluation and we discuss how a set of hardware and software mechanisms need to be improved in order to reduce the virtualization overhead.

3.1 Hardware-Assisted Virtualization

In order to understand the evaluation of a virtual machine system, it is essential to understand first the hardware mechanisms required to build such a system (see Figure 3.1). In this section, we present the efficient virtualization of a processor.

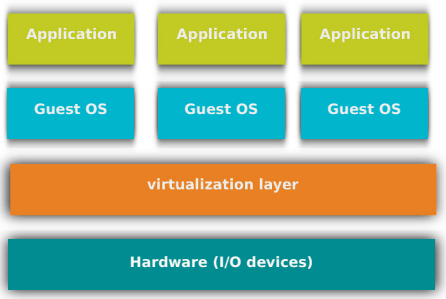


Figure 3.1: Virtual Machine System Concept.

3.1.1 Resource Virtualization - Processors

There are two ways of virtualizing a processor. The first is emulation, and the second is direct native execution on the host machine. Emulation involves examining each guest instruction in turn, and emulating on virtualized resources the exact actions that would have been performed on real resources. Emulation is the only processor virtualization mechanism available when the ISA (instruction set architecture) of the guest program is different from the ISA of the host machine.

The second processor virtualization method uses direct native execution on the host machine. This method is possible only if the ISA of the host machine is similar to the ISA of the guest program. In this case, the guest program will often run on a virtual machine at about the same speed as on a native hardware, unless there are memory or I/O resource limitations. The overhead of emulating any remaining instructions depends on several factors, including the actual number of instructions that must be emulated, the complexity of discovering the instructions that must be emulated and the data structures and algorithms used for emulation.

3.1.1.1 Conditions for ISA Virtualization

In a virtual machine environment, an operating system running on a guest virtual machine should not be allowed to change hardware resources in a way that affects the other virtual machines. Hence, even the operating system on a virtual machine must execute in a mode that disables the direct modifications of system resources (for example the CPU timer interval). Consequently, all of the guest operating system software is forced to execute in user mode. This represents a problem that prevents the construction of an efficient virtual machine monitor. But before explaining the reason of this problem we need to define two terms.

Sensitive instruction. A sensitive instruction is an instruction that attempts to read or change the resource-related registers and memory locations in the system, for example, the physical memory assigned to a program. The POPF, Intel IA-32 instruction is an example. This instruction pops a word from the top of a stack in memory, increments the stack pointer by 2, and stores the value in the lower 16 bits of the EFLAGS register. One of the bits in the EFLAGS register is IF, the interrupt-enable flag that is not modified when POPF is executed in user mode. The interrupt-enable flag can only be modified in privileged mode.

Privileged instruction. A privileged instruction is defined as one that traps if the machine is in user mode and does not trap if the machine is in kernel mode.

The reason why a VMM could not be constructed efficiently is due to the fact that if a sensitive instruction such as POPF is executed by the guest operating system, and that this guest OS is running in user mode, this instruction will not trap. So the VMM could not take control of the machine and execute on behalf of the guest OS. The only way to force the control back to the VMM, is the use of emulation. It would be possible for a VMM to intercept POPF and other sensitive instructions if all guest software were intercepted instruction by instruction. The VMM could then examine the action desired by the virtual machine that issued the sensitive instruction and reformulate the request in the context of the virtual machine system as a whole. The use of interpretation clearly leads to inefficiency, in particular when the frequency of sensitive instructions requiring interpretation is relatively high.

To avoid this problem, it is necessary for an ISA to be efficiently virtualizable that all the sensitive instructions are a subset of the privileged instructions (Popek and Goldberg, 1974). More precisely, if a sensitive instruction is a privileged instruction, then it will always trap when executed in user mode. All non-privileged instructions can be executed natively on the host platform and no emulation is required.

3.1.1.2 Intel Virtualization Extension

To enhance the performance of virtual machine implementations, hardware manufacturers developed a dedicated technology for their processors. The main feature is the inclusion of a new processor operating mode. For example, the Intel VT-x feature has added a new processor mode called VMX. In this mode, the processor can be in either VMX *root operation* or VMX *non root operation*. In both cases, all four IA-32 privilege levels (rings) are available for software. In addition to the usual four rings, VT-x, provides four new less privileged rings of protection for the execution of guest software, as shown in Figure 3.2.

The processor in the VMX root operation behaves similarly to a normal processor without the VT-x technology. The main difference relies in the addition of a set of new VMX instructions.

The behavior of the processor in a non-root operation is limited in some respects. The limitations are such that critical shared resources are kept under the control of a monitor running in VMX

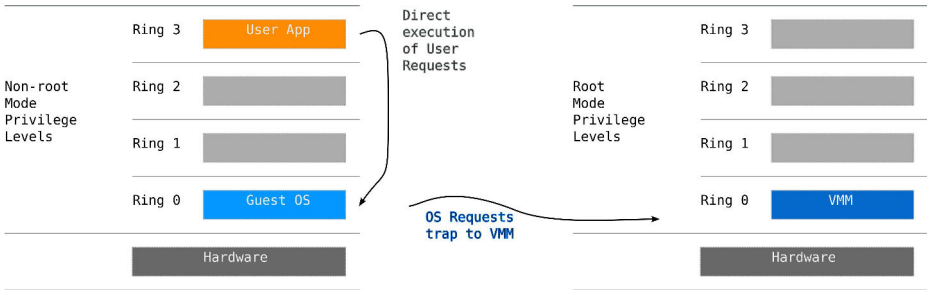


Figure 3.2: Intel ISA's operation modes and privilege levels.

root operation. This limitation of control extends also to non-root operation in ring 0, which, in normal processors, is the most privileged level. Thus the intention is for the VMM to work in VMX root operation, while the virtual machine itself, including the guest operating system and application, work in VMX non-root operation. Because VMX non-root operation includes all four IA-32 privilege levels (rings), guest software can run in the rings in which it was originally intended to run, that is, the guest operating system kernel can run in ring 0 and guest applications can run in ring 3.

A key aspect of the VT-x technology that allows faster virtual machine systems to be built is the elimination of the need to run all guest code in the user mode, essentially by providing a new mode of operation specifically for the VMM. For code regions that do not contain instructions that affect any critical shared resources, the hardware executes as efficiently as it would have on a normal machine. It is only in few cases where this is not possible that a certain degree of emulation must be performed by the VMM. Thus, once in the virtual machine, the exits back to the monitor are far less frequent in the hardware-assisted virtualization case than in the emulation case.

3.1.1.3 ARM Virtualization Extension

In the ARM architecture there are two CPU mode, kernel and user. With the support of hardware virtualization, ARM introduced a third CPU mode called Hyp mode (see Figure 3.3). This mode allows a guest operating system to run inside a virtual machine as it would run on a physical machine but, if it executes a sensitive instruction, the processor traps to the Hyp mode and a virtual machine monitor takes control over the guest OS execution in order to emulate the required operation.

In contrast, Intel has a root mode and non-root modes which are orthogonal to CPU protection modes, and can trap operation from non-root to root mode. As mentioned above, Intel's root

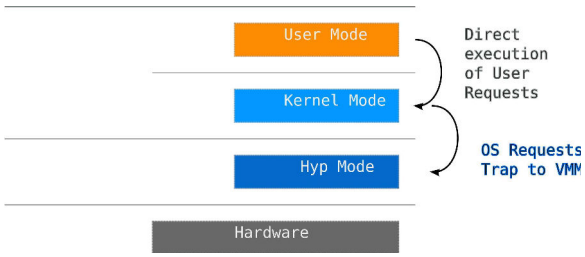


Figure 3.3: ARM ISA's operation modes and privilege levels.

mode supports the same four CPU privilege levels of user and kernel mode as in its non-root-mode, whereas ARM'Hyp mode is a strictly different CPU mode with its own set of features.

Another noticeable difference between ARM and Intel in terms of CPU virtualization is that Intel provides specific hardware support for virtual machine CPU data structure which is automatically saved and restored when switching to and from root-operation mode using a single instruction. This is used to automatically save and restore guest state when switching between guest and VMM execution. In contrast, ARM do not provide any such feature and saving or restoring the guest context need to be realized in software.

In the next section, we revisit the implementation of kvm, a virtual machine system that supports the hardware-assisted virtualization. kvm is integrated into the mainline Linux kernel and support Intel and ARM architectures.

3.2 Linux Kernel Virtual Machine

In our experiments we used the hosted virtual machine system Linux Kernel-based Virtual Machine (kvm) (Kivity et al., 2007). In kvm the host is the Linux operating system and the virtual machine monitor is composed of two components, the Kernel Virtual Machine (alias kvm) is the privileged component, and Qemu the unprivileged component. The software architecture of Linux, kvm and Qemu is illustrated in Figure 3.4.

kvm virtualizes the processor by creating a virtual machine data structure to hold the *virtual CPU* registers. It also virtualizes the memory by configuring the MMU hardware to translate the guest virtual addresses to host physical addresses if the architecture supports the *nested paging*. Otherwise it uses a *shadow page table* to emulate a hardware MMU. kvm traps the I/O instructions

and forwards them to Qemu which feeds them into a *device model* in order to emulate their behavior, and possibly triggers real I/O such as transmitting a network packet.

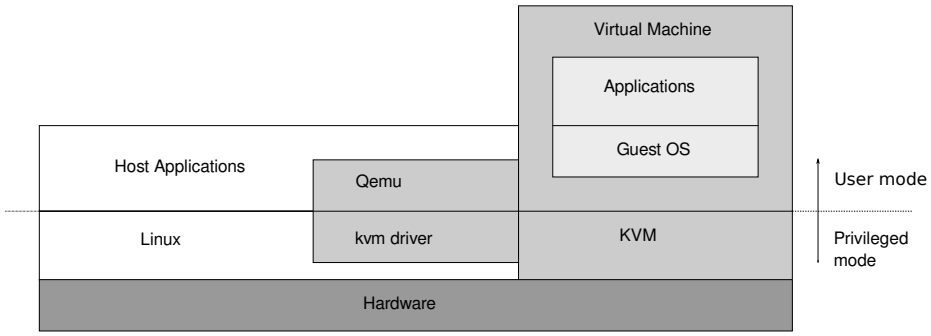


Figure 3.4: Linux Kernel Virtual Machine and Qemu.

3.2.1 Qemu

Qemu is a computer emulator software (Bellard, 2005) usually used to emulate a hardware architecture on another different architecture, for example emulating a Power-PC ISA using an IA-32 ISA.

When Qemu is executed with the -enable-kvm option, the CPU emulation mechanism of Qemu is disabled. The Qemu software invokes the services provided by kvm to execute the code of the guest operating system natively on the hardware. This operation is only possible when the guest OS is targeted for the same architecture of the host machine processor. For example, the guest OS is an x86 version of Linux and the host machine processor is an x86.

Qemu is used by kvm to emulate I/O devices. When a guest I/O operation, such as sending a packed on the network, or reading from disk is encountered, it traps to the kvm code which forwards it to Qemu. If the requested device is supported by the Linux host OS, the request is then converted into a Linux host OS system call. Now kvm, through Qemu, acts as a user-level application under Linux. When the application returns from the system call, the control gets back to kvm and then into the guest OS running on the virtual machine.

3.2.2 Virtual Machine Process

Starting a virtual machine under kvm could be done by starting a Qemu user process. When the Qemu process starts executing, it requests the creation of the virtual machine data structure. kvm

creates a virtual machine data structure and associates it to the Qemu process. Then, when the Qemu process is scheduled by the Linux kernel, it requests from the host to start executing the code of the guest operating system.

Figure 3.5: Scheduling of the "I/O thread" (qemu-system-x86-2080) and the "virtual CPU thread" (qemu-system-x86-2082) created by kvm and Qemu. The guest OS executes one task doing a set of arithmetic computation for $500ms$ periodically every $1000ms$.

Figure 3.6: Here we zoomed into one $500ms$ execution to show how the ksoftirq which is a Linux host thread handles the periodic timer interrupt every $1ms$. Actually, it is a virtual timer interrupt generated by the guest OS every $1ms$ to mark the scheduling tick.

After that, the processor starts executing the guest OS code until it encounters a sensitive instruction, an I/O operation, or until the occurrence of an interrupt. The Linux operating system schedules this virtual machine process as it schedules the other regular processes. Figure 3.5 shows how the Linux kernel installed on a quad-core Intel core-i7 hardware, schedules the kvm and Qemu processes. Figure 3.6 shows how the ksoftirq thread of the Linux kernel is scheduled to handle the virtual timer interrupt generated by a guest operating system running on a kvm virtual machine.

In the next section, we present an evaluation of the virtualization overhead on a real-time operating system using kvm virtual machine system.

3.3 Scheduling Latency Evaluation

One of the most used metric to evaluate the real-time capability of a real-time operating system is the *scheduling latency*. The scheduling latency is a delay incurred by a real-time task when it is released. In general, a real-time task is activated in response to external events (*e.g.*, when a sensor triggers) or by periodic timer expirations (*e.g.*, once every $10ms$). Following the activation of a task, the OS kernel go through the following sequences of steps:

1. the processor is interrupted and the control is transferred to an interrupt handler to acknowledge the timer or device interrupt,
2. the interrupt handler identifies the task to release and adds it to the ready queue,
3. then the scheduler is invoked to decide if the resumed task should be scheduled immediately and on which processor,
4. and if the resumed task has a higher priority than the currently running task then, a context-switch is performed after the task have been dispatched.

While in theory, a highest priority task is dispatched immediately after its release, in practice, the precedent steps are subject to delays. Step 1 is delayed if interrupts are disabled by critical section in the kernel. Step 2 is delayed due to cache misses, bus memory contention, and in multiprocessor lock contention. Step 3 is delayed if preemption are temporarily disabled by critical sections in the kernel. And step 4 is delayed due to a TLB flush on hardware without tagged TLB.

As a result, there is always a delay incurred even by the highest priority task. This delay, known as the *scheduling latency*, impacts the response time of all tasks and imposes a lower bound on deadlines that can be supported by the operating system. For this reason, it is mandatory to estimate the scheduling latency to decide whether the system is able to meet the temporal requirements.

The cyclictest benchmark (Molnar, 2004) was developed to measure the *scheduling latency* of the Linux kernel and its real-time variant PREEMPT_RT. It is used as the standard metric to evaluate the real-time performance of the mainline Linux before and after applying of the PREEMPT_RT patch to the kernel. Recall that PREEMPT_RT aims at improving the real-time latency by reducing

the number and the length of critical sections in the kernel that mask interrupts or disable preemption.

In our evaluation, we measured the real-time performance of Linux-PREEMPT_RT. We used Linux-PREEMPT_RT as a guest RTOS, *i.e.*, the RTOS that is installed on a virtual machine. We used Linux-kvm as a hosted virtual machine system. We installed Linux-kvm on a hardware platform consisted of a quad-core Intel Q6600 2.4GHz with 4GB RAM, enabled with hardware-assisted virtualization. We configured the host Linux kernel with the PREEMPT_RT patch to improve its responsiveness. Note that in this experiment, the host OS and the guest OS are both real-time Linux.

In order to provide the virtual machine with maximum resources from the host platform, we raised the SCHED_FIFO[1] real-time scheduling priority of the virtual machine process using the "chrt " Linux command. We pinned the virtual machine process to the first core of the machine using "taskset " Linux command to avoid any migration overhead from affecting the measurement, and we configured the kvm virtual machine with the option that locks the code of Qemu and the guest OS in cache memory and prevent any contention in the memory hierarchy from disturbing the measurement.

We configured cyclictest to create one thread, thus it uses one processor of our quad-core hardware. This thread executes a while( ) loop in which it records the current time, then calls the sleep( ) function to wait for a specified amount of time (in our experiment it is equals to $10ms$), after its wake up it records the current time again. Then it calculates the difference between its effective wake up time and its supposed wake up time, this difference represents the *scheduling latency*. The thread repeats these operations periodically every $10ms$ until the end of the experiment.

We executed cyclictest for more than 24 hours, which generate 10 millions samples per configuration. The result of each execution is a histogram of observed scheduling latencies, where the x-axis represents the measured delay and the y-axis the absolute frequency of the corresponding value plotted on a log scale. Sample were grouped in buckets of size $1\mu s$. Figure 3.7 shows the histogram of the experiment from the real hardware and Figure 3.8 shows the result from the virtual machine. Figure 3.9(a) and Figure 3.9(b) show the same experiment but using a more recent Intel

[1]The SCHED_FIFO is a POSIX's implementation of a real-time *first-in first-out* scheduling algorithm in Linux. The priorities range between 0 to 99, with 99 being the highest priority and 0 the lowest.

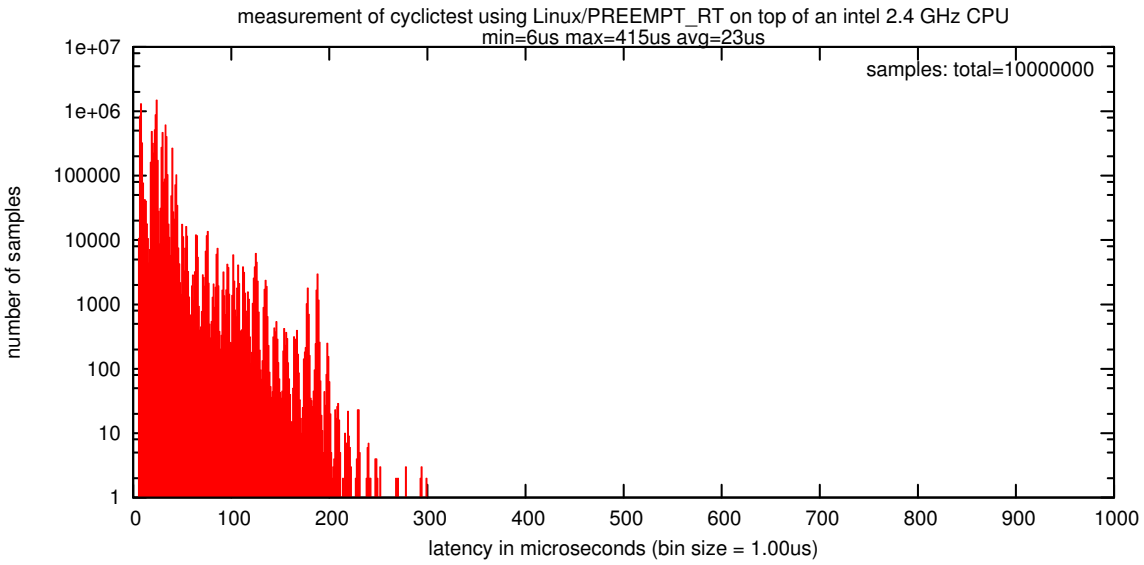


Figure 3.7: Scheduling latency of a real-time Linux running natively on an quad-core Intel 2.4GHz hardware. Here, cyclictest is executed on the real-time Linux installed on the real hardware.

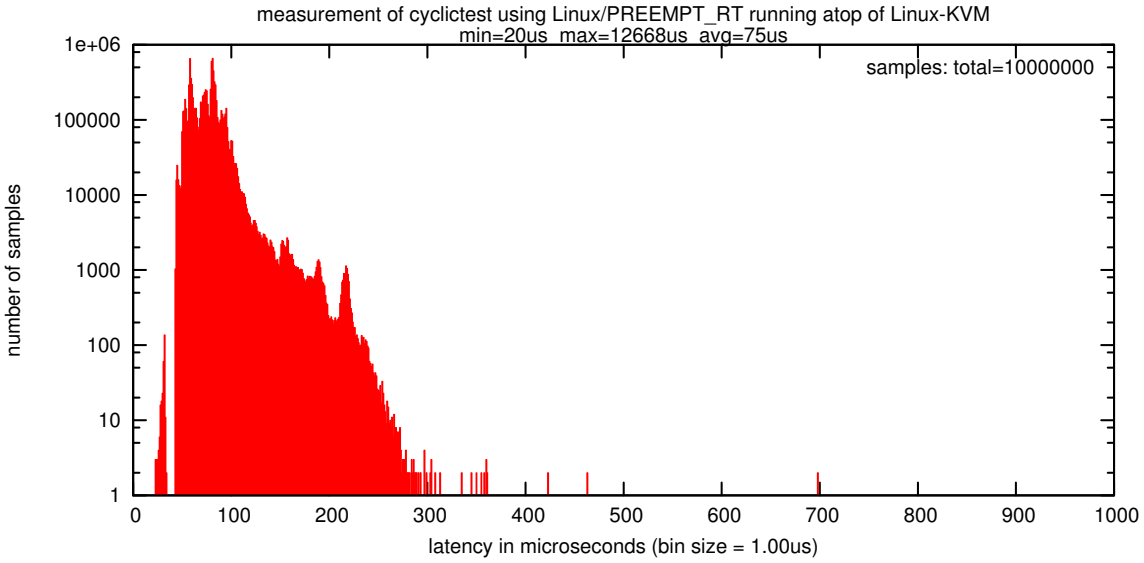


Figure 3.8: Scheduling latency of a real-time Linux running on a virtual machine. Here, cyclictest is tested on the real-time Linux executed on the virtual machine, the host OS is also a real-time Linux and the real hardware is the same as above.

core i7 2.6GHz 8GB RAM hardware platform and during a shorter period of time (three hours in virtual, and six hours in native case).

As can be seen in Figure 3.7 and Figure 3.8, the shape of the two histograms shows that both configurations exhibited comparable latencies with a slight shift in the virtual machine. Scheduling latencies under the virtualized RTOS are higher than the native RTOS, with an average of $75\mu s$, while it is centered around $23\mu s$ in the native case. This suggests that the additional latency is added by the virtualization overhead.

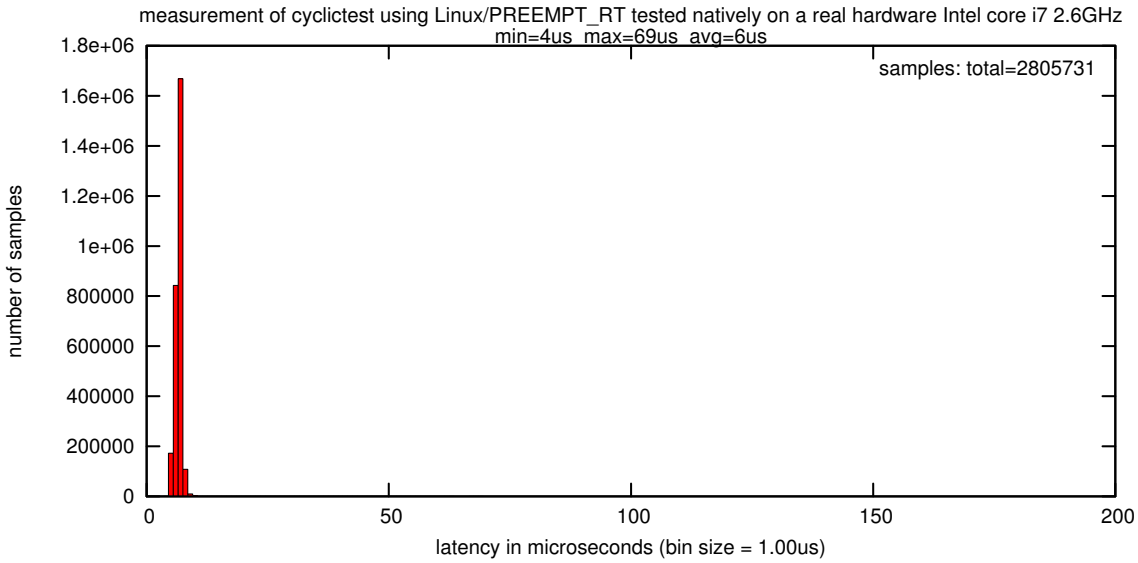
The histograms from Figure 3.9(a) and Figure 3.9(b) show clearly that the virtualization overhead added approximately $100\mu s$ to the scheduling latency in the average case.

However, comparing the maximum latencies, it can be seen that there is a considerable difference between the two maximum scheduling latencies. The maximum observed latency is around $415\mu s$ in the native case, while it reaches the $12668\mu s$ in the virtual case as indicated in Figure 3.7 and Figure 3.8.

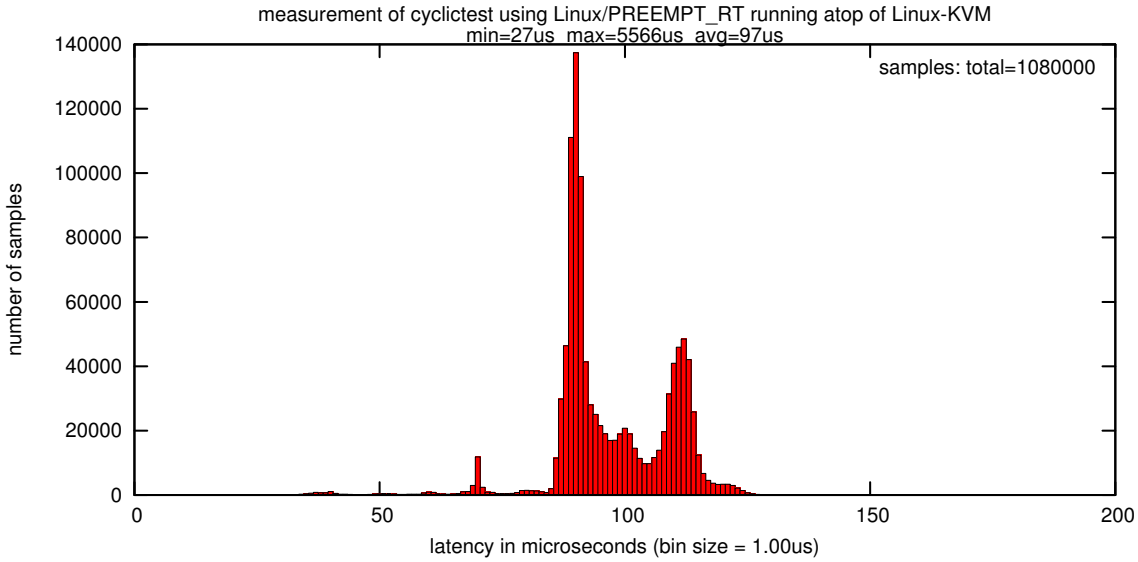
Given the distribution of the latencies, this worrying high latency seems to be an outlier, and we are tempted to say that it does not reflect the performance of the virtual machine system, but it is very difficult to confirm this conclusion. Especially, when it is difficult to reproduce it, that is, running the experiment for more longer time may lead to higher maximum as it may lead to lower maximum.

It is also very difficult to track down the source of such a very high latency. As we locked the code of kvm virtual machine, and the code of cyclictest in cache memory, we may eliminate the cache miss and page fault from the list of the sources that caused such a delay.

Kiszka (2011) conducted a similar experiment but by adding a disk I/O and a network workload on the host in order to measure the maximum latency in an overloaded situation. He observed a higher maximum latency than the one we obtained, and he suggested that this could be caused by the global mutex lock in Qemu. This lock protects access to the device emulation layer, which allow only one thread to request an emulated device. As solution, Kiszka (2011) proposed a complete re-engineering of a list of software modules in Qemu. However, this is not the case in our experiment because we executed cyclictest alone, without any background that could pollute the cache memory or generate any resources contention.



(a) On a real hardware



(b) On a virtual machine

Figure 3.9: Scheduling latency of a real-time Linux measured on a recent Intel core i7 2.6GHz hardware.

Zuo et al. (2010) suggested that such a delay may be caused by the handling of a non-maskable interrupt in the Intel x86 architecture, namely the SMI (system management interrupt). Such an interrupt forces the processor to interrupt the current running process, saves its context, and enters the system management mode to execute a code of separate operating environment such as a firmware. In their experiment, the authors found out that the USB legacy device was the main source of SMI in their platform. And disabling this device in the BIOS lowered the maximum latency to below $1ms$. We believe that this does not apply in our case because our hardware platform was not accessed during all the experiment.

It is clear that optimization of the software design and implementation, and judicious configuration of the hardware platform allow better performance, but in our case we are more concerned by investigating the common problems and overheads related to the hardware virtualization extension, such as CPU virtualization, Memory Management Unit and I/O virtualization, and their efficient use by the virtual machine monitors in general.

Given the degree of uncertainty regarding the cause of such a maximum latency, we preferred to decompose this scheduling latency into finer-grained overheads and latencies as defined in the different four steps described above in order to investigate the reason of the virtualization overhead.

3.4 Fine-Grained Overheads and Latencies Evaluation

In this section we present our evaluation of a virtualized RTOS. First, we define the overheads and latencies that are of interest. Second, we describe the hardware platform and the RTOS that we used in our experiments. Then, we present the synthetic workloads used to measure the overheads and latencies.

3.4.1 Overheads and Latencies

In the previous section we presented the scheduling latency which is the delay incurred by a real-time task when it is released. We described also how this delay could be divided into four steps. In the following list of overheads and latencies, we re-define each step using an overhead or latency internal to the operating system kernel, then we will measure each overhead and latency separately in order to observe where the bottleneck is.

- **Event Latency** (Δ^{event}) is the delay from the raising of the interrupt signal by the hardware device until the start of execution of the associated interrupt service routine (ISR).
- **Release Overhead** ($\Delta^{release}$) is the delay to execute the release ISR. The release ISR determines that a job J_i has been released and updates the process implementing a task T_i to reflect the parameters of the newly-released job.
- **Scheduling overhead** (Δ^{sched}) is the time taken to perform a process selection.
- **Context-switch overhead** (Δ^{cxs}) is the time required to perform a context switching.

Figure 3.10 illustrates how these overheads and latencies are ordered on a timeline.

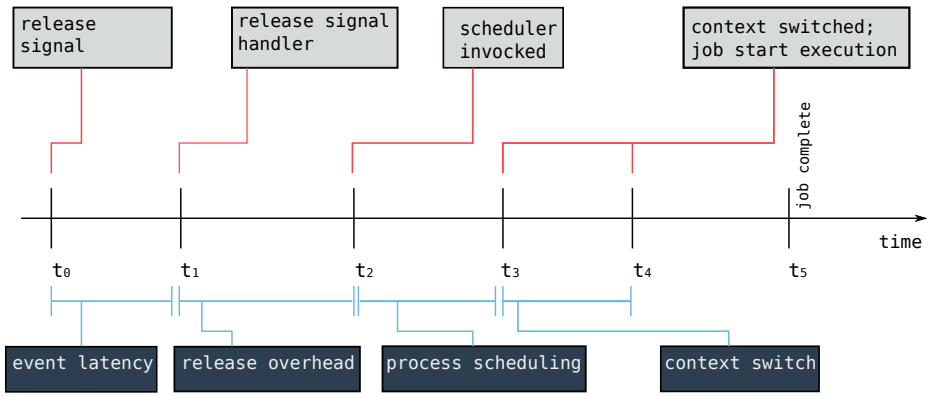


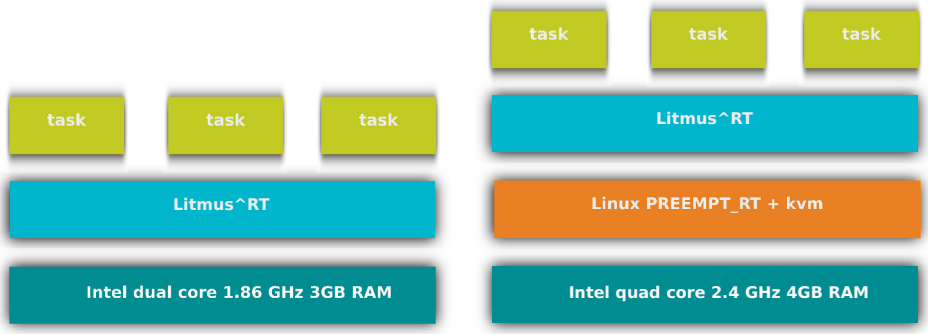
Figure 3.10: The delay from the raising of an interrupt by hardware until the associated task's job starts executing, detailed in terms of separated overheads and latencies internal to the RTOS kernel.

3.4.2 Hardware platform

Our evaluation compares two different configurations, the native and the virtual configurations. In the native configuration, we are more concerned about the native RTOS, and we used a dual-core Intel 1.86 GHz as a hardware platform. On this platform we installed a real-time Linux, LITMUS$^{\mathrm{RT}}$ (Brandenburg et al., 2007) as a native RTOS that we configured with the P-FP (*partitioned-fixed priority*) scheduler plug-in.

In the second configuration, we used the quad-core Intel 2.4GHz enabled with the VT-x feature to support hardware virtualization. As a host operating system, we used a real-time Linux by configuring a stock Linux kernel with the PREEMPT_RT patch. We used the *hosted VM* system kvm. We installed the LITMUS$^{\mathrm{RT}}$ real-time operating system on a virtual machine.

Note that in both configurations, only one core of the machine was used as a virtual CPU. We should note also that the difference in the CPU frequency between both host hardware did not influenced the conclusion drawn from the results, as we will see in the results section there is no diffrence in average-case, and the only notable difference is in the worst-case which is independante form the underlying hardware as we will explain later.



(a) Native Platform (b) Virtual Platform

Figure 3.11: Architecture of the native and the virtual platform used in the experiments.

3.4.3 LITMUS^{RT} and Feather Trace toolkit

LITMUS^{RT} (see Section 2.4.3) is a native real-time Linux version. It extends the Linux kernel with multiprocessor real-time scheduling policies and locking protocols. With regard to our purpose LITMUS^{RT} kernel source code is instrumented in the way that permits the measurement of each overhead and latency separately. In LITMUS^{RT}, the Feather-Trace (Brandenburg and Anderson, 2007) infrastructure is used to trace the duration of each step.

Feather-Trace is a light-weight event tracing toolkit based on a static instrumentation of the kernel. Its main characteristic is the low level overhead that it introduces, which is an important feature in our case because it ensures that the measurement's traces do not disturb the results.

Feather-Trace relies on two components: static triggers, and a wait free multi-writer, single-reader FIFO buffer. Feather-Trace works by directly rewriting the kernel's code.

When a trigger is activated, a parameter of an x86 jump instruction (JMP) is rewritten to call a user-provided function instead. To enable an event, the offset parameter of the jump instruction is set to zero, which effectively disables the jump (see Figure 3.12). As a result, the code that directly follows the jump instruction pushes the required context information on the stack and transfers

control to a call-back function. No operating system support is necessary and no locking or mutual exclusion support is required.

If a tracing event is disabled, then only one additional instruction is executed compared to the case where the kernel code is not instrumented. On the other hand, if a tracing event is enabled then, only one additional instruction is executed compared to a normal function call.

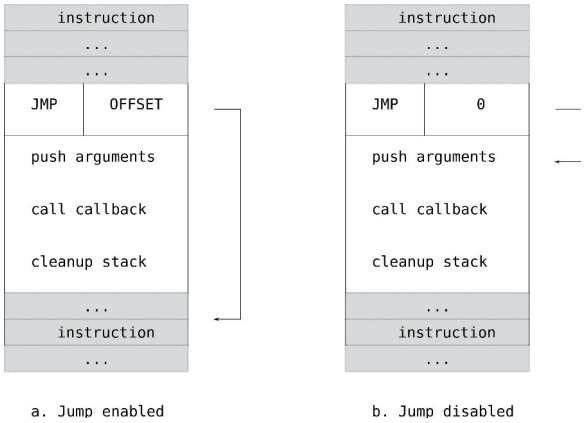


Figure 3.12: (a) If the jump is enabled the code does not trigger the tracing, (b) otherwise the jump is disabled and the tracing code is executed.

Testing whether a given event is enabled with only a single instruction that does not access memory and which has no effects on either branch prediction or pipelining in both the enabled and the disabled case is arguably optimal.

The second component of Feather-Trace is the wait free multi-writer, single-reader FIFO buffer. The buffer is implemented without any need for locks, each read and write operation completes in a bounded number of steps, and the support for arbitrarily many writers makes the collect of performance data an operation with very little overhead. Moreover, to improve performance, a single reader is used to flush the content of the buffer from memory to stable storage. The reader could execute safely in parallel with multiple writers.

In LITMUS$^{\mathrm{RT}}$, Feather-Trace is used to record timestamp at various points during the execution of the scheduler. For example, there is a trigger just before a context-switch, and one just after it. When a trigger calls a function, it records the current time based on the number of cycles provided in the TSC register. These timestamps are then exported to user space by means of the Feather-Trace character device. Based on the recorded pairs of timestamps before and after an event, it is later

possible to reconstruct how long each event took. Using these overhead samples, it is then possible to compute overhead statistics such as average and maximum observed overheads.

3.4.4 Synthetic Workloads

The experimental methodology we used in our evaluation is inspired by the methodology used to evaluate the LITMUS$^{\mathrm{RT}}$ kernel (Brandenburg, 2011). To measure the overheads and latencies we used a synthetic task set system. Each task set has a size $n = m * k$, where m is the number of processors, and k is the number of tasks per processor and ranges from 1 to 20. For each value of n, 5 task sets systems were generated and each task set within a system was executed for 60 seconds.

The task sets were generated by randomly choosing their CPU utilization of each included task until the CPU utilization capacity was reached. The utilization of each task was randomly generated using one of the following distributions: light uniform, light bimodal, light exponential, medium uniform, and medium bimodal, as proposed by Baker (2005). The task periods were generated using a uniform distribution within a [$10ms$, $100ms$] range. Then, the utilization and the period values were used to compute the execution time of each task.

These distributions are well known to stress specific sources of algorithmic and overhead-related capacity loss. For example, using light utilization distributions produces task sets with many tasks where each task has a low CPU utilization which results in a large number of interrupt sources and long ready queues. Using medium utilization distribution produces task set with a mix of low and high CPU utilization tasks.

In addition to real-time workload, m background tasks were launched that create memory and cache contention by repeatedly accessing large arrays. This avoids the underestimation of the worst-case overheads.

The measurements of overheads and latencies results in a large log events records. From this large log events, we extract the measurement for each overhead and latency. Then, for each overhead and latency the average-case and the worst-case statistics are distilled.

3.5 Results

In total, the overhead experiments resulted in 1GB of events records, which contained more than 500 thousands valid overhead samples. Figure 3.13(b), Figure 3.14(b), Figure 3.15(b), and Figure 3.16(b) show the average-case (green curve) and the worst-case (red curve) trends of all the overheads and latencies from the virtualized RTOS. Figure 3.13(a), Figure 3.14(a), Figure 3.15(a), and Figure 3.16(a) show the similar measurements from the native RTOS. The values of overheads and latencies in the graphs are given in microsecond and plotted as a function of the number of tasks per processor.

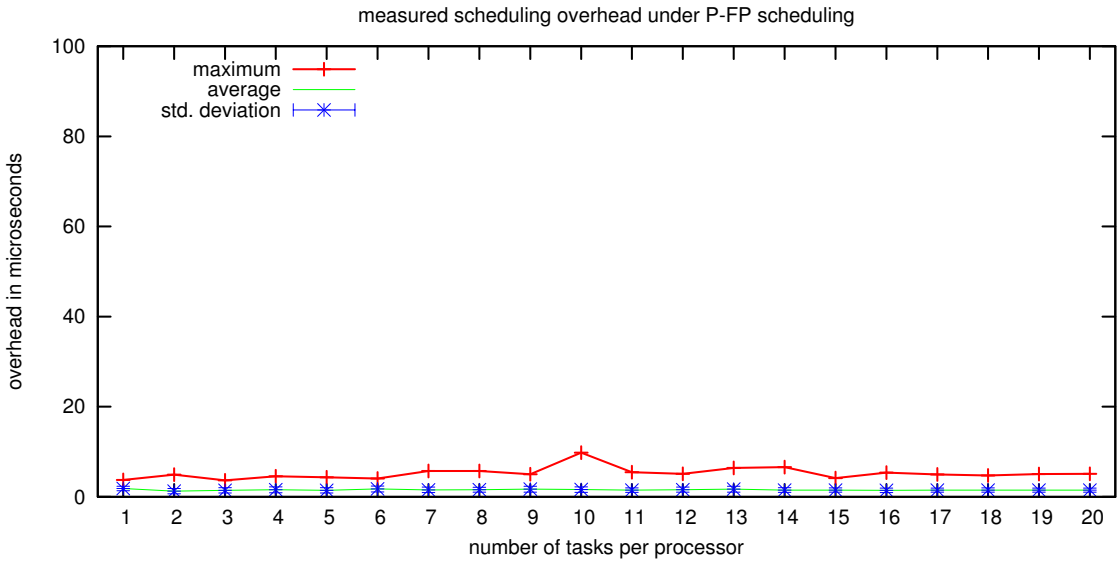
By comparing the results between the native case and the virtual case we can observe that in the average-case the delays are in the same order-of-magnitude. This similarity is explained by the fact that in most cases the guest code is executed natively on the machine, therefore it runs at the same speed as the native code.

However, by analyzing Figure 3.13(b), Figure 3.14(b), Figure 3.15(b), and Figure 3.16(b), we can see that the worst-case values of the virtualized RTOS are very far from the average-case overheads. While most worst-case overheads and latencies could be attributed to virtualization overhead, for example when the VMM intervene to handle a page fault generated by the guest, or to emulate an I/O operation, other very high worst-case values could not be intuitively explained.

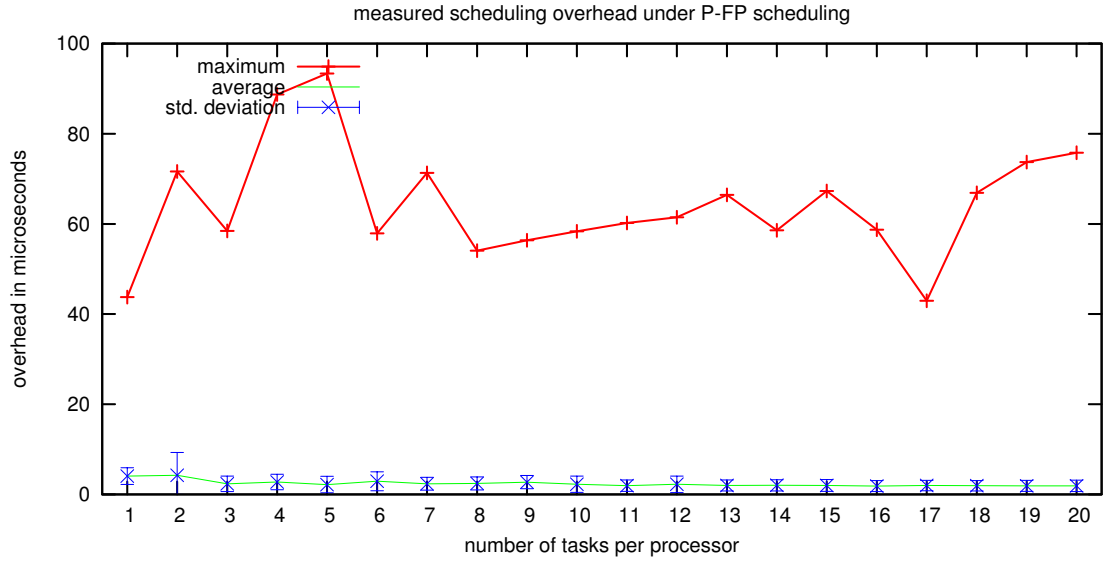
In an attempt to further understand the cause of such high overhead, we plotted the distribution of the occurrence of the worst-case values that are far from the average case. Figure 3.17 shows the histogram of the event latency for $n = 14$ tasks per processor in the virtual case. Figure 3.18 and Figure 3.19 show the histogram of the Δ^{cxs} at $n = 10$ and $n = 20$ tasks respectively, and Figure 3.20 shows the distribution of the scheduling overhead at $n = 5$ tasks per processor.

Given the average case values and the low probabilities of the very high maximum values observed, it is difficult to say that the worst-case observed overheads and latencies are caused by the virtualization overhead. Nevertheless, it is still difficult to say definitely what causes these worst-case measured values.

The variability in the worst-case values is more important in the virtual case than in the native case. We conjecture that this could be attributed to the fact that the virtual machine process is subject to scheduling by the host operating system.

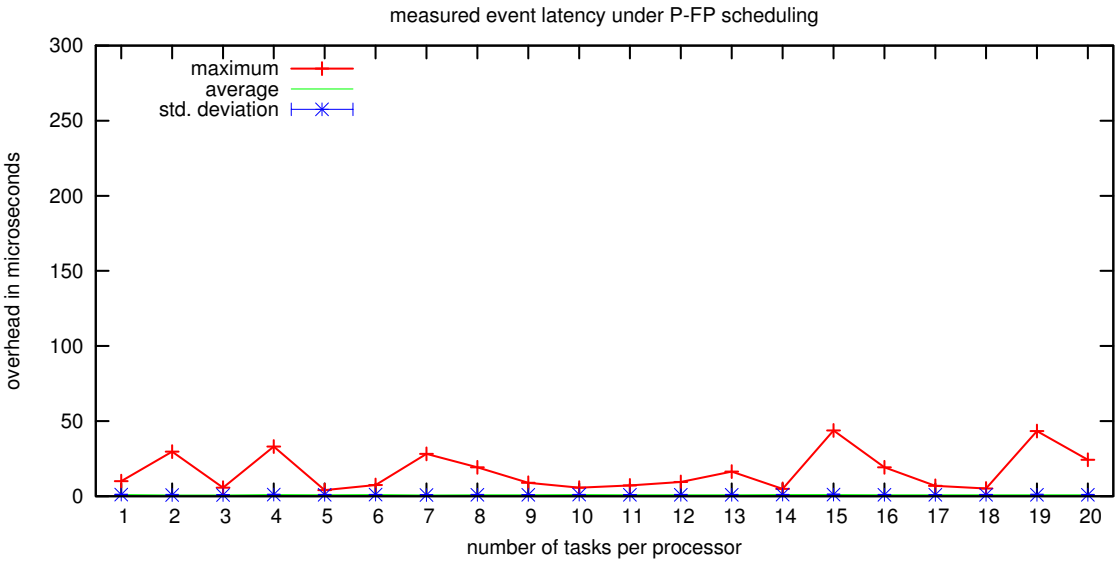


(a) Scheduling overhead in the native case

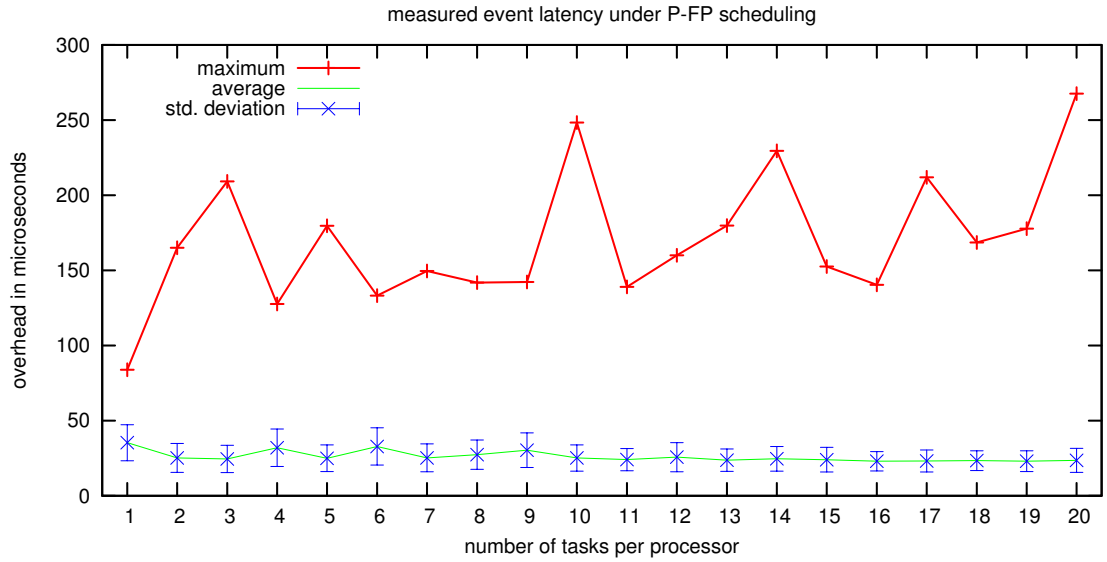


(b) Scheduling overhead in the virtual case

Figure 3.13: In the average case, the scheduling overhead of the virtualized RTOS is roughly comparable to scheduling overhead from native RTOS. A key observation from Figure 3.13(b) and Figure 3.13(a) is that, in the average-case the scheduling overhead (Δ^{sched}) under either configuration (native and virtual) does not appear to be correlated to the task set size. This is because in LITMUS$^{\mathrm{RT}}$ the *partitioned fixed-priority* scheduler is efficiently implemented using a bit-field-based ready queues to enable fast lookup of ready processes. As a result, the runtime complexity of finding the next highest-priority job does not depend on the number of ready tasks. Another contributing factor is that task sets with high task counts also have a high utilization, which means that the background processes that creates memory contention executes less frequently and results in an increased cache hit rate.

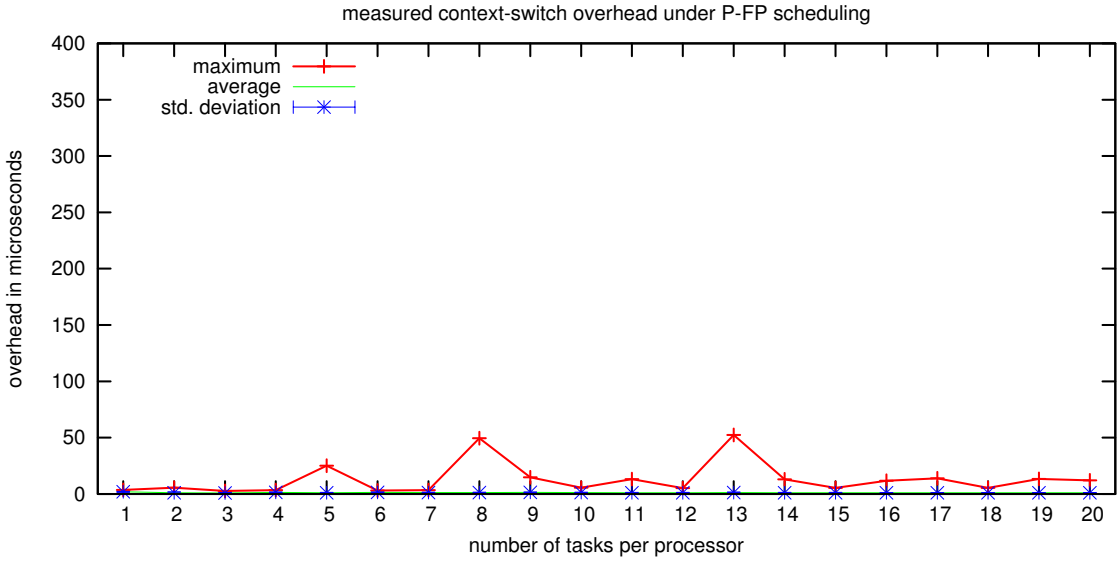


(a) Event latency in the native case

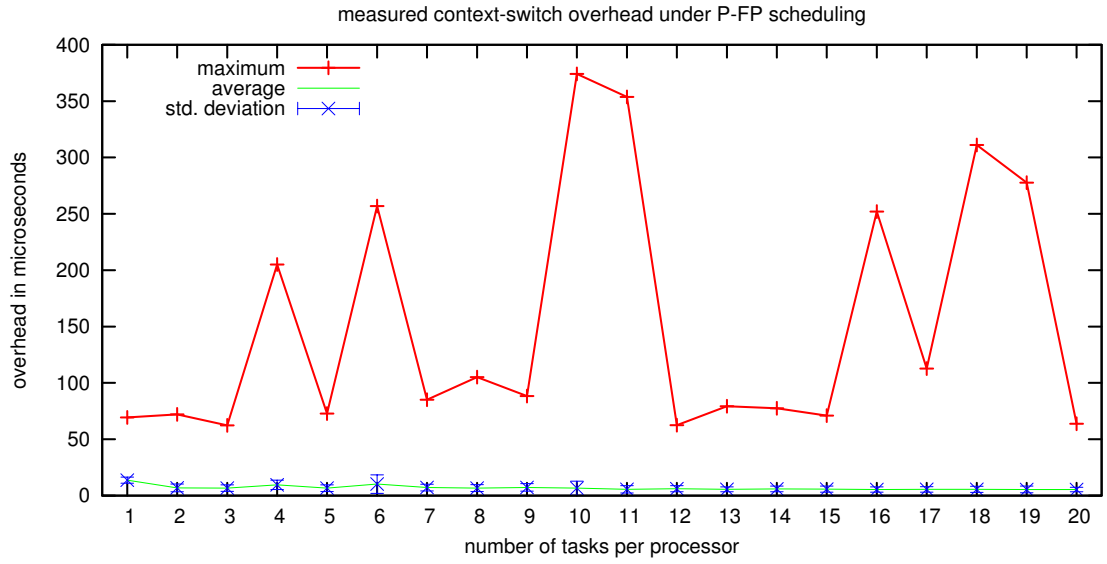


(b) Event latency in the virtual case

Figure 3.14: In the virtualized RTOS we observed an increase of the event latency in the average-case in comparison to the native RTOS. Recall that the event latency is the delay from the raising of the interrupt signal by a hardware device until the start execution of the associated ISR. This difference is due to the fact that the event latency is related to the virtualization of a device interrupt (in this case a timer) as we will explain in details in the next section.

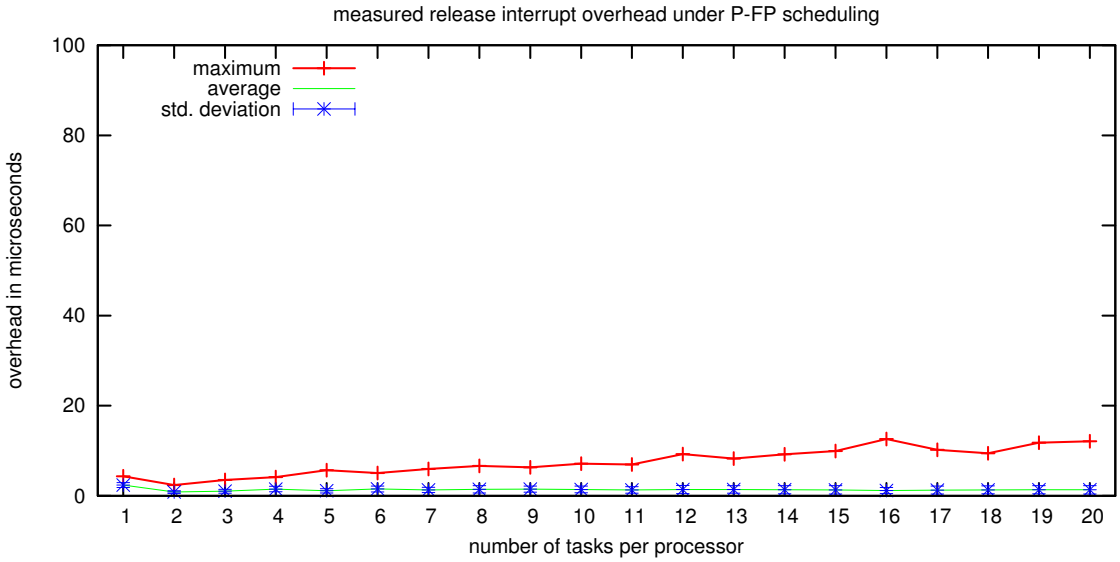


(a) Context-Switch overhead in the native case

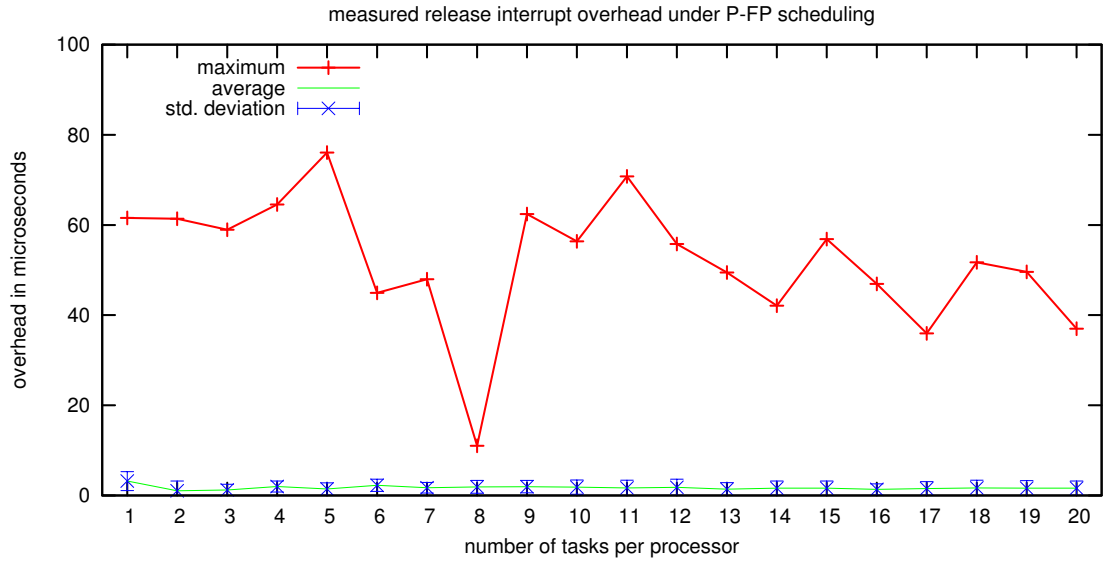


(b) Context-Switch overhead in the virtual case

Figure 3.15: Measurement of the context-switch overhead (Δ^{cxs}). As in the event latency case (Δ^{event}), we can see that even in the native case (Figure 3.15(a) at $n = 8$ and $n = 13$) the worst-case context-switch overhead appears to be different from the overall trend. In our experiment, this variation in the worst-case overhead trend occurred frequently in the measurements of event latency and context-switch overhead because they are strongly affected by interrupt delivery. In contrast, this variation occurred rarely in the measurements of scheduling overhead since interrupt delivery is disabled throughout most parts of the measured scheduling code path.



(a) Release overhead in the native case



(b) Release overhead in the virtual case

Figure 3.16: Release overhead (Δ^{rel}) is the delay to execute the job release handler. This function is executed while the interrupts are disabled, which explain the little variation in the worst-case values throughout the experiment. However, in the virtual case (Figure 3.16(b)), we can see that the worst-case is very high in comparison to the average-case. We explain this by the fact that even if the release handler is executed while interrupts are disabled in the guest operating system, it does not mean that the guest operating system could not be preempted by the virtual machine monitor. The guest OS is not allowed to disable the interrupt in the system, and therefore it is subject to perturbation from the host workload. This preemption of the guest operating system could delay the response time of the kernel critical functions.

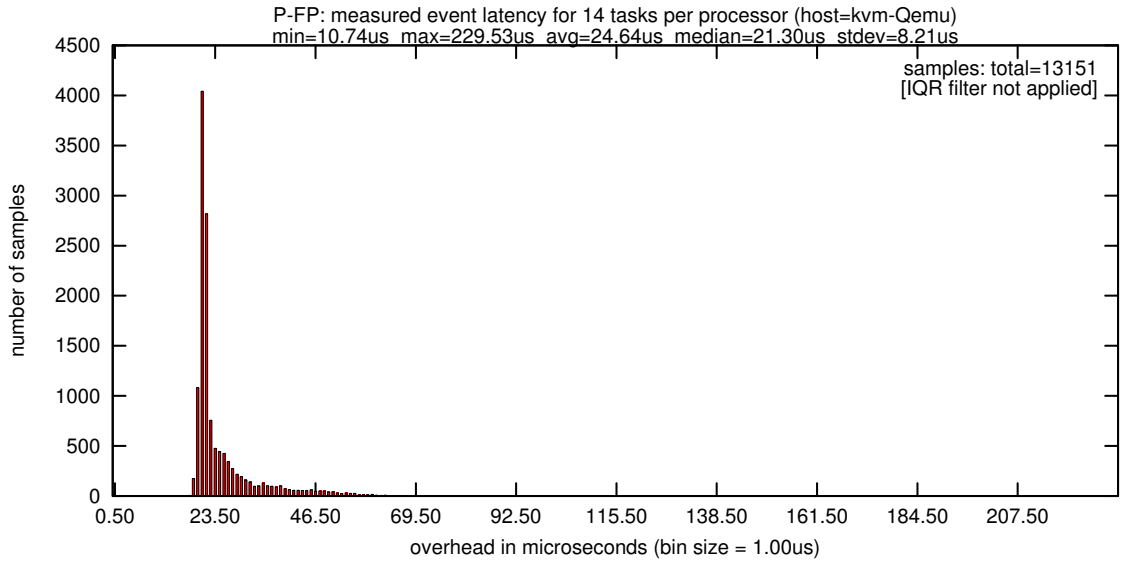


Figure 3.17: The maximum observed event latency occurred at $n = 14$ (see Figure 3.14(b)) and is equal to 229.53 μs. By looking at the histogram of samples this values is not visible due to its very low occurrence.

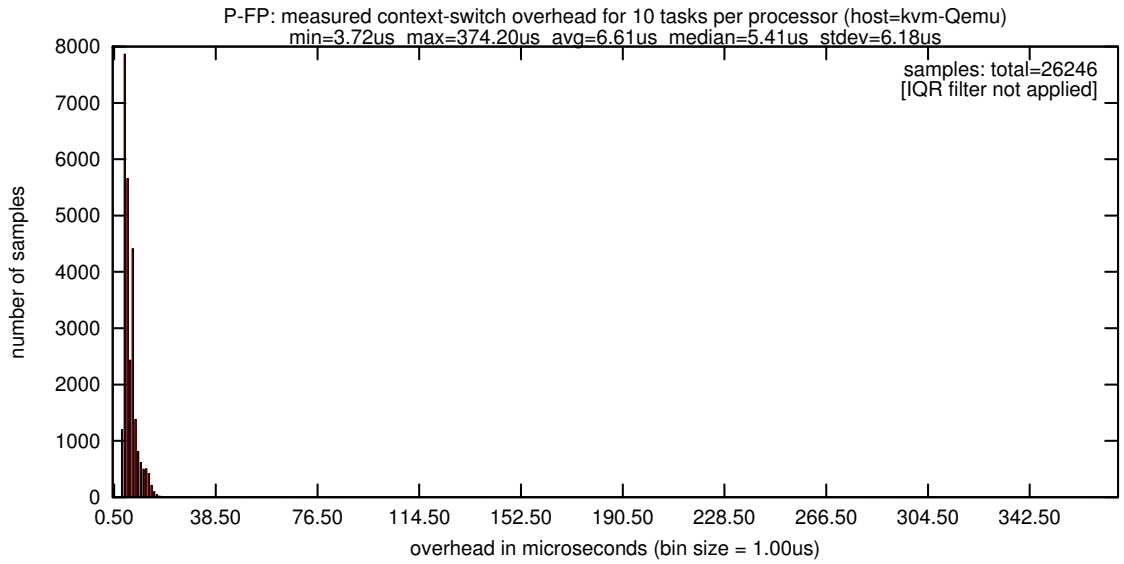


Figure 3.18: Distribution of the context-switch overhead for n equals 10 tasks per processor.

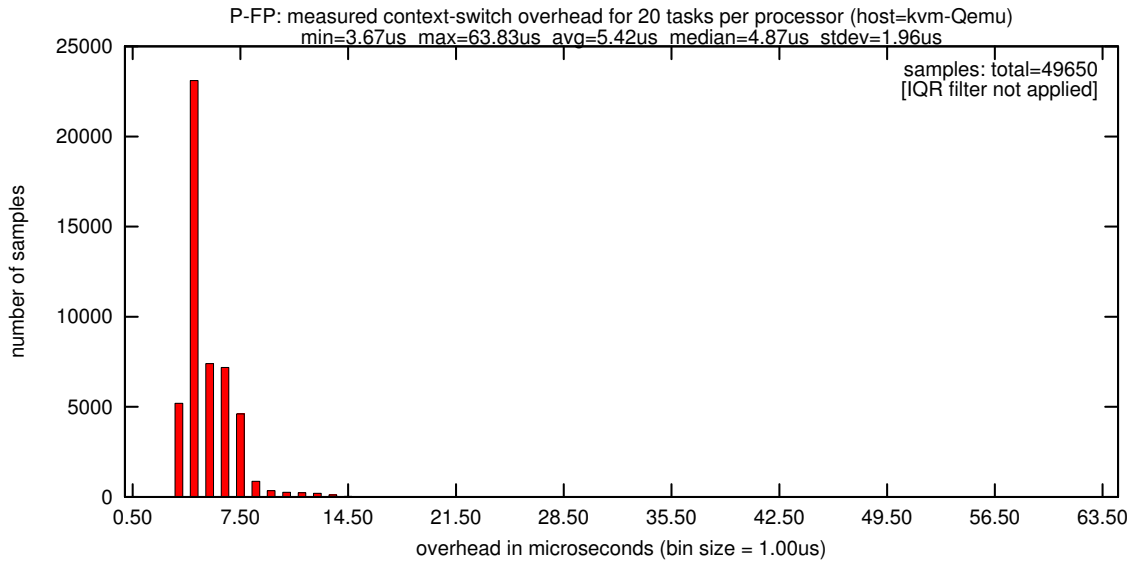


Figure 3.19: Distribution of the context-switch overhead for n equals 20 tasks per processor. Here, we can clearly see that most of values are centered around $5.42\mu s$, which is the average case context-switch overhead at $n = 20$ tasks.

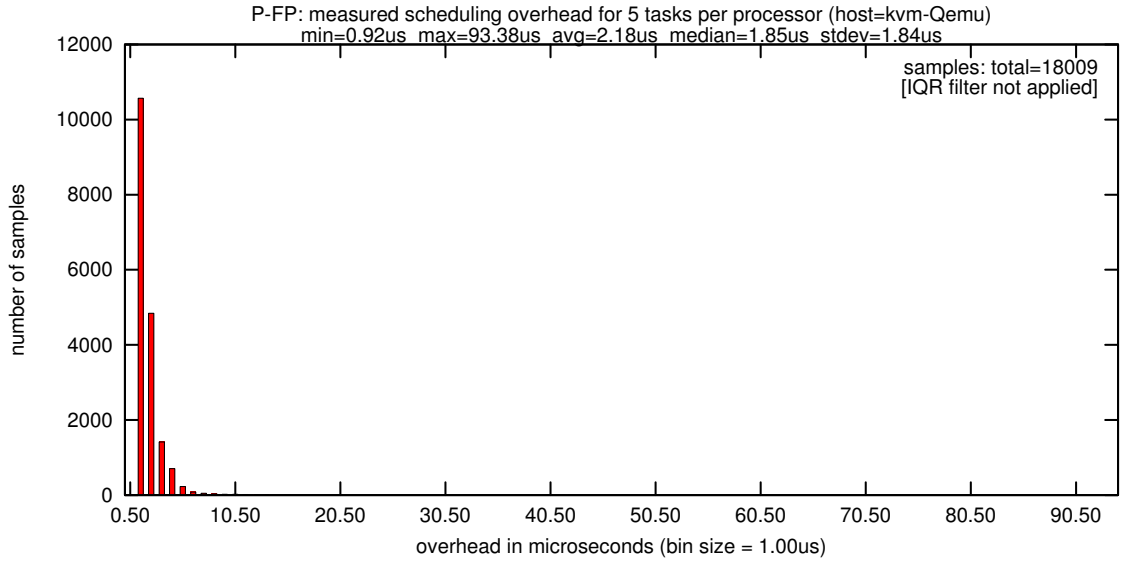


Figure 3.20: Distribution of scheduling overhead at n equals 5 tasks per processor, in the virtual case.

For instance, the handling of an interrupt by the host OS could preempt the virtual machine process and delay its execution. When the host OS finishes servicing the interrupt a scheduling is required to re-schedule the virtual machine process. This duration is added to the waiting delay. After that a context-switch needs to be performed by the host OS.

If the elected process is the virtual machine process, the cost of the context-switch is higher than the simple context-switch between two user processes or between an ISR and another kernel routine, because restoring the context of a virtual machine involves much more registers to load. The duration of the context-switch is also added to the waiting delay of the virtual machine process.

This delay could also be increased if there is a cache-miss or a page fault generated by the execution of the code while the virtual machine was waiting.

Note that this delay could be worse if the interrupt releases a task in the host OS that has a higher priority than the virtual machine. In the next chapter, we will investigate this case in more details and propose a method to alleviate it. We will show how this delay could not be upper-bounded without a proper scheduling of the virtual machine process by the host OS scheduler.

However, one of the performance degradation that we measured in our experiment and that we are able to attribute to virtualization overhead is the event latency (Δ^{event}). A pairwise comparison between Figure 3.14(b) and Figure 3.14(a) illustrates how the virtualization increased the (Δ^{event}) in average-case. In the next section, we explain the reasons of this increase, and discuss how assistance from the hardware architecture allows to build an efficient solution that reduces the Δ^{event} delay.

3.6 Emulation of the I/O interrupts

Among the measurements of the fine-grained overheads and latencies previously presented, the event latency (Δ^{event}) is the feature in the virtual RTOS that is largely impacted by the virtual machine system in the average case. Here we explain the reason of this degradation.

When an interrupt is raised by a physical device, it is intercepted by the virtual machine monitor, converted into a virtual interrupt, and injected into the virtual machine as a pending virtual interrupt. On the next activation of the guest virtual machine, the virtual interrupt is delivered to it, which transfers it to its appropriate ISR in the guest OS. As the event latency is the delay from the raising of the interrupt signal by the hardware device until the start of execution of the associated interrupt

service routine (ISR), we can see from Figure 3.21 how this delay is increased due the emulation of the I/O interrupt by the virtual machine monitor and the multiple VM exits and entries that induces.

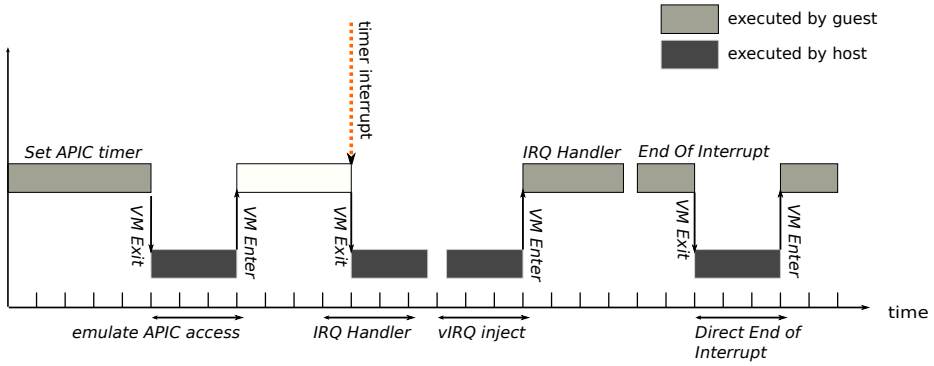


Figure 3.21: Scheduling of virtual machines according to the fixed-priority algorithm.

To avoid this overhead, hardware manufacturers added a new feature to their processors to enable the virtualization of interrupts. For example, the Intel VT-d (Intel Virtualization Technology for Directed I/O) (Intel, 2012) enables the virtualization of the Advanced Programmable Interrupt Controller (APIC). When this feature is used, the processor will emulate many accesses to the APIC, tracks the state of a virtual APIC, and delivers virtual interrupts, all in VMX non root operation without any exit from the virtual machine to the virtual machine monitor. Currently, a patch (Sekiyama, 2012) is under development to support this feature in kvm.

The primary evaluation of this feature using the cyclictest benchmark revealed that the scheduling latency was lowered after the application of the patch.

3.6.1 Comparison with ARM I/O virtualization

In order to improve performance, the ARM architecture allows many traps to be configured so they trap directly into a VM's kernel mode instead of going through the virtual machine monitor. Recall that in the ARM architecture there are three processor modes. The kernel mode and the user mode are used by the guest OS running in the virtual machine, and the Hyp Mode is reserved to the VMM (see Figure 3.3).

For example, traps caused by normal system calls or undefined exceptions from user mode in the guest OS can be configured to trap to a VM's kernel mode so that they are handled by the guest

OS without the intervention of the VMM. This avoids going to the Hyp mode on each system call or undefined exception, reducing virtualization overhead.

ARM architecture provides the Generic Interrupt Controller (GIC) and the Virtual GIC (VGIC).

Interrupts could be configured to trap to Hyp mode or kernel mode. This allows to re-direct the interrupts into guest OS without the intervention of the VMM. However, this is not a good idea because it prevents the VMM from taking control over the hardware. In the other hand, letting the VMM receiving all the interrupts results in higher latency as we have previously showed on the Intel architecture, where the acknowledge and the end of interrupt operations must go through the VMM.

To resolve this problem, ARM has provided the hardware support for virtual interrupts to reduce the number of traps to Hyp mode. Hardware interrupts trap to Hyp mode to retain VMM control, but virtual interrupts trap to kernel mode so that guest OSes can acknowledge, mask, and signal their completion without trapping into VMM.

Each CPU has a virtual CPU interface that the guest OS can interact with through memory-mapped I/O without trapping to Hyp mode, and a virtual CPU control interface, which can be programmed to raise virtual interrupts using a set of *list registers*, which are only accessed by the VMM. When a virtual device triggers an interrupt, it is treated as a virtual interrupt and the GIC traps to the VM directly in kernel mode.

In addition to the the counter that measures the time, and a timer for each CPU, ARM also provides a virtual counter and a virtual timer which allows a VM to access, program, and cancel virtual timers without causing traps to Hyp mode. In this way the VMM can be configured to use physical timers while VMs are configured to use virtual timers. Such a functionality is not present in the x86 hardware virtualization, forcing the VMM to emulate the virtual timer in software.

Unfortunately, due to architecture limitation virtual timer can not raise virtual interrupt, instead hardware interrupt is raised which trap to the VMM. This will force the VMM to create a corresponding virtual interrupt, inject it to the VM, and performing the hardware acknowledgement and completion.

We believe that the assistance from the ARM hardware architecture to virtualize the interrupts would lower the event latency (Δ^{event}) observed in our evaluation and facilitate the implementation of the virtual machine monitor because the emulation of interrupt is no longer required.

3.6.2 Comparison with Custom ARM Hardware Architecture

Given that the virtualization extension provided by ARM is only available on the ARM v8 instruction set architecture, Garcia et al. (2013) modified the ARM v5TE architecture and micro-architecture to improve the management of the interrupts. More specifically, an Atmel AT91SAM9XE, which implements the ARM v5TE ISA, was cloned and implemented on a Virtex 5 FPGA (*field programmable gate array*) in order to test some hardware design decisions such as adding a new processor mode reserved to the execution of a virtual machine monitor, called a Hypervisor mode, or an efficient management of virtual interrupts.

The Atmel AT91SAM9XE processor provides an Advanced Interrupt Controller (AIC) and a Programmable Interrupt Timer (PIT). The PIT was extended with an additional Timer counter, accessible in Hypervisor CPU mode, and used to drive the VMM's scheduling tick. Through this extension the VMM's timer interrupt bypasses the AIC and feeds the processor causing a transition to the Hypervisor CPU mode. The virtual Timer counter is used whenever a virtual machine processor context is restored by the VMM to update the standard timer counter which simplify the guest time-keeping.

The results of the evaluation of the hardware implementation on a Xilinx ML505 board with caches disabled showed that the number of instructions required to adjust the PIT counter when a guest context is restored dropped from 15 to 2. The time required to deliver an urgent interrupt to a real-time guest dropped from 6692 clock cycles to 12, because the VMM do not execute its scheduling function to select the guest to which the interrupt should be delivered.

3.7 Summary

In this chapter we evaluated the impact of the virtualization overhead on a hosted RTOS. We first evaluated the *scheduling latency* of a real-time operating system that was running on a bare hardware and on a virtual machine. We showed that in the average case this metric was approximately similar in both configurations, however, we observed a very high maximum in the virtual case comparing to the native case.

Then we divided this *scheduling latency* into a set of fine-grained overheads and latencies, and we measured separately each of these delays. The results of this evaluation confirmed the results of the first evaluation, and showed that the average-case overheads and latencies of a virtualized RTOS are similar to a native RTOS, except for the event latency where we observed a slight increase in the virtual case.

This second evaluation showed that the worst-case overheads and latencies were very far from the average-case. The analysis of the probabilities of these worst-case values led us to conjecture that these events are caused by two combined factors: interference from the interrupts that occurred in the host OS and virtualization overhead, such as switching between two worlds (the virtual machine and the virtual machine monitor), emulation of code, page-fault, cache miss, *etc.*

Given the average-case performance and the lower probability of the very high overheads and latencies, we can conclude that a soft real-time application should present the same performance when it is running on a virtual machine as it is running on a native RTOS.

In the next chapter, we show how the scheduling of the virtual machines affects the performance, and we argue that a non appropriate scheduling could even be more ”harmful” to the real-time performance than the virtualization overhead. We will provide an overhead-aware schedulability analysis that allows to guarantee the timing requirements for the hosted real-time applications, and we present an empirical evaluation of the method.

CHAPTER 4

Real-Time Scheduling of Virtual Machines

In this chapter, we show how the scheduling of the virtual machines is responsible for guaranteeing the timing requirements of the guest OS and its application. First, we define a set of basic concepts as a background for real-time scheduling theory. Then we present a schedulability analysis in the context of a virtual machine system. After that we extend this schedulability analysis by integrating the overheads measured in the previous chapter. Finally, we present an empirical evaluation of the proposed method.

4.1 Real-Time Task Model

In the following subsections, we summarize the main theoretical background necessary to understand the schedulability analysis in the context of a virtual machine system. We refer the interested reader to (Brandenburg, 2011), who survey the major real-time theoretical frameworks.

Under the *periodic task model* (Liu and Layland, 1973), a real-time workload consists of a set of n sequential tasks $\tau = \{T_1, ..., T_n\}$. Each task T_i is repeatedly released after the arrival of an asynchronous event, such as a device interrupts. When it is released, the task T_i executes a *job* to process the triggering event.

Each task T_i is characterized by three parameters (e_i, p_i, d_i). Its maximum execution requirement e_i, its *period*, p_i (with $p_i > e_i$), and its *relative deadline* d_i, (with $d_i > e_i$). A task T_i releases a job at least every p_i time units, executes for at most e_i time units, and each job execution should not exceed a d_i time units after its release.

The period p_i of a task T_i determines the successive arrival of jobs, such that $a_{i+j} \geq a_j + p_i$. A job $J_{i,j}$ executes at most e_i time units, then *completes* or *finishes* after $f_{i,j}$ time units. A job $J_{i,j}$ is said to be *pending* from its release to its completion.

The *response time* $r_{i,j}$ measures how long $J_{i,j}$ remains pending, and defined by $r_{i,j} = f_{i,j} - a_{i,j}$. Figure 4.1 illustrates all the task's temporal parameters that we defined above.

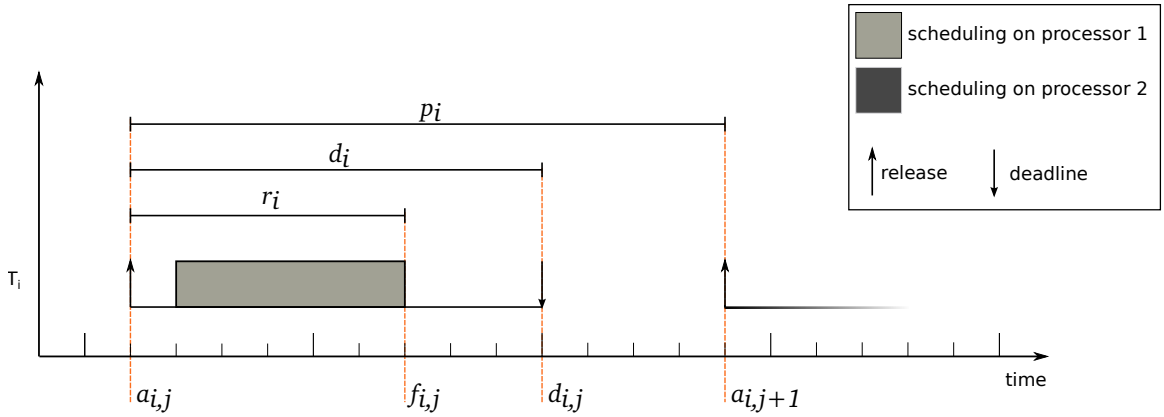


Figure 4.1: Illustration of the temporal properties of a periodic task.

The *utilization* of a task T_i, $u_i = \frac{e_i}{p_i}$, is the fraction of one processor that T_i requires. If T_i is not allocated this required amount of processing time, its jobs may miss their deadlines. And the *total utilization* of a task set τ is given by:

$$u_{\text{sum}}(\tau) = \sum_{T_i \in \tau} u_i.$$

4.1.1 Temporal Correctness

The ability of a real-time system to satisfy a temporal specification determines its temporal correctness. In a periodic task model, timeliness requirements are expressed as deadline constraints. In a *hard real-time* (HRT), each job must complete by its deadline, and in the *soft real-time* (SRT), a limited number of missed deadlines is tolerated by the system.

In the HRT case, the maximum response time r_i of a task T_i must be bound by its relative deadline d_i, which means that all jobs must complete by their absolute deadlines. The response time of a job depends on the scheduling algorithm, its processor requirement, and on the concrete release times and the execution of higher-priority jobs that prevent it from being scheduled. The *HRT schedulability* of a task set is defined by:

Definition 4.1. Given a scheduling algorithm $\mathcal{A}$, a task set τ is *HRT schedulable* if and only if, the condition $r_i \leq d_i$ is satisfied by all the released jobs, for each task $T_i \in \tau$.

And for the systems that can tolerate some deadline misses, the *SRT schedulability* of task set is defined by:

Definition 4.2. Given a scheduling algorithm $\mathcal{A}$, a task set τ is *SRT schedulable* if and only if, there exists a constant B such that $r_i \leq d_i + B$ for each task $T_i \in \tau$ and for each release of a job $J_{i,j}$.

4.1.2 Schedulability Test

The *feasibility* of task sets and *optimality* of scheduling algorithms are both defined in terms of schedulability. A task set is said to be feasible if there exist a scheduling algorithm that could schedule it, and a scheduling algorithm is optimal if it can successfully schedule all feasible task sets.

Definition 4.3. A task set τ is HRT (respectively, SRT) *feasible* if and only if there exists a scheduling algorithm $\mathcal{A}$ such that τ is HRT (respectively, SRT) schedulable under $\mathcal{A}$, with respect to an implementation on a given platform.

Definition 4.4. A scheduling algorithm $\mathcal{A}$ is optimal in an HRT (respectively, SRT) sense if and only if every task set τ that is HRT (respectively, SRT) feasible is HRT (respectively, SRT) schedulable under $\mathcal{A}$, with respect to an implementation on a given platform.

The *schedulability test* for a scheduling algorithm $\mathcal{A}$ is to determine *a priori*, during the design phase whether a task set will be schedulable under $\mathcal{A}$, either in a HRT or SRT sense, when implemented on a given platform.

4.2 Real-Time Scheduling

In general, a scheduler is responsible for assigning the pending job to processors if the number of pending jobs exceeds the number of processors *m*. A *static scheduler* uses a priority table dedicated to a task set as an allocation policy to assign the pending jobs to processors. In contrast, *dynamic scheduler* makes online scheduling decisions based on the current system state such as the set of pending jobs and their parameters.

4.2.1 Fixed-Priority Scheduling

In the fixed-priority scheduling (FP), tasks are in general indexed in order of decreasing priority. The question is then how to determine the priority of tasks in a task set. As an answer, Liu and Layland (1973) proposed the *rate-monotonic* (RM) priority assignment. Under RM scheduling, the priority of a task is determined by its period. Hence, given the periods p_i and p_j, of the tasks T_i and T_j respectively, if $p_i < p_j$, then the priority of task T_i is higher than the priority of task T_j.

Theorem 4.1 (Liu and Layland (1973))**.** *On a uniprocessor, a set of n implicit-deadline*[1] *periodic tasks $\tau = \{T_1, ..., T_n\}$ is schedulable under RM scheduling if $u_{sum}(\tau) \leq n(2^{1/n} - 1)$.*

The limit $n(2^{1/n} - 1)$ converges to $ln2 \approx 0.69$ for $n \to \infty$. Liu and Layland demonstrated that this is the highest-achievable utilization bound for RM.

	$prio_i$	e_i	p_i	u_i		
T_1	1	1	4	$\frac{1}{4}$	$= \frac{45}{180}$	≈ 0.25
T_2	2	1	5	$\frac{1}{5}$	$= \frac{36}{180}$	≈ 0.20
T_3	3	3	9	$\frac{1}{3}$	$= \frac{60}{180}$	≈ 0.33
T_4	4	3	18	$\frac{1}{6}$	$= \frac{30}{180}$	≈ 0.17
$u_{\text{sum}}(\tau)$						≈ 0.95

Table 4.1: Example of real-time task set schedulable under RM scheduling.

Example 4.1. Figure 4.2 shows the scheduling of the task set defined in Table 4.1 according to the RM scheduler. All tasks are released at time 0. At time 4, the job $J_{3,1}$ is preempted by the job $J_{1,2}$ because task T_1 has a higher priority than task T_3. Job $J_{4,1}$ starts execution only at time 7 when all higher priority tasks completes their first jobs. However it completes just before its deadline at time 18. ◊

Note that by applying Theorem 4.1 to the task set from table Table 4.1 we obtain the bound $4(2^{1/4} - 1) \approx 0.76$. This limit is lower than $u_{\text{sum}}(\tau) \approx 0.95$, which exceeds the limit given by the theorem, but the task set seems to be schedulable in the example shown in Figure 4.2. This is because theorem Theorem 4.1 is not an equivalence.

[1] An *implicit deadlines* task set τ, is characterized by the property of $d_i = p_i$, for each $T_i \in \tau$.

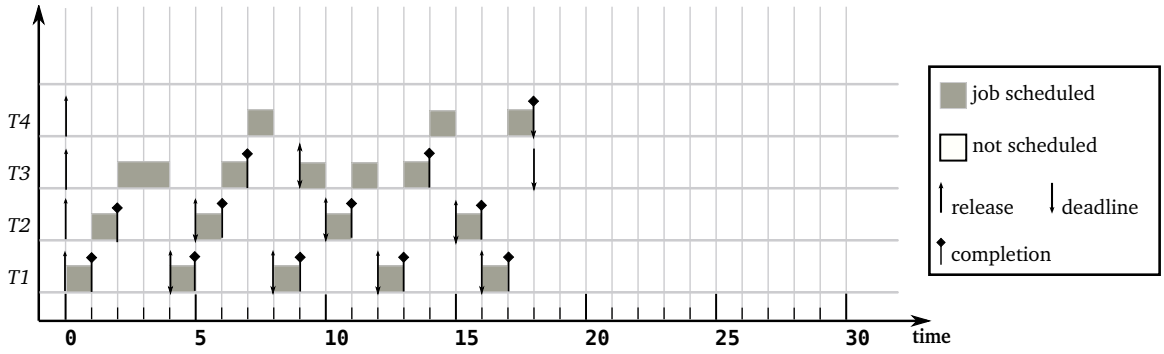


Figure 4.2: Scheduling of the task set from Table 3.1 according to the RM algorithm.

Theorem 4.2 (Joseph and Pandya (1986)). *Let $\tau = \{T_1, ..., T_n\}$ denote a set of constrained-deadline*[2] *sporadic*[3] *tasks indexed in order of decreasing priority. On a uniprocessor, under FP scheduling, the response time r_i of task $T_i \in \tau$ is bounded by the smallest R_i, where $R_i \leq e_i$, that satisfies the following equation:*

$$R_i = e_i + \sum_{h=1}^{i-1} \left\lceil \frac{R_i}{p_h} \right\rceil \cdot e_h.$$

Each R_i can be computed by using e_i as an initial value for R_i and by repeatedly re-evaluating the right-hand side until it and the left-hand side converge. Convergence is guaranteed as long as $u_{\text{sum}}(\tau) \leq 1$ (Joseph and Pandya, 1986).

Example 4.2. Consider the task set τ from Example 4.1 with parameters as given in Table 4.1. Applying Theorem 4.2 to each $T_i \in \tau$ yields the following response-time bounds:

$$\begin{aligned}
R_1 &= e_1 \\
R_2 &= e_2 + \left\lceil \frac{R_2}{p_1} \right\rceil \cdot e_1 \\
R_3 &= e_3 + \left\lceil \frac{R_3}{p_1} \right\rceil \cdot e_1 + \left\lceil \frac{R_3}{p_2} \right\rceil \cdot e_2 \\
R_4 &= e_4 + \left\lceil \frac{R_4}{p_1} \right\rceil \cdot e_1 + \left\lceil \frac{R_4}{p_2} \right\rceil \cdot e_2 + \left\lceil \frac{R_4}{p_3} \right\rceil \cdot e_3
\end{aligned}$$

Replacing the parameters by their actual values and iterating from $R_i = 1$ we obtain:

[2] A *constraint-deadlines* task set τ is characterized by $d_i \leq p_i$ for each $T_i \in \tau$.

[3] The *periodic task model* is generalized by the *sporadic task model* (Mok, 1983). In this case, the jobs of a task T_i are not released at fixed time multiple of the period p_i of the task, instead, the jobs are released at least p_i time units and not necessarily a multiple of p_i.

$$
\begin{aligned}
&R_1 = 1 \\
&R_2 = 1 \implies 1 + \left\lceil \frac{1}{4} \right\rceil \cdot 1 = 2 \neq 1 \\
&R_2 = 2 \implies 1 + \left\lceil \frac{2}{4} \right\rceil \cdot 1 = 2 \\
&R_3 = 1 \implies 3 + \left\lceil \frac{1}{4} \right\rceil \cdot 1 + \left\lceil \frac{1}{5} \right\rceil \cdot 1 = 5 \neq 1 \\
&R_3 = 2 \implies 3 + \left\lceil \frac{2}{4} \right\rceil \cdot 1 + \left\lceil \frac{2}{5} \right\rceil \cdot 1 = 5 \neq 2 \\
&R_3 = 3 \implies 3 + \left\lceil \frac{3}{4} \right\rceil \cdot 1 + \left\lceil \frac{3}{5} \right\rceil \cdot 1 = 5 \neq 3 \\
&R_3 = 4 \implies 3 + \left\lceil \frac{4}{4} \right\rceil \cdot 1 + \left\lceil \frac{4}{5} \right\rceil \cdot 1 = 5 \neq 4 \\
&R_3 = 5 \implies 3 + \left\lceil \frac{5}{4} \right\rceil \cdot 1 + \left\lceil \frac{5}{5} \right\rceil \cdot 1 = 6 \neq 5 \\
&R_3 = 6 \implies 3 + \left\lceil \frac{6}{4} \right\rceil \cdot 1 + \left\lceil \frac{6}{5} \right\rceil \cdot 1 = 7 \neq 6 \\
&R_3 = 7 \implies 3 + \left\lceil \frac{7}{4} \right\rceil \cdot 1 + \left\lceil \frac{7}{5} \right\rceil \cdot 1 = 7 \\
&R_4 = 18 \implies 3 + \left\lceil \frac{18}{4} \right\rceil \cdot 1 + \left\lceil \frac{18}{5} \right\rceil \cdot 1 + \left\lceil \frac{18}{9} \right\rceil \cdot 3 = 18
\end{aligned}
$$

As we can see from the application of the theorem that the task's response time is equal to the actual response time of its first job in Figure 4.2. This is due to the fact that the newly-released job's response time is maximized when all tasks release a job at the same time, which is the case in the example of Table 4.1 where all tasks release jobs at time 0. Given that $R_i \leq d_i = p_i$ for each T_i, the task set is HRT schedulable under RM scheduling. $\Diamond$

4.2.2 Dynamic-Priority Scheduling

One of the most known example of dynamic priority real-time scheduling is the *earliest-deadline-first* policy. According to the EDF policy, a job is prioritized by its absolute deadline. If there are multiple equal absolute deadlines, the jobs are then selected using their index in the list of the ready tasks.

Theorem 4.3 (Liu and Layland (1973); Liu (1969))**.** *On a uniprocessor, a set of n implicit-deadline periodic tasks $\tau = \{T_1, ..., T_n\}$ is HRT schedulable under EDF if $u_{sum}(\tau) \leq 1$.*

Example 4.3. Figure 4.3 depicts the EDF schedule of the task set shown in Table 4.1 that we previously used in Example 4.1. With the exception of the scheduling of the second job of task T_2, most of the scheduling decisions taken by EDF is similar to those taken by FP. At time 5, $Deadline(J_{3,1}, 5) = 9 < 10 = Deadline(J_{2,2}, 5)$, which results in $J_{3,1}$ having a higher priority than $J_{2,2}$. Thus, the allocations of the jobs on the processor are switched in comparison with the FP schedule depicted in Figure 4.2. ◊

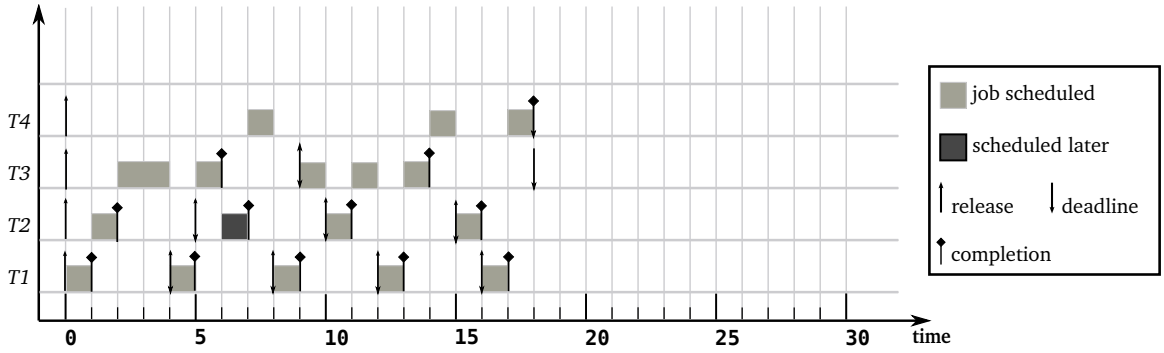


Figure 4.3: Scheduling of the task set from Table 3.1 according to the EDF policy.

4.3 Algorithmic Analysis

kvm and the Linux host scheduler employ the POSIX SCHED_FIFO (*fixed-priority first-in first-out*) algorithm when scheduling the virtual machines. While this policy is efficient in the case where there is only one virtual machine that is running on a CPU, it could create a problematic situation in the case where there are multiple virtual machines that share the CPU. Thereby, preventing a real-time system from executing correctly.

To illustrate this observation, we experimented the situation of two virtual machines sharing the same CPU as presented in Figure 4.4. We set to VM^1 and VM^2 the same priority. VM^1 is executing one periodic task ($500ms$, $1000ms$) and VM^2 is executing an endless while loop.

According to the POSIX SCHED_FIFO algorithm, when VM^1 is scheduled by the host OS, it starts running and executes the code of the guest operating system. Then the scheduler of the guest OS schedules the periodic task. After the completion of the first job of the task, the guest operating

system arms a timer for the next release of the periodic task, enqueues it to the waiting list and switches to the idle task.

Figure 4.4: Scheduling of virtual machines according to SCHED_FIFO scheduling algorithm. The process "qemu-system-x86-2108 " is the vCPU of VM^1 and "qemu-system-x86-2202 " represents the vCPU of VM^2. This experiment shows how the VM^1 process is starved by the VM^2 process because it never releases the CPU.

At this time, the host OS detected the internal idle state of the guest operating system, preempts VM^1 process and schedules VM^2. Conforming to the Linux's documentation, there is no time slice in the SCHED_FIFO scheduling algorithm, that is, a process is allowed to execute until it explicitly releases the CPU or be preempted by a higher priority process. This results in the situation where the CPU is not allocated to VM^1, and the periodic task is never executed.

As a side note from the observation of the scheduling diagram in Figure 4.4, we can better understand the reason of the rare very high overhead and latency measured in the previous chapter. If we assume that the guest OS running on VM^1 was executing a routine such as the scheduling, it is easy to see how this routine would be delayed by the execution of another workload on the host (in this case VM^2). While in this extreme situation the overhead would be extremely high, we conjecture that what happened in the rare cases where we recorded a very high overhead and latency the situation was similar but in which a process executed by the host OS would delay the execution of the guest OS by a more reasonable amount of time.

This problem could be resolved by adopting a scheduling method that enforces the temporal isolation between the virtual machines. Such a scheduling method defines for each virtual machine

a tuple (Θ, Π), where the *budget* Θ and the *period* Π together represent the CPU share that a VM requests. The VM will receive at least Θ units of time in each period of length Π.

Figure 4.5 shows an example of the algorithm that shares the CPU between two virtual machines. For example, by assigning $(\Theta = 2, \Pi = 4)$ to each virtual machine and setting a higher priority to VM^1 than VM^2, 50% of the CPU time is allocated to VM^1 and VM^2. Given these temporal parameters, the algorithm prevents VM^1 from over utilizing the CPU resources after consuming its budget and allocates to VM^2 the remaining CPU time. Hence it allows the real-time task T_2 to run and respect all its deadline.

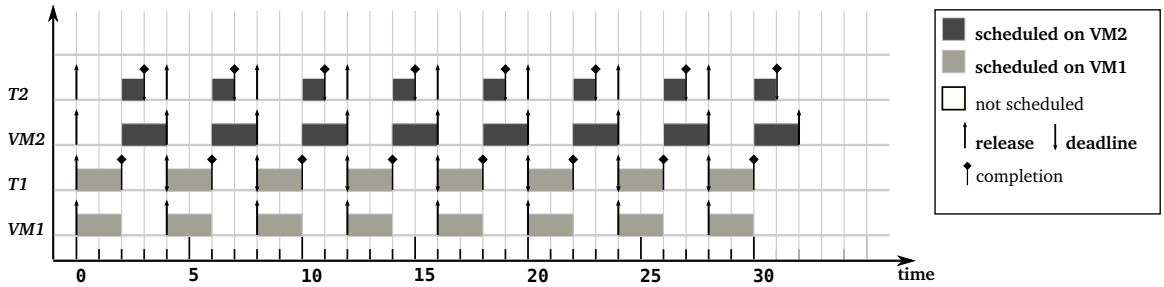


Figure 4.5: Scheduling of virtual machines according to the RM algorithm.

4.4 Computing of the Efficient Scheduling Parameters

The simple scheduling scenario illustrated in Figure 4.5 demonstrated the ability of an algorithm that uses the periodic resource model (Θ, Π) to ensure the temporal isolation between the VMs. From this scenario, we can intuitively deduce the requirements in terms of budget and period for a virtual machine to guarantee the real-time requirement.

In general terms, we denote $\mathbf{V}$ the set of all virtual machines, V_l, in the system. Each V_l is assigned a budget Θ_l and a period Π_l. The VM scheduler then allows every VM to run a maximum amount of time Θ_l every Π_l time units. In the simple case where a VM is executing one real-time task, the budget and period can be set according to the task parameter. And the inequality, $\Theta_l \leq \Pi_l$, must hold for all VMs in order to respect all the timing requirements.

However, in the other case where a virtual machine executes multiple real-time tasks, the period and the budget for each VM must be calculated in order to respect the *schedulability* of all the tasks, and the *optimal utilization* of the CPU resources.

Masrur et al. (2011) developed an analytical method to compute the efficient budget and period of a virtual machine. In the following sections we provide the intuition and the equations of the method, and we refer the interested reader to (Masrur et al., 2010, 2011) for a detailed discussion of the mathematical proof.

4.4.1 Execution Length of a Virtual Machine

The *execution length* is the largest amount of time that it takes a virtual machine to execute its assigned budget. The execution length depends on the scheduling of virtual machines, impacts the schedulability of real-time tasks running on them, and is the first parameter to define in the schedulability analysis.

First, we consider that the virtual machines are scheduled according to *fixed-priority rate-monotonic* (RM) algorithm. Second, we assume that every virtual machine V_l can finishes executing its assigned budget Θ_l within Π_l time units from its release.

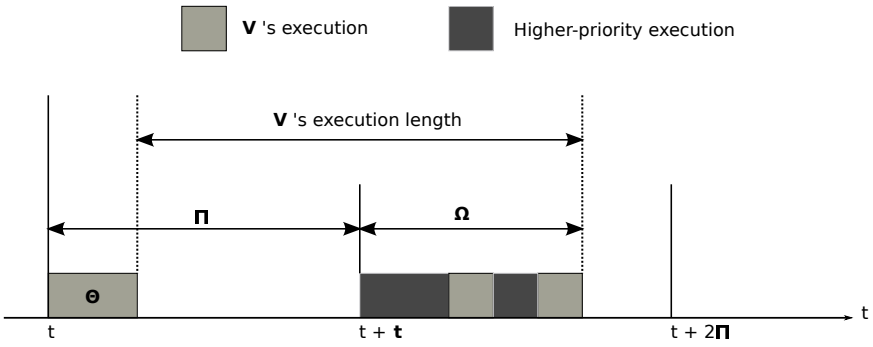


Figure 4.6: V_l execution length.

As we can see in Figure 4.6, the execution length of a VM depends on the interference from the execution of the other higher-priority VMs. To determine the time at which the VM finish executing, we can use the worst-case response time analysis (Lehoczky et al., 1989; Audsley et al., 1993):

$$t^{(c+1)} = \Theta_l + \sum_{j=1}^{l-1} \left\lceil \frac{t^{(c)}}{\Pi_j} \right\rceil \cdot \Theta_j. \tag{4.1}$$

According to the rate-monotonic algorithm, the VMs with shorter periods are the highest priority VMs. Then, using this policy, the VMs are sorted by decreasing priority. That is, the highest priority VMs are scheduled before the lowest priority VMs. Thus, the second term at the right of

Equation (4.1) corresponds to all higher-priority VMs in the system. This equation can be verified iteratively starting from $t^{(1)} = \Theta_l$ and until $t^{(c+1)} = t^{(c)}$ is satisfied for some $c \geq 1$. The value of $t^{(c+1)}$ is equal to the V_l's worst-case response time, denoted by Ω_l. From Figure 4.6 we define the V_l's execution length L_l by:

$$L_l = \Pi_l - \Theta_l + \Omega_l. \tag{4.2}$$

We denote by $d_{l,min} = \min_{i=1}^{|\tau_l|}(d_i)$ the smallest deadline in the task set τ_l, where $|\tau_l|$ is the number of tasks in τ_l. And we denote by $e_{l,min}$ the worst-case execution time of the task with $d_{l,min}$. The task with $d_{l,min}$ is the highest priority task in V_l and its execution is not interrupted once V_l starts running.

We assume that Θ_l is at least equal to $e_{l,min}$:

$$e_{l,min} \leq \Theta_l. \tag{4.3}$$

Given the precedent two equations we can derive a *necessary condition* to ensure that V_l guarantee that all $d_{l,min}$ are respected, that is, the execution length L_l must be less than or equal to $d_{l,min}$:

$$\Pi_l - \Theta_l + \Omega_l \leq d_{l,min}. \tag{4.4}$$

4.4.2 Schedulability Condition on a VM

After defining the condition that allows a VM to meet all deadlines of the highest-priority task, in this section we analyze the schedulability of a real-time task set running in a VM.

The *worst-case execution demand* of a task $T_i \in \tau$, within d_i time units is denoted by ω_i and given by:

$$\omega_i = e_i + \sum_{j=1}^{i-1} \left\lceil \frac{d_i}{p_j} \right\rceil \cdot e_j. \tag{4.5}$$

If we assume that all tasks in τ_l are sorted by decreasing priority, which corresponds to increasing period under RM algorithm, then, the second term at the right of Equation (4.5) determines the worst-case execution demand of all the $(i-1)$ higher-priority tasks on V_l.

And if we assume that the worst-case execution demand within d_i time units of a task $T_i \in \tau_l$, is less than or equal to d_i, that is, $\omega_i \leq d_i$. Then, the necessary *schedulability condition* for a task T_i to meet its deadline on V_l is:

$$k_{l,i} \cdot \Theta_l + min(\Theta_l, \alpha_l(t_i - k_{l,i} \cdot \Pi_l)) \geq \omega_i, \tag{4.6}$$

where t_i is equal to $d_i - (\Pi_l - \Theta_l)$ and $k_{l,i}$ is computed by $\left\lfloor \frac{t_i}{\Pi_l} \right\rfloor$. The function $\alpha_l(t)$ returns the amount of time that V_l is able to run in a time interval of length t. This function takes into account that V_l is released together with all higher-priority VMs at the beginning of the interval of length t.

To understand the *schedulability condition* let us examine the inequality in the simple case where $\min(\Theta_l, \alpha_l(t_i - k_{l,i} \cdot \Pi_l)) = 0$, then, we obtain this inequality:

$$k_{l,i} \cdot \Theta_l \geq \omega_i. \tag{4.7}$$

As we assumed that $\omega_i \leq d_i$, and if V_l can execute ω_i time units before d_i expires, then the previous condition holds, and the task T_i respects its deadline. Figure 4.7 illustrates this simple example where the virtual machine is scheduled at time $t + \Pi$, and finishes executing before the expiration of the deadline d_i of a task T_i.

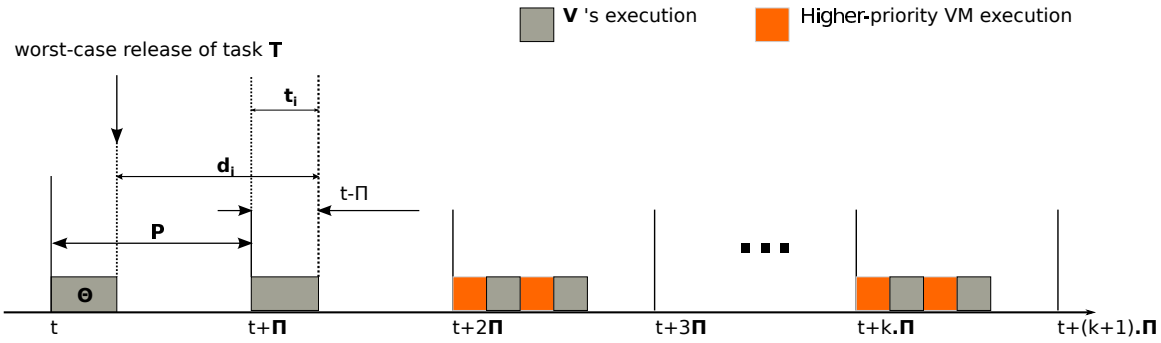


Figure 4.7: Schedulability condition of task T_i, in the case where the virtual machine V_l executes its time slice Θ_l before the expiration of the deadline d_i of task T_i, and this time slice covers the worst-case execution demand of task T_i.

Figure 4.8 illustrates a second example, in which the execution of the VM V_l is interleaved by the execution of a higher priority VM. In this case the term $\min(\Theta_l, \alpha_l(t_i - k_{l,i} \cdot \Pi_l))$ depends on the value returned by $\alpha_l(t)$. The function $\alpha_l(t)$ returns the maximum value of a variable E_l that can

be calculated using the following equation:

$$t^{(c+1)} = E_l + \sum_{j=1}^{l-1} \left\lceil \frac{t^{(c)}}{\Pi_j} \right\rceil \cdot \Theta_j. \tag{4.8}$$

The value of E_l is at maximum equal to Θ_l because V_l could not execute more than Θ_l during an interval of time equal to Π_l. Again, as we assumed that $\omega_i \leq d_i$, and if V_l can execute ω_i time units before d_i expires, then the schedulability condition holds, and the task T_i will respect its deadline.

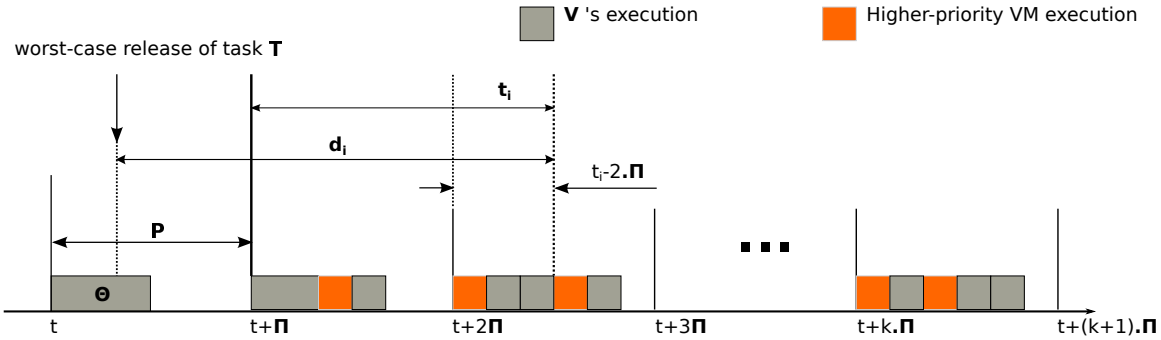


Figure 4.8: Schedulability condition of task T_i, in the case where the execution of virtual machine V_l is interleaved by the execution of higher priority VM. Even though there is an interference the virtual machine V_l completes the execution of its time slice Θ_l before the expiration of the deadline d_i of task T_i, and this time slice covers the worst-case execution demand of task T_i.

Figure 4.9 illustrates the general case, as we can see V_l is executed $k_{l,i}$ times within t_i. The term $\min(\Theta_l, \alpha_l(t_i - k_{l,i} \cdot \Pi_l))$ represents the V_l's additional execution time in the time interval $[t_i - k_{l,i} \cdot \Pi_l]$ assuming the worst-case situation, that is, V_l is released at the same time with higher priority VMs.

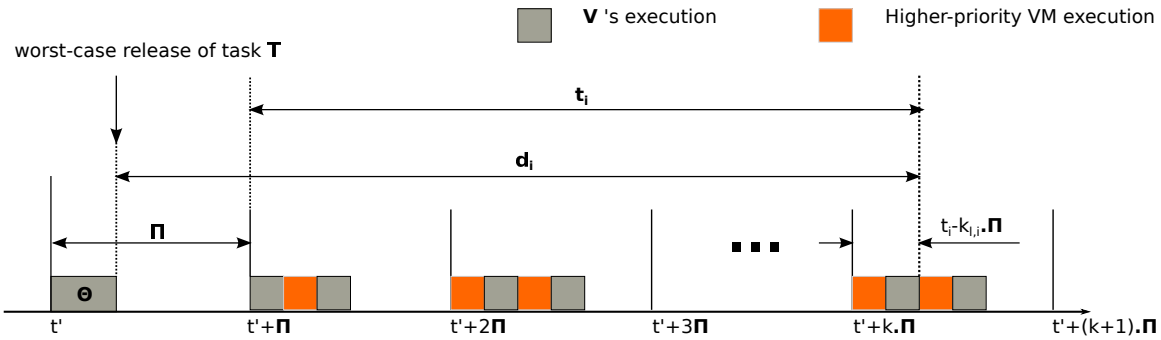


Figure 4.9: Schedulability condition of task T_i executed on the virtual machine V_l.

And given that the V_l worst-case execute time is Θ_l in a period Π_l, and $t_i - k_{l,i} \cdot \Pi_l < \Pi_l$ (see Figure 4.9), the V_l additional execution time is lower than, or equal to Θ_l at maximum.

4.4.3 Computing of the Highest-Priority VM's Parameters

Using the minimum execution length of a VM and the schedulability condition, we present in this section the method to calculate the budget Θ_l and the period Π_l of a virtual machine V_l such that all deadlines are respected.

Assuming that all VMs are scheduled under RM algorithm, that is, Π_l specifies the priority of a V_l. So, if Π_l verifies the Equation (4.4), then V_l is able to schedule the task with the minimum deadline $d_{l,min}$. And therefore, V_l is the highest priority VM.

Now, the idea of the algorithm is to calculate the period and the budget of a V_l in order to respect the deadline of all the tasks. Starting by calculating the parameters of the highest priority VM, then to continue with the lower priority VMs according to decreasing priorities. The reason for this order stems from the fact that the algorithm uses the worst-case response time analysis, in which the computing of the lower priority VM's parameters depends on the highest priority VM's values.

Given that the VMs are sorted by decreasing priority, we set the virtual machine V_1 as the highest priority VM. We assigned to V_1 the highest priority task that has the minimum deadline $d_{1,min}$. The budget of V_1 is Θ_1 and represents its worst-case response time Ω_1 because there is no interference with other VMs when V_1 starts executing which results in $L_1 = \Pi_1$, and using Equation (4.2) we obtain:

$$\Pi_1 = \Pi_1 - \Theta_1 + \Omega_1. \tag{4.9}$$

By selecting Θ_1 to be equal to $e_{1,min}$, the worst-case execution time of the task with minimal deadline, it is clear that for the highest priority task to respect its deadline, Π_1 must verify the necessary condition given by Equation (4.4). So, by replacing in Equation (4.4) the new value of Π_1 we obtain:

$$\Pi_1 - \Theta_1 + \Omega_1 \leq d_{l,min} \tag{4.10}$$

$$\Pi_1 - \Theta_1 + \Omega_1 - \Theta_1 + \Omega_1 \leq d_{l,min} \tag{4.11}$$

$$\Pi_1 \leq d_{1,min}. \tag{4.12}$$

After setting Π_1 to be equal to $d_{1,min}$, the idea is to recompute the value of Θ_1 in order for V_1 to guarantee a deadline $d_i \geq d_{1,min}$. In the case of V_1, the function $\alpha_1(t_i - k_{1,i} \cdot \Pi_1)$ reduces to $t_i - k_{1,i} \cdot \Pi_1$ because V_1 is the highest priority VM and there is no interference from other VMs. So, by replacing the values of Θ_1 and Π_1 in Equation (4.6) we obtain:

$$k_{1,i} \cdot \Theta_1 + \min(\Theta_1, \alpha_l(t_i - k_{1,i} \cdot \Pi_1)) \geq \omega_i. \tag{4.13}$$

If we suppose that the term $\min(\Theta_1, \alpha_l(t_i - k_{1,i}.\Pi_1))$ is equal to zero, and replacing $k_{1,i}$ by $\left\lfloor \frac{t_i}{\Pi_l} \right\rfloor$ where $t_i = d_i - (\Pi_1 - \Theta_1)$, we obtain:

$$\left\lfloor \frac{d_i - (\Pi_1 - \Theta_1)}{\Pi_1} \right\rfloor \cdot \Theta_1 \geq \omega_i. \tag{4.14}$$

After removing the floor function, and transforming the inequation we obtain the following quadratic equation on Θ_1:

$$(\Theta_1)^2 + (d_i - \Pi_1) \cdot \Theta_1 - \omega_i \cdot \Pi_1 = 0. \tag{4.15}$$

The solutions for this quadratic equation could be positive or negative. If at least one solution is positive, it means that a task T_i could be scheduled on V_1. The solution found is an approximation of Θ_1, that allows the scheduling of T_i. To verify that the new values of Θ_1 are optimal or not, we can use Equation (4.6). If $k_{1,i} \cdot \Theta_1 + \min(\Theta_1, \alpha_l(t_i - k_{1,i}.\Pi_1)) == \omega_i$, then the solution is a minimum possible budget that allows the scheduling of T_i. Otherwise it can be reduced without affecting the scheduling of T_i.

In the case where the new value of Θ_1 do not verify Equation (4.6), the value of Θ_1 need to be increased, while respecting the condition of being lower than or equal to Π_1.

So, if the solution of Equation (4.15) is not the minimal value of Θ_1, or the value of Θ_1 does not verify Equation (4.6), we can replace in the following Equation the solution found in order to calculate the difference between the approximation of Θ_1 given by the solution, and the correct value of Θ_1:

$$\Delta_{\Theta_1} = \omega_i - k_{1,l} \cdot \Theta_1 - min(\Theta_1, t_i - k_{1,i} \cdot \Pi_1). \tag{4.16}$$

If Δ_{Θ_1} is positive then the Θ_1 needs to be increased, and if it is negative then Θ_1 could be decreased.

An optimization could also be applied in order to distribute the Δ_{Θ_1} between all the V_1 executions before d_i. This could done by: first calculating the number of times that V_1 executes before d_i. Then, recomputing Θ_1 by adding to the current approximation of Θ_1 found using the solution of the quadratic equation, the value of $\frac{\Delta_{\Theta_1}}{\eta_1}$, where η_1 is:

$$\eta_1 = k_{1,i} + \left\lceil \frac{min(\Theta_1, t_i - k_{1,i} \cdot \Pi_1)}{\Theta_1} \right\rceil. \tag{4.17}$$

The same principle could be applied to calculate the parameters of a lower priority VM (Masrur et al., 2011).

Example 4.4 illustrates the use of the precedent equations to compute the budget and period of a highest priority VM.

	$priority_i$	e_i	p_i	d_i	u_i		
T_1	1	1	5	2.5	$\frac{1}{5}$	$= \frac{8}{40}$	≈ 0.20
T_2	2	2	5	5	$\frac{2}{5}$	$= \frac{16}{40}$	≈ 0.40
T_3	3	1	20	7	$\frac{1}{20}$	$= \frac{2}{40}$	≈ 0.05
T_4	4	3	20	10	$\frac{3}{20}$	$= \frac{6}{40}$	≈ 0.15
T_5	5	4	40	40	$\frac{4}{40}$	$= \frac{4}{40}$	≈ 0.10
$u_{\text{sum}}(\tau)$							≈ 0.90

Table 4.2: Task set of the simplistic automotive applications as proposed by (Masrur et al., 2011). The first two tasks implements the Electronic Stability Control software (ESC), and the second two tasks implements the Engine Management software (EM).

Example 4.4. The example of the task set in Table 4.2 represents two applications, the first application includes tasks T_1, T_2, and the second application includes tasks T_3, T_4, and T_5.

We design the scheduling by running the ESC application on VM^1 and the EM application on VM^2. We give to VM^1 a higher priority than VM^2.

To compute the budget and the period for VM^1 we use the parameters of task T_1 because it is the highest priority task, thus $\Theta_1 = 1$ and $\Pi_1 = 2.5$. Now, to ensure that T_2 could be scheduled by VM^1, we need to recompute the Θ_1 using the quadratic Equation (4.15). But we need first to compute ω_2, the worst-case execution demand of task T_2 using Equation (4.5). In this case the ($\omega_2 = 1 + 2 = 3$) because only tasks T_1 and T_2 are running on VM^1, and task T_2 can only be delayed by the execution of task T_1.

Using the deadline of task T_2, $d_2 = 5$ and replacing in Equation (4.15) we obtain:

$$\Theta^2 + (5 - 2.5) \cdot \Theta - 3 \cdot 2.5 = 0.$$

This equation has a negative root, -4.26, and a positive root, 1.76. Obviously, only the positive root could be used. Now the new value of Θ_1 is 1.76. Next, we need to verify that this value respects the schedulability condition. Thus we need to compute $t_2 = d_2 - (\Pi_1 - \Theta_1)$ and $k_{1,2} = \left\lfloor \frac{t_2}{\Pi_1} \right\rfloor$. Which results in $t_2 = 5 - (2.5 - 1.76) = 4.26$ and $k_{1,2} = \frac{4.26}{2.5} = 1$.

To verify that the schedulability condition holds, we replace in Equation (4.6):

$$\begin{aligned} 1 \cdot 1.76 + \min(1.76, 4.26 - 1 \cdot 2.5) &\geq 3, \\ 3.52 &\geq 3. \end{aligned}$$

As the comparison holds, and $\Delta_{\Theta_1} = 3 - 3.52 = -0.52$ is negative, we need to increase Θ_1 by Δ_{Θ_1}. But instead of adding the complete amount of time Δ_{Θ_1} to Θ_1, this amount could be distributed by a number of times that VM^1 need to execute before d_2. The number is given by:

$$\begin{aligned} \eta_1 &= k_{1,i} + \left\lceil \frac{\min(\Theta_1, t_i - k_{1,i} \cdot \Pi_1)}{\Theta_1} \right\rceil \\ \eta_1 &= 1 + \left\lceil \frac{\min(1.76, 426 - 1 \cdot 2.5)}{1.76} \right\rceil \\ \eta_1 &= 2 \end{aligned}$$

Using η_1 we recompute the new value of Θ_1:

$$\Theta_1 = 1.76 + \frac{(-0.52)}{2} = 1.5$$

Verifying again the value of Θ_1 in Equation (4.6) :

$$1 \cdot 1.5 + \min(1.5, 4.26 - 1 \cdot 2.5) \geq 3,$$
$$3 == 3.$$

As we can see, the new value is minimum possible value of Θ_1, and in order for VM^1 to schedule tasks T_1 and T_2, its parameters must be set as $\Theta_1 = 1.5$ and $\Pi_1 = 2.5$. $\Diamond$

4.5 Overhead-aware Schedulability Analysis

In the previous chapter we measured the overheads and latencies of a guest RTOS. However, the theoretical method presented in the the previous sections does not take into account the overhead observed in practice. In this section, we integrate these overheads into a schedulability analysis.

Our measurement indicates that in practice the execution of a task is delayed by a set of overheads and latencies. Figure 4.10 illustrates the timeline of a task's release event. As we can see the task's execution is delayed by its own event latency (Δ^{event}), release overhead (Δ^{rel}), scheduling overhead(Δ^{sched}), and context switch overhead (Δ^{cxs}). These overheads and latencies need to be accounted for the execution time of a real-time task. We define the overhead related to a release event by:

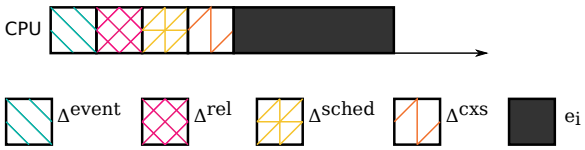


Figure 4.10: Overhead related to release event. The execution of a task T_i is delayed by the overheads and latencies internal to the RTOS when it is released.

$$\Delta^{\mathsf{relEv}} = \Delta^{event} + \Delta^{rel} + \Delta^{sched} + \Delta^{cxs}$$

Consequently, the task's execution time needs to be inflated by this overhead as follows:

$$e'_i = e_i + \Delta^{\mathsf{relEv}}$$

This method is then used to recompute the parameters of all the tasks τ_i in a workload $\tau = \{\tau_1, \tau_2, \cdots, \tau_n\}$ of each component C in the system. Using these inflated tasks' parameters and the method presented in the previous section we can compute the periodic resource model $\Gamma = (\Theta, \Pi)$ for each component C. To validate this overhead-aware schedulability analysis we use Lemma 4.1:

Lemma 4.1 (Phan et al. (2013))**.** *A Component $C = \langle \tau, \mathcal{A} \rangle$ is schedulable by a periodic resource model Γ in presence of inflatable overheads if its inflated workload τ' is schedulable by Γ under the algorithm $\mathcal{A}$ when there are zero overheads.*

The proof of the Lemma 4.1 relies on the overhead accounting technique. As we have shown above, the WCET e'_i of a task is composed by the original WCET e_i and the Δ^{relEv}. This means that the task's WCET never exceeds e'_i in presence of overhead. Suppose that the inflated execution time e'_i of all tasks in τ' are used to compute the periodic resource model $\Gamma = (\Theta, \Pi)$, and this PRM verifies the schedulability condition in a VM (see Equation (4.6)), that is, all tasks in τ' meet their deadline, then τ' is schedulable under algorithm $\mathcal{A}$. So, if τ' is schedulable under $\mathcal{A}$ assuming zero overhead, then τ is schedulable under $\mathcal{A}$ in presence of inflatable overheads.

From our overhead measurement we used the worst-case observed value when executing 20 tasks per processor as an estimation of the overheads and latencies. Thus, Δ^{sched} equals to 75μs, Δ^{event} equals to 250μs, Δ^{cxs} equals to 20μs, and Δ^{rel} equals to 40μs. In total, the Δ^{relEv} equals to 385μs, could be used to inflate the tasks' execution time.

In the next section, we present an implementation of the periodic resource model, and we use the overhead-aware schedulability analysis to compute the PRM parameters $\Gamma = (\Theta, \Pi)$ for each components in the system. Then we experiment this method using a case-study real-time application.

4.6 Empirical Evaluation

In order to evaluate the presented method it is necessary to implement the periodic resource model (PRM), because this model permits to assign a budget Θ and a period Π to the virtual machines, and thus guarantees that the resources required by each virtual machine will be provided by the host. To experiment the use of the periodic resource model on top Linux kvm we used two different implementations: the Vsched user-level library and SCHED_DEADLINE real-time scheduling class internal to Linux kernel.

The Vsched library (Lin and Dinda, 2005) is a user-level library that implements an EDF real-time scheduling algorithm that co-exists with the default scheduling classes of Linux. The Vsched library allows to attribute to each virtual machine a PRM interface (Θ, Π).

The SCHED_DEADLINE (LWN, 2014) scheduling class is an implementation of the EDF scheduling algorithm in Linux, this policy is reinforced by the *Constant Bandwidth Server* mechanism that ensures the temporal isolation between processes using a PRM interface and thus prevents a misbehaving process from affecting the correctness of the others.

In a first experiment we used Vsched to run two virtual machines on the same CPU in order to verify that the temporal isolation is guaranteed among the VMs. We reused the same test-case that we already presented in Section 4.3 when we analyzed the problem of the scheduling of virtual machines (see Figure 4.4). In that case we demonstrated how one virtual machine was exposed to a starvation problem when executed on the same CPU with a second virtual machine that has the same priority. The virtual machines were scheduled using the SCHED_FIFO real-time scheduling algorithm. The experiment consisted of two virtual machines, VM^1 was executing a periodic task $T_1(500ms, 1000ms)$ and VM^2 was executing a process that performs an endless while() loop.

We repeated this experiment but we configured the PRM interface for each virtual machine in a way that ensures the proportional share of the CPU resource between the two VMs, each virtual machine was executed for $200ms$ every $250ms$, VM^1 ($200ms$, $250ms$) and VM^2 ($200ms$, $250ms$). We ran the workload for 10 one-minute runs. Figure 4.11 shows the correctness of the scheduling.

Moreover, we evaluated the real-time performance of the guest OS running on VM^1. We measured the *deadline miss ratio* metric, which is the number of deadline misses divided by the total

number of completed jobs of the periodic task. The obtained results indicate that no deadline was missed during the experiment.

Figure 4.11: Scheduling of virtual machines using the periodic resource model. The CPU was allocated in fair-share fashion to both virtual machines.

Application	Task	execution time	period
Task set 1	T_1	$300ms$	$1500ms$
	T_2	$500ms$	$2000ms$
Task set 2	T_3	$300ms$	$1200ms$
	T_4	$400ms$	$2400ms$

Table 4.3: Simplified real-time applications.

In a second experiment we tested two real-time applications. We designed the system to use two virtual machines, VM^1 executed task set 1 and VM^2 executed task set 2, the parameters of the tasks are given in Table 4.3. Each virtual machine was executing a real-time Linux. The recent integration of SCHED_DEADLINE to the mainline Linux allowed us to use it instead of Vsched as an implementation of the periodic resource model for the scheduling of virtual machines. Table 4.4 summarizes the platform setup.

Hardware	Intel core i7 2.6GHz VT-x 8GB RAM
VMM	kvm and Qemu
Host OS	Linux-3.14.rc6
Guest RTOS	Linux-3.4 PREEMPT_RT

Table 4.4: Real-Time Virtual Machine System configuration.

The PRM parameters of each virtual machine were computed based on the *necessary condition*, defined by Equation (4.4) and the *schedulability condition* defined by Equation (4.6). The

period of the virtual machines VM^1 and VM^2 was set according to the *necessary condition* using Equation (4.4):

$$\Pi_l \leq d_{i,min}.$$

Recall that $d_{i,min}$ is the deadline of the highest priority task T_i executed on a virtual machine V_l. As in our experiment we used the period of a task as its deadline, so, $d_{i,min}$ is $1500ms$ in the case of VM^1, and is $1200ms$ in the case of VM^2 (see Table 4.3). So, we assigned $\Pi_1 = 200ms$ for VM^1, and $\Pi_2 = 200ms$ for VM^2, in order to verify the *necessary condition*.

The budget of VM^1 and VM^2 was set according to the *schedulability condition* which is derived from Equation (4.6):

$$k_{l,i} \cdot \Theta_l \geq \omega_i,$$

where ω_i is the *worst-case execution demand* of a task T_i executed on the virtual machine that we calculated using Equation (4.5), and $k_{l,i}$ is the number of times that the virtual machine V_l needs to execute before the expiration of the deadline d_i of a task T_i, and calculated using: $k_{l,i} = \left\lfloor \frac{d_i - (\Pi_l + \Theta_l)}{\Pi_l} \right\rfloor$.

The same budget was set to both virtual machines, VM^1 ($160ms$, $200ms$) and VM^2 ($160ms$, $200ms$). This budget verifies the schedulability condition in both cases. For instance, in the case of VM^1, and task T_1 we have $k_{1,1} = \left\lfloor \frac{1500-(200+160)}{200} \right\rfloor = 5$, and $\omega_1 = 300$ because T_1 is the highest priority task in VM^1, then replacing this value in the *schedulability condition* we obtain:

$$5 \cdot 160 \geq 300$$

$$800 \geq 300$$

In the case of task T_2 the worst-case execution demand is $(\omega_2 = (300 + 500) = 800)$ because T_2 has lower priority than T_1 then could only execute after T_1 finishes executing, and $k_{1,2} = \left\lfloor \frac{2000-(200+160)}{200} \right\rfloor = 8$. These values verify the *schedulability condition*:

$$8 \cdot 160 \geq 800$$

$$1280 \geq 800$$

As we can see from the above inequations, the budget $\Theta = 160ms$ could be reduced without affecting the *schedulability condition*, however we kept its value sufficiently high to integrate the virtualization overhead which we estimated to be equal to $385\mu s$ in the previous section.

Note that the original theoritical method that we used to calculate the budget and the period of the virtual machines assumes that the virtual machines are scheduled according to a fixed-priority rate-monotinic scheduling. Where the virtual machine that execute the highest priority task would be assigned the highest priority. However, in our experimentation we did not affect a priority to any virtual machine because we used the SCHED_DEADLINE scheduling class in Linux since it is the only scheduling class that implements the PRM interface, and this scheduling class employs dynamic priority scheduling. Thus, in our setup we set the same budget and period to each virtual machine to force the scheduler to attribute the same priority to both virtual machines at runtime.

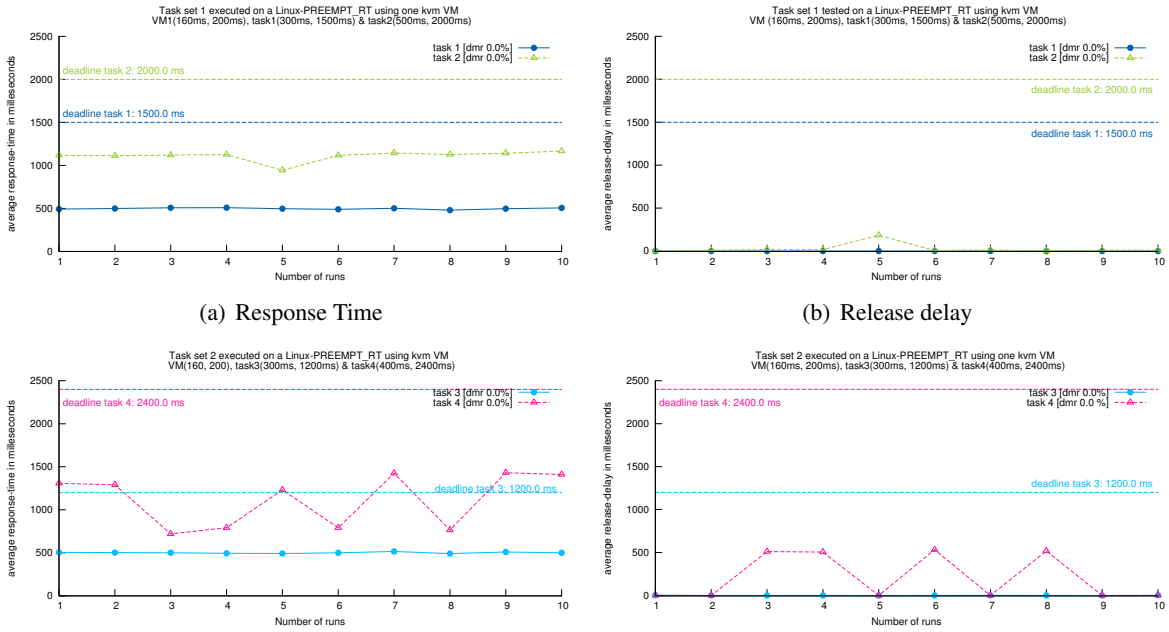
Periodic Task Model Implementation in Linux. We implemented the periodic real-time tasks using Linux processes. Knowing that the minimal scheduling tick (jiffy) in Linux kernel that could be configured is $1ms$, we used this quantum as lower execution bound for the real-time task. We first calibrated the amount of work that needs exactly $1\mu s$ on one core, and then scaled it to generate any workload specified at a millisecond resolution. Using POSIX interfaces, every task was scheduled using SCHED_FIFO algorithm, and the priority was set according to the *rate-monotonic* policy.

We used a real-time clock to trigger interrupts to release each job of a task, and recorded the first job release time. When each job finished, its finish time was recorded using the x86 RDTSC instruction, which reads the TSC register (timestamp counter) and provides the number of cycles since the boot of the machine.

After all tasks finished, we used the first job's release time to calculate every job's release time and deadline, and compared each deadline with the corresponding job's finish time. Then we calculated the deadline miss ratio (DMR) for each individual task. For data collection, we stored the dispatch time and the finish time of every job in locked memory to avoid memory paging overhead.

Results. After executing this workload for 10 one-minute runs, we used the DMR metric to verify that no deadline was missed. As in the first experiment we did not observed any deadline miss in both task set, thus we present the results using two different metrics in order to observe the behavior of the system. The first metric is the job's *response time* of a periodic task, which is the difference between the job's finish time minus the job's dispatch time. The second metric is the job's *release delay*, which is the difference between the job's actual dispatch time minus its "theoretical" release time.

We present the average-case response time and release delay results in Figure 4.12(a), Figure 4.12(b) for VM^1, and Figure 4.12(c), Figure 4.12(d) for VM^2.



(a) Response Time

(b) Release delay

(c) Response Time

(d) Release delay

Figure 4.12: Two tasks set of the the synthetic real-time application executed on two separate virtual machines that were scheduled by SCHED_DEADLINE.

Note that observing the response time alone is not sufficient to see that the task did not miss its deadline. It is also necessary to observe the tasks' release delay to see at what time the jobs were actually dispatched. Because if a job is dispatched too late it could miss its deadline even if its response time corresponds to its execution time.

Comparing the trend of the response time and the release delay with the deadline of all tasks demonstrate that no deadline miss occurred. This proves the efficiency of the method to guarantee the respect of the tasks' deadlines even in the presence of virtualization overhead.

We can also see how the observed response time of a task T_i differs from its worst-case execution time e_i. For example the *wcet* e_1 of task T_1 is $300ms$, while its response time is $500ms$ (see Figure 4.12(a)). This is because when VM^1 is scheduled executes task T_1 for $160ms$ then it is preempted by VM^2. Then it is re-re-activated and it finishes executing task T_1. As indicated in Figure 4.13, at t = 60 we can see that task T_1 is released it waits until the activation of VM^1, then it is executed for $160ms$. After that it is suspended for $160ms$ because VM^1 is preempted by VM^2, and finally it continues executing to finish at time t = 65.

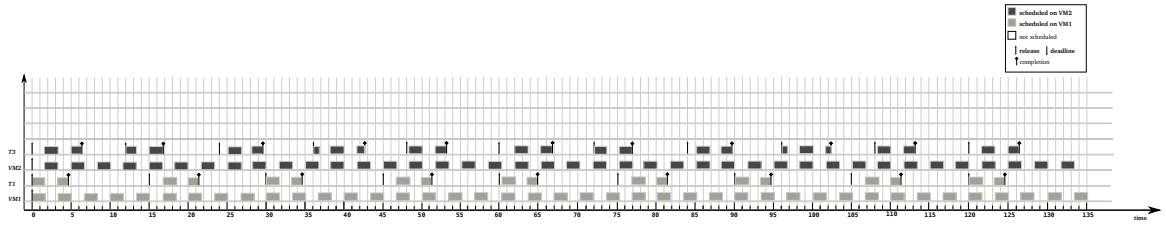


Figure 4.13: Scheduling of virtual machines VM^1 and VM^2 using the same priority. Here the scheduling graph shows the execution of both virtual machines and their highest priority tasks (T_1 and T_3) during the hyperperiod, which is the smallest interval of time after which the periodic pattern of all tasks is repeated and calculated using the least common multiple of all tasks' period (in this case it is equal to $12000ms$).

Moreover, we can observe how the release time of a task influences its response time, for example from Figure 4.12(c) and Figure 4.12(d) we can see that when the jobs of task T_3 and T_4 are released at the same time (case number 1, 2, 5, ...) the response time of task T_4 is higher than the case where the jobs are not released at the same time (case number 3, 4, 6, and 8). This is explained by the fact that when both tasks are released at the same time, task T_4 is delayed by the execution of task T_3 and the virtual machine VM^2 shares the CPU with virtual machine VM^1.

4.7 Summary

In this chapter, we examined how the scheduling of virtual machines could affect the real-time performance of a guest RTOS and its applications. We argued for the adoption of the *periodic resource model* interface to specify for each virtual machine a budget and a period. This interface

guarantees that the CPU resource will be provided by the host when it is required by a virtual machine and reinforces the predictability of the system. In other words, the very high worst-case overheads and latencies observed in our first evaluation of a virtual machine system had no effect on the predictability of the real-time application executed by a guest RTOS if the virtual machine monitor uses the periodic resources model to schedule the virtual machines.

We analyzed an analytical method that allows to compute the PRM interface for each virtual machine in order to meet the timing requirements of all real-time periodic tasks running on the guest OS. We also extended this method to integrate the overhead incurred by the virtualization system.

We experimented two different implementations that use the periodic resource model interface to allocate the CPU resource for each virtual machine, we showed how temporal isolation was guaranteed in both cases.

In our experiment we used a user-level library to evaluate in practice the theoretical technique regarding the periodic scheduling of virtual machines. The Vsched user-level library allowed us to reduce the development time because we employed it at the early stage of our research experiment that was started before the integration of the SCHED_DEADLINE scheduling class into the mainline Linux. The results of the evaluation revealed that such an approach offers a good tradeoff between flexibility and performance.

This raises the question of adopting a user-level approach to configure an RTOS? A question that we explore in the next two chapters.

CHAPTER 5

RTOS Models Transformation and Configuration

In this chapter we present our work aiming at transforming a simulation RTOS model into an executable RTOS model and enabling its configuration. In Section 5.1 we review the classical system-level design flow for System-On-Chip. In Section 5.2 we present the OveRSoC methodology, and discuss a solution to transform a simulation model into a model that is executable on a real hardware. In Section 5.3 we review the model-driven engineering approach, show how to use this method to create a model-to-model transformation and finally we discuss the limitation of the approach and its improvement.

5.1 Software/Hardware Co-design Process

Usually, when developing application-specific System-On-Chip, system designers start by writing the software models of the system in a high-level programming language such as C in order to validate the specifications. On the other hand, they implement the hardware part using hardware description language such as VHDL, or Verilog, then they verify their design using a simulation tool before using a high-level synthesis tool to transform the description into a configuration of a programmable hardware circuit (*e.g.* FPGAs, *field programmable gate array*).

Testing the software and the hardware together allows to verify the specification, and if the results are conform to the specification, then the hardware design is used to create the ASIC (*application specific integrated circuit*) System-On-Chip.

In order to accelerate this design flow (see Figure 5.1), and to explore the design space which allows to decide early what functionality should be implemented in software or in hardware, system designers may use system-level simulation library such as SystemC. This C++ library allows to create accurate models of the system. Since the software and the hardware are both defined at the

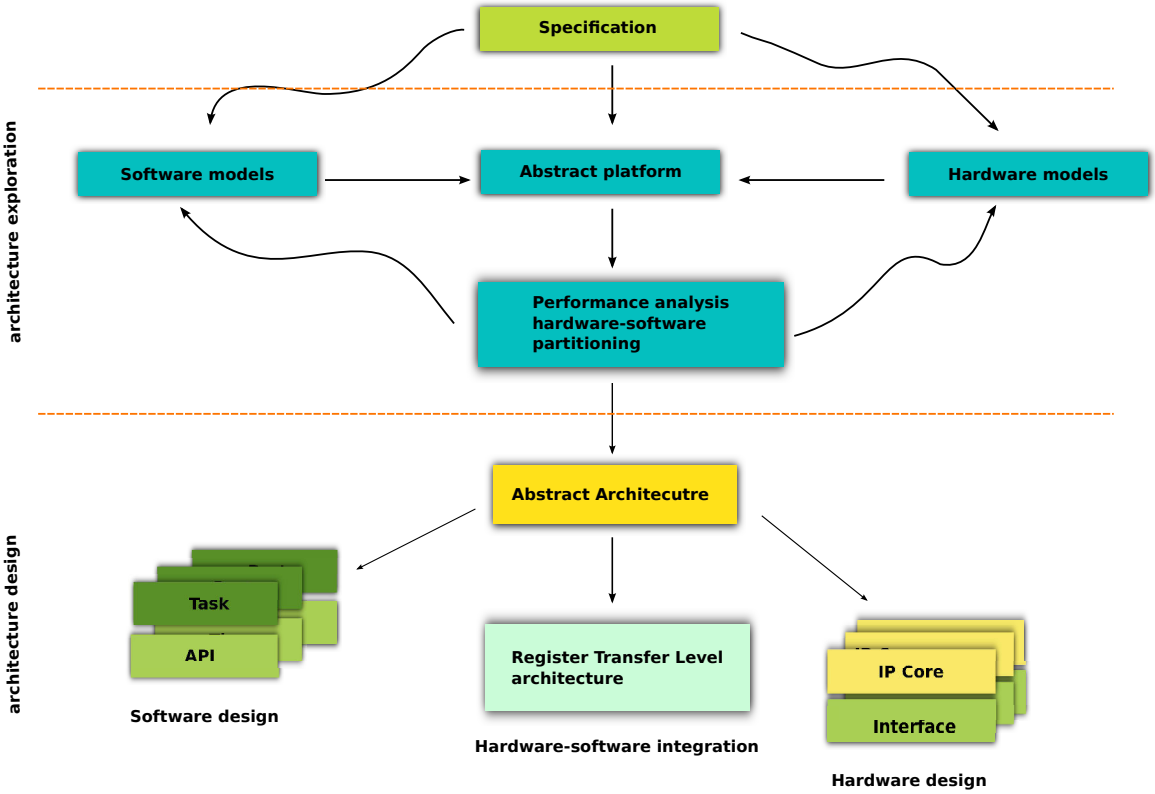


Figure 5.1: System-level design flow for SoCs.

same level, it is possible to test and simulate the whole system using the software and the hardware models.

For example, the ARM Fast Models (ARM, 2014) are SystemC models of the CPU created by ARM, and used to validate the latest developed ARM instruction set architecture before licencing the IP (*intellectual property*) of the hardware logic to chip founder such as Freescale, Texas Instruments, Samsung, Qualcomm, *etc.* A second example is the SoCLib library (Lip6, 2014) which is an open platform for virtual prototyping of multiprocessor System-on-Chip.

A system developer may then use SystemC models as a virtual platform to accelerate the development phase and test the complete software and hardware system months before the availability of the hardware prototypes.

5.2 OveRSoC Methodology

The OveRSoC methodology (Miramond et al., 2009) aims to build a platform that permits the software-hardware co-design of real-time operating systems for embedded reconfigurable system-

on-chip platforms. Such a heterogeneous platform combines a multitude of hardware units, for instance general processing units (*e.g.* CPU), reconfigurable hardware units (*e.g.* FPGA), and even computation specific hardware units (*e.g.* DSP, GPU). Furthermore, the OveRSoC methodology emphasizes that the use of *dynamically* reconfigurable hardware units permits to adapt the architecture to various incoming tasks at runtime.

This heterogeneity complicates the design process because the system designer have to decide in one hand, how the application should be partitioned onto the processing cores, and on the other hand, how the dynamically reconfigurable hardware resources should be managed, what services should be provided to the programmer, where they should be implemented, *etc.*

With regards to these questions, the use of an RTOS is more commonly adopted by the different approaches. In one approach, an existing RTOS (*e.g.*. VxWorks, μcOS-II, QNX Neutrino, *etc.*) is modified to integrate the new services that permit the management of all the hardware architecture. In a second approach, an RTOS is created from scratch, and integrates the support of all the required services.

The OveRSoC methodology adopted an approach in which high-level SystemC models of the RTOS and the reconfigurable system-on-chip (RSoC) hardware are developed together in order to explore the efficiency of different critical design choices. Once the candidate design solution is validated, then it is refined towards low-level abstraction down to real hardware implementation.

The OveRSoC framework proposes a set of RTOS's services required to explore the management of the reconfigurable hardware unit as well as the standard OS services such as tasks management, scheduling, and synchronization.

Through the use of an API implemented by these services a programmer may then create an application composed by a set of software and hardware tasks. The functional behavior of each task should be written in pure C code independently of its nature software or hardware. The system designer may then use the models provided by the OveRSoC framework to compose the complete RSoC platform. Figure 5.2 illustrates an example of a graphical RSoC platform model created using the Dogme tool (Aichouch et al., 2008).

Figure 5.3(b) shows all the available components proposed by the framework. Some components are stand-alone, that is, they could be used directly to provide a set of services, *e.g.* the "basic_PE" represents a processor. In contrast, the other components are *behavioral* components, that

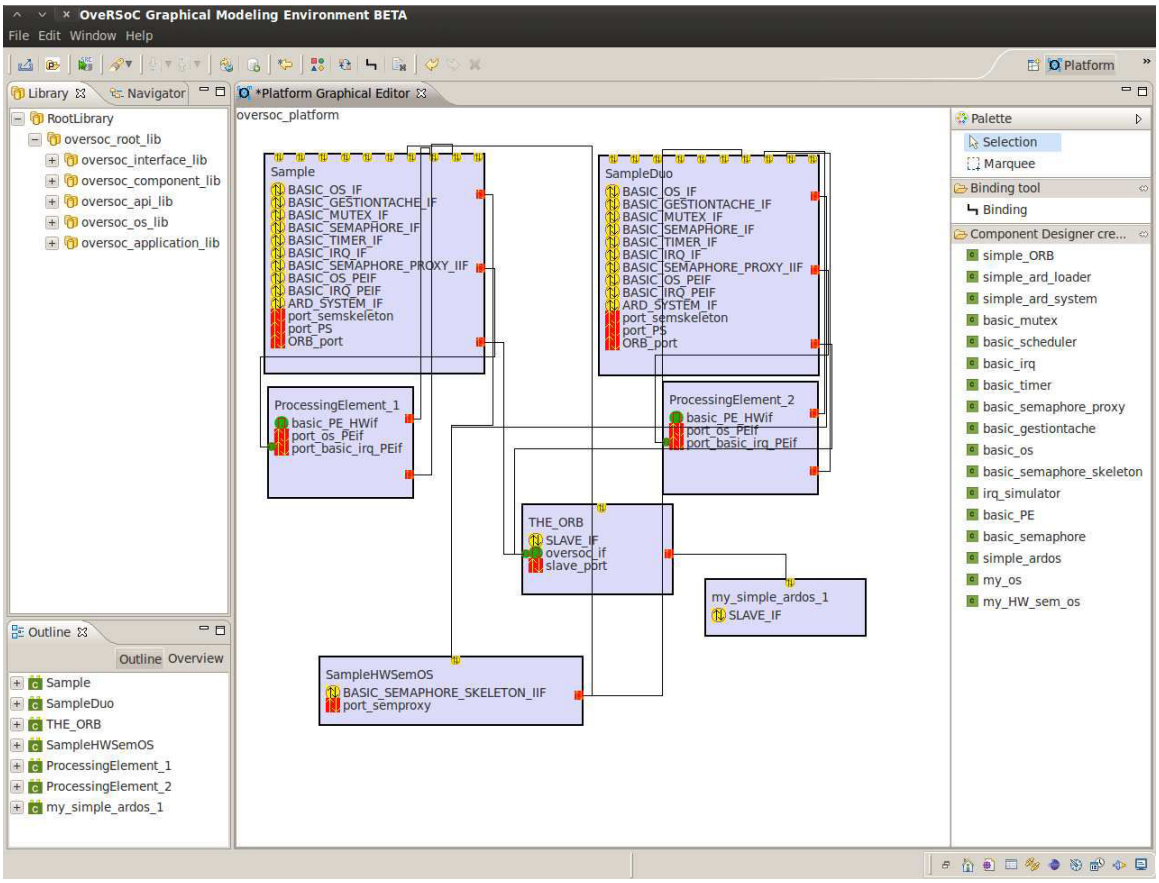
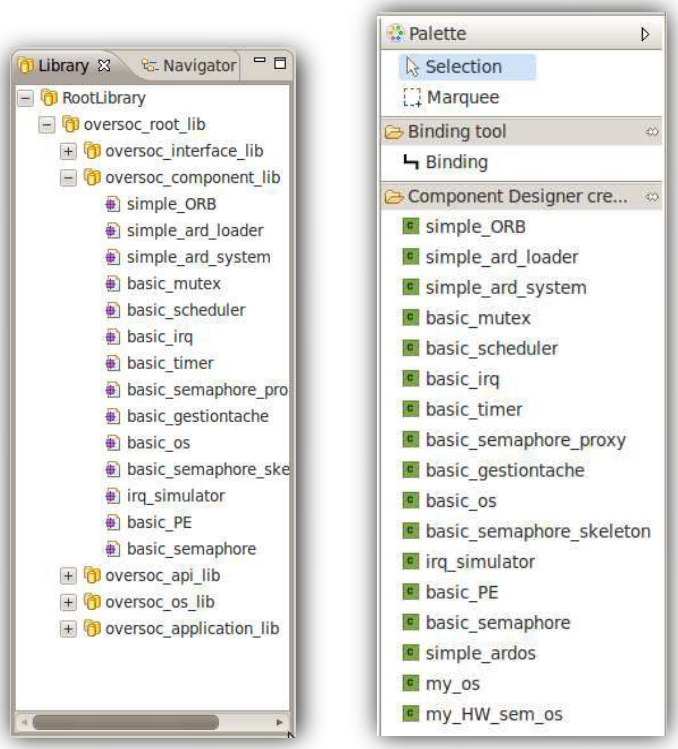


Figure 5.2: OveRSoC development tool. In the hardware model, two Processing Elements (PEs) and one reconfigurable hardware unit (named "my_simple_ardos_1" in the figure) are instantiated. These hardware components are needed to execute the software and hardware tasks of the application. Also one communication element ("THE_ORB" in the figure) and one synchronization mechanism ("SampleSemHWOS" in the figure) represent a communication medias and a locking mechanism. Alongside the hardware part, the system designer deploys two operating systems (named "Sample" and "SampleDuo" in the figure) on the two distinct processors (PEs).



(a) OveRSoC's Library. (b) OveRSoC's design toolbox.

Figure 5.3: OveRSoC's Library and design Tool.

is, they are intend to build a *structural* component, *e.g.* the ”basic_scheduler” is a sub-component of the ”OS component”.

Figure 5.4 shows the internal view of the OS component. A structural component does not implement any functional behavior, all the services that it provides are implemented by the sub-component that it contains. All these components are defined using SystemC models stored in a library, that is shown partially in a tree view in Figure 5.3(a). This library could be enriched by new developed models simply by adding the new SystemC definition and the XML description of the model to the library.

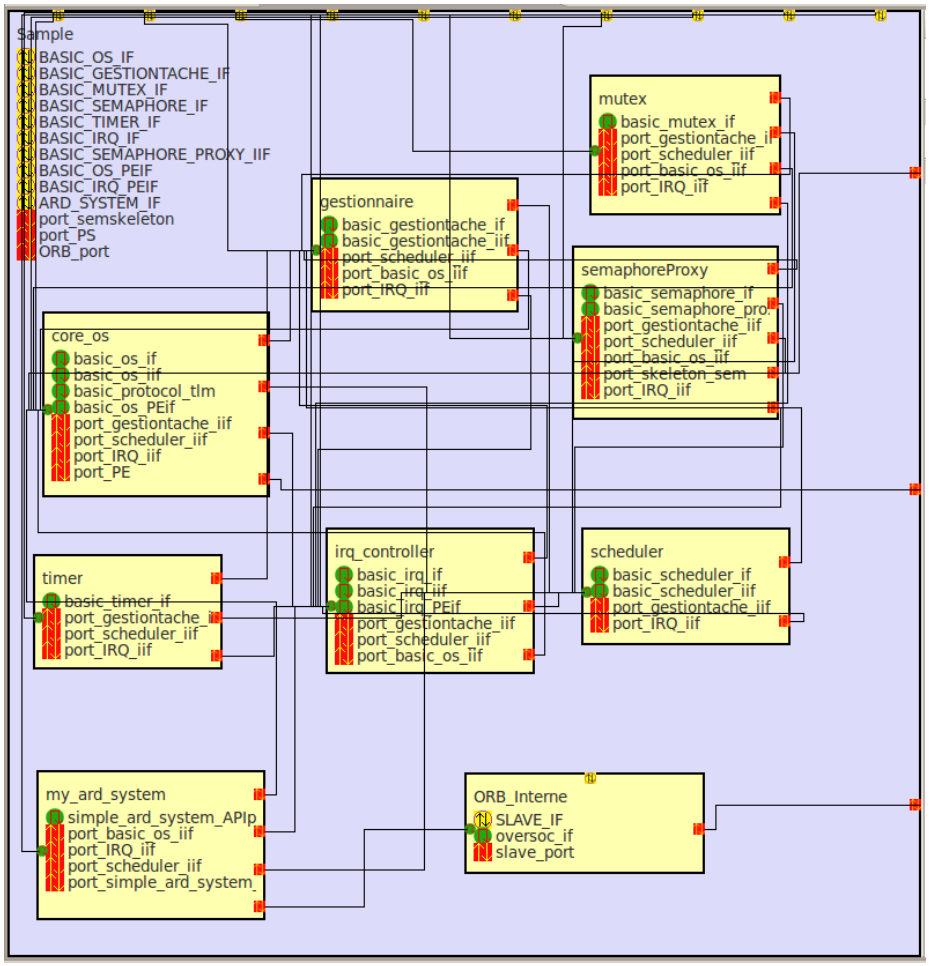


Figure 5.4: OveRSoC graphical component designer.

Once the platform is completely defined, the system designer may adjust some attributes regarding the available resources, specify a set of metrics to evaluate the resources utilization, necessary

to validate the design, then request an automatic generation of the structural source code of the platform.

The Dogme tool generates the main module of the platform and a module for each structural component, that is, all the components present in the platform are instantiated, as well as their inter-connection, and the sub-components contained by a structural component.

The generated SystemC source code combined with the source code of the application are then compiled into one binary code given as an input to the SystemC simulation engine. The SystemC models are then simulated at high level during a specified amount of time.

After the end of the simulation, the system designer uses the set of the recorded metrics to check the respect of the timing constraints and the correctness of the functional behavior. After the analysis of the performance, the designer may then decide whether the platform is valid or some adjustment of the attributes are needed, and iterate over the global simulation to explore the new design.

Based on the new overall performance, the designer may decide to validate the design choices and continue to the next step where the models are progressively refined.

5.2.1 From Simulation Models to Executable Models

One of the ultimate goal of the OveRSoC methodology is to produce a binary code that could be used on real hardware.

Obviously, once a platform model is validated, a part of the hardware models could be replaced by the corresponding hardware (*e.g.* CPU, DSP, Memory, *etc.*), a second part could be automatically synthesized into hardware circuit configuration, and a third part that could not be synthesized needs to be refined into a hardware description that could be synthesized.

Then, simulation models that are intended to be implemented in software have to be transformed into a source code that could be compiled and executed on a real hardware.

So, the transformation mainly concerns the conversion of the RTOS simulation software models into an RTOS executable software models. Such a transformation implies the complete re-writing of the software in order to make it executable on real hardware. Unfortunately, this operation is a difficult and time-consuming engineering effort.

To simplify this operation, we proposed a method based on a model-driven engineering technique. Given that the software simulation models are composed by structural and behavioral components, thus it is possible to use a model-to-model transformation technique in order to automatically create an executable model representing the structural part of the simulation model.

Thereafter, using the information extracted from the structural executable model, and an existing source code that could be executed on a real hardware, it is possible to automatically generate programs that are executable on real hardware.

In the next section we review the necessary background to understand the model-driven engineering approach, and present how to use a model-to-model transformation technique to convert simulation models into executable models.

5.3 Model Driven Engineering

The Model-Driven Engineering (MDE) is an approach for software development, which is based on models as a first artifact in the development process. Then, a transformation is applied on these models to map the information from one model to another or to generate executable programs.

A model is an abstraction, a sufficient simplification to understand the real system. In this context, a system may be defined using different sub-models connected to each others.

The definition of a modeling language, called meta-modeling, is the key issue of the model-driven engineering. A modeling language defines the rules and constraints that are required to build a specific model. Once a model is completely defined, it is often necessary to apply models' transformation in order to generate custom code, documentation, test, validation, verification and binary code (see Figure 5.5).

After the adoption of the object oriented approach by the software industry, the model-driven approach may be seen as the continuity of the initial approach. While the object oriented approach is founded on the notion of "an object that inherits from" and "an object is an instance of one particular class", in MDE the main concept is a *model*. The standards consortium, Object Management Group (OMG, 2014), defines a model by:

Definition 5.1. A model is an abstraction of a system, modeled upon a set of facts which was built for particular intend. A model should be used to answer the question about the modeled system.

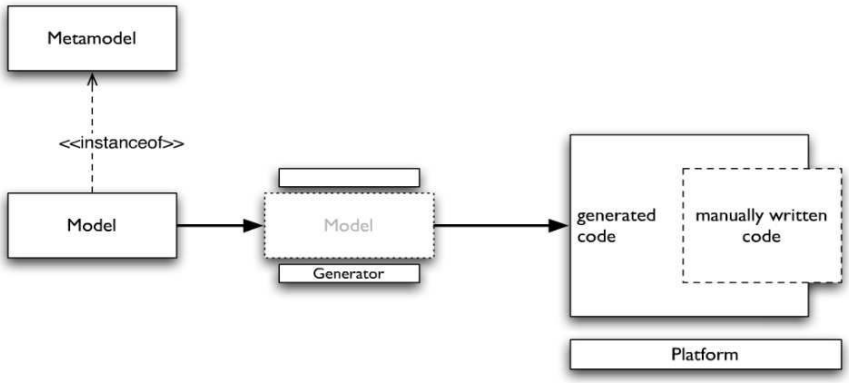


Figure 5.5: Model-Driven software development process.

In the MDE approach, the notion of model refers explicitly to the notion of well-formed language. More specifically, an operational model is a model that can be manipulated by a computer. This well-formed language should be clearly defined, and the definition of a modeling language has been formalized using particular models, called *meta-model*:

Definition 5.2. A meta-model is a specific model defining a language to describe other models.

The Object Management Group uses these two notions to define the set of the *Unified Modeling Language* standards. UML is the most widely used standard for describing systems in terms of object concepts. UML is very popular in the specification and design of software, most often to be written using an object-oriented language. UML emphasizes the idea that complex systems are best described through a number of different views, as no single view can capture all aspects of such system completely. Moreover, it includes several different types of model diagrams to capture usage scenarios, class structures, behaviors, and implementations.

5.3.1 Model Driven Architecture

The adoption of UML has been a major point in the transition towards model-driven engineering. After the acceptance of the key concept of meta-model, many meta-models have emerged. In order to avoid the multiplicity of these meta-models within a domain and to circumvent the incompatibility between them, the OMG proposed a standard language to define meta-models. This language constitutes a model itself and is referred to as *meta-meta-model* named MetaObject Facility (MOF):

Definition 5.3. The meta-meta-model MOF is a model that defines a modeling language, that is, the necessary modeling element to define a modeling language. And it should have the ability to define itself.

Using these definitions of the different abstraction levels, the OMG has organized these notions of modeling hierarchically. The ”real world” is represented at the lower level (M3), the models representing this reality are based at level (M2), the meta-model used to define these models are at level (M1), and finally, the meta-meta-model, unique and self-defined, is represented at the top level (M0).

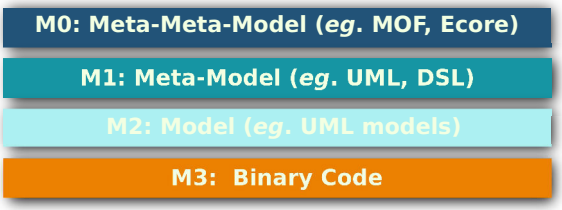


Figure 5.6: Hierarchical Modeling Levels.

The Model-driven Architecture (MDA) relies on the UML standard to describe the different phases of the development project cycle. In MDA, a Computational Independent Model (CIM) is elaborated in order to specify the solution to the requirement. Then, a Platform-Independent Model (PIM) of the system is developed and the model is transformed to obtain a Platform-Specific Model (PSM). This facilitates early validation and implementation on different platforms. All these concepts inherently increase productivity, reduce software development time and provide high quality products.

5.3.2 Domain Specific Language

In the MDA approach, it may be noticed that the model-driven engineering is tightly associated to UML. However, an important point here is to separate the MDA approach from the UML formalism. The reason is that the model-driven engineering scope is wider than UML. Sometimes UML must be reduced or extended through mechanisms like *profiles* (*e.g.* UML-Marte, SysML, *etc.*). The model-driven approach encourages the creation of domain-specific language that the user can handle easily.

Definition 5.4. A Domain-Specific Language (DSL) is a language designed to be useful for a specific set of tasks, as opposed to a general purpose language.

With DSL, software designers are currently able to create models very rapidly and efficiently. They are also capable of generating executable code from the defined models in a very simple manner.

Summary. In this section, we reviewed the basic concepts of the model-driven engineering approach. In the next section, we present the RTOS meta-model, and define the domain-specific language used to create RTOS models. Finally we show how it is used to transform RTOS simulation models into executable models.

5.4 RTOS-specific Modeling Language

A modeling language like any other language has two main properties, a *semantic* and a *concrete syntax*. The semantic of a modeling language is defined by a meta-model, and specifies the meaning of each ”word” in the model. The concrete syntax is the way how a model is created or written, it could be in *textual* or *graphical* form.

5.4.1 RTOS Meta-Model

The meta-model forms the set of concepts, rules, constraints, constructions, and all the elements that define the semantic of the modeling language. To define these elements in our RTOS meta-model we used the meta-meta-modeling language, MetaObject Facility. The MOF standard is implemented by frameworks such as the Eclipse Modeling Framework (Steinberg et al., 2008; Gronback, 2009), referred to as *Ecore* language, and the Microsoft Visualization and Modeling SDK (Cook et al., 2007).

MOF is defined by a set of basic concepts inspired from the object oriented approach. Figure 5.7 shows how these concepts are constructed. The concept of *Class* is used to represent real world objects. A Class is characterized by properties called *references*, if their type is a complex type, referred to as a ”TypedElement”, and *attributes* if their type is primitive type, called a ”DataType” (*e.g.* Boolean, String, Int, *etc.*).

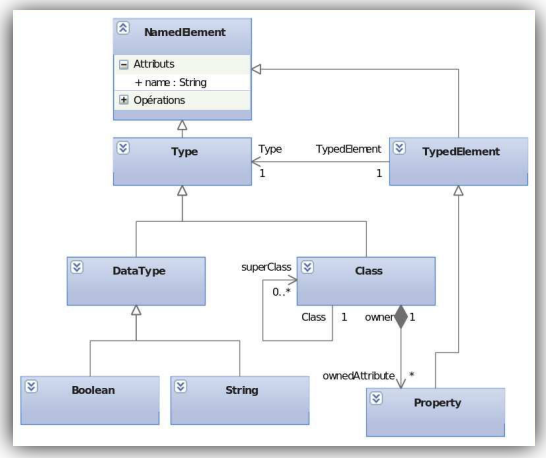


Figure 5.7: Meta Object Facility language.

To create a meta-model that define the structure of an RTOS, we were inspired by the construction of an RTOS in its most abstract form. That is, if we observe an RTOS from an abstract perspective, we may see that it is composed by a set of services, and each of these services provides a set of operations.

So, an RTOS meta-model could be defined by three main classes: the ”RTOSModel”, ”Service” and ”Operation”, and two containment relationships between them. The ”RTOSModel” class has a list of ”Service” objects, and the ”Service” class contains a list of ”Operation” objects as depicted in Figure 5.8. These three entities and their relationships are sufficient to define the structure of an RTOS in its abstract form.

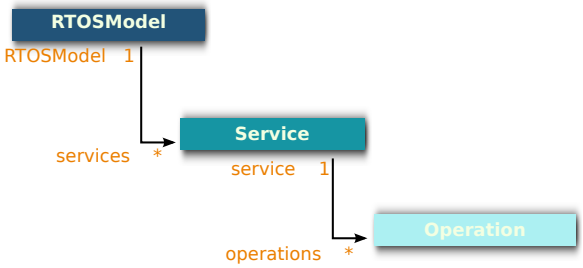


Figure 5.8: A meta-model reflecting an abstract RTOS structure.

To create our RTOS meta-model we used the Microsoft Visualization and Modeling framework, the reason for this is simply due to the familiarity with the underlying programming language used

to define models transformation. For more implementation details we refer the interested reader to the project public source code repository[1].

5.4.2 Concrete Syntax

Using the concrete syntax, the user can create a model of the RTOS structure. We defined the concrete syntax using a graphical notation, then we created a mapping between this notation and the RTOS meta-model elements. We decided that the ”RTOSModel” class should appear as the diagram containing the ”Service” models. And we associated a rectangular shape with compartment to the ”Service” class in order to display its ”Operation” list. Figure 5.9 illustrates a model of the μcOS-II RTOS created graphically using the associated modeling tool.

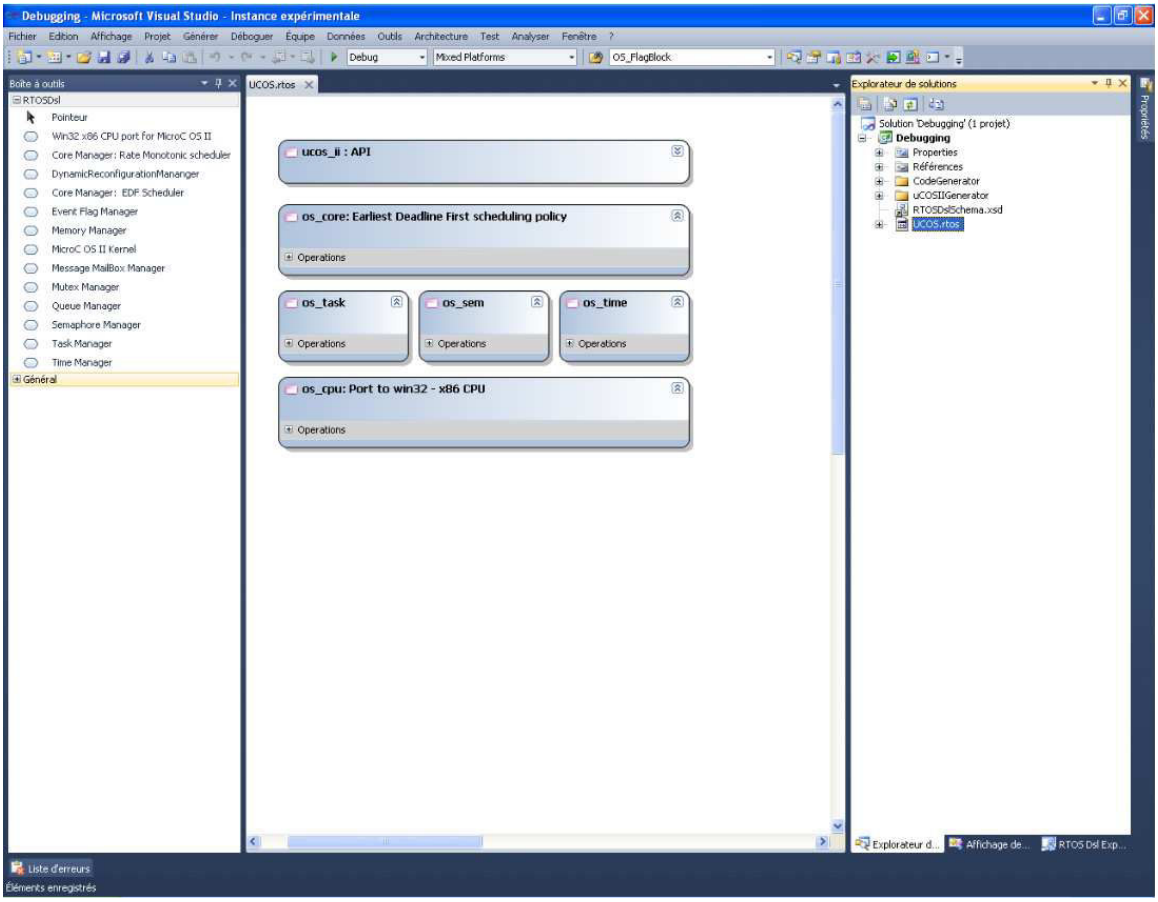


Figure 5.9: RTOS-specific modeling language tool.

[1]See `http://code.google.com/p/rtos-dsl/`.

5.4.3 Model-to-Code Transformation

Being able to represent the structure of an RTOS, the next step is to produce the final source code. Our assumption was that the behavioral of each service present in the RTOS simulation model is already defined by its corresponding service from an existing RTOS that is executable on real hardware. Thus, the idea is to parse the RTOS model representing the structure, and to generate the source code for each service present in the RTOS structural model using a set of existing source code *templates*.

In order to generate the source code of the RTOS model, we use a transformation based on source code *template* . The key to this technique is that some elements in the source code that are outside of special control markers (#+ and #) are provided directly to the output source file, whereas elements of code within these markers are evaluated and used to add structure and dynamic behavior. Listing 5.1 describes a simple example of a source code template.

Listing 5.1: Example of template source code

```
<#+
/* if GEN_FLAG_EN is false then the flag is disabled in the
generated code */
if (GEN_FLAG_EN == false) {
#>
#define OS_FLAG_EN 0u
<#+
}
/* otherwise the flag is enabled in the generated code */
else {
#>
#define OS_FLAG_EN 1u
<#+
}
#>
```

The example shows a small if and else branch test depending on the value of the parameter GEN_FLAG_EN which is true if the RTOS structural model contains an ”Event Flag Manager” service. Note that in the above example, the only code that is executed is the code surrounded by the control markers (#+ and #). The *code generator* reads an RTOS structural model as an input, iterates over each ”Service” to generate the output source code of the ”Service” depending on a source code template and the parameters that are defined by the developer.

In our implementation, we created a template based on the μcOS-II source code. μcOS-II is a preemptive, real-time multi-tasking kernel for microprocessors and micro-controllers. It is implemented in ANSI C and certified by the Federal Aviation Administration for use in software intended to be deployed in avionics equipment. It has been massively used in many embedded and safety critical systems products worldwide. The main services provided by μcOS-II are depicted in Figure 5.10.

μcOS-II is implemented as a monolithic kernel, *i.e.*, it is built from a number of functions that share common global variables and data types such as *task control block*, *event control block*, *etc.* It is a highly configurable kernel, whose configuration relies on more than 70 parameters. Since the kernel is provided with its source files, configuration is performed via conditional compilation at pre-compilation time, based on #define constants.

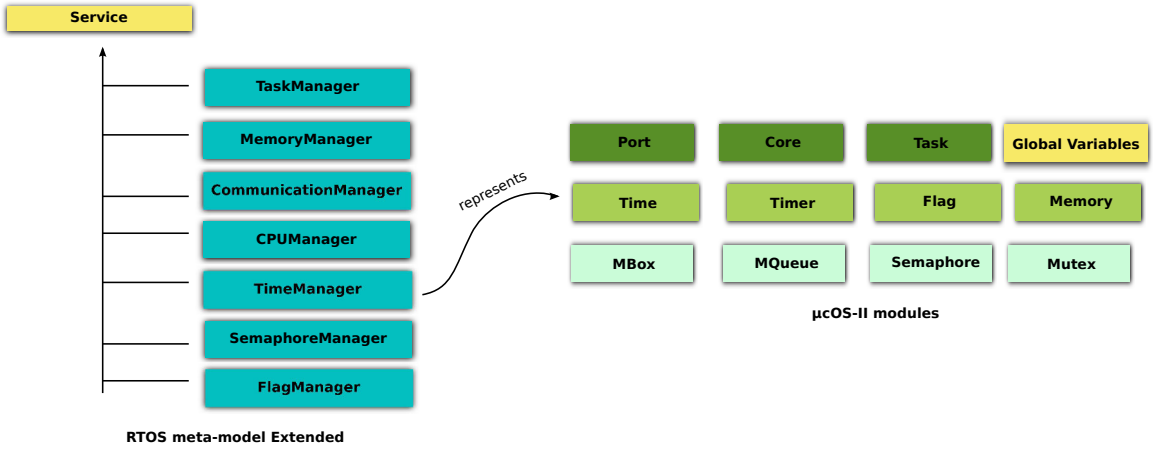


Figure 5.10: Extended RTOS meta-model.

The selection of μcOS-II is based on two main reasons. First, the modularity of μcOS-II allows the designer to add or remove services depending on the RTOS model. Second, the source code of

μcOS-II has been ported onto multiple embedded platforms (DSPs, micro-controllers, soft cores in FPGAs, *etc.*).

However, in order for the code generator to produce the source code from an RTOS model using a set of templates built from the existing μcOS-II source code, the RTOS meta-model needs to be extended. Because the μcOS-II RTOS has different modules, and each module has a set of source files, thus each module needs to be explicitly represented by a meta-model entity which includes information about the source files wherein the module is defined.

For instance, the Task module is represented by the "Task Manager" class. Each "ServiceManager" class inherits from the abstract "Service" class. Figure 5.10 shows the inheritance relationship and the mapping between the RTOS meta-model entities and the μcOS-II modules.

5.4.4 Test of the Transformation

We have tested our prototype to generate a specific μcOS-II version compliant with the x86 architectures. First, we created a minimal RTOS model containing the following services: Task, Time, and Core management with the rate-monotonic scheduling policy (see Figure 5.9). Then, we have transformed the model into source code.

A typical real-time application has been written according to the proposed API and a first test has been led with a RM scheduling policy.

After executing the code, performances in terms of execution time and deadlines respect have been analyzed and compared to the classical μcOS-II kernel.

Note that, if the results do not meet specific constraints that are required by the application (for example, deadline constraints), it is very simple to generate another version of the OS with other services' attributes until a satisfactory solution is reached. In a second example, we have tested the same application but with a different scheduling policy. An EDF scheduler has been used and a new simulation has been performed.

5.4.5 Model-To-Model Transformation

After creating a model representing the structure of an RTOS and generating the final source code that is executable on a real hardware, we need now to find a mechanism to transform an OveR-

SoC simulation model into a structural RTOS model. This technique is called a model-to-model transformation in the model-driven engineering discipline.

This technique relies on the fact that any meta-model is mandatory defined using the MetaObject Facility language. So, it is possible to create a mapping between each element from a meta-model A and each element from a meta-model B as long as both meta-models are defined using the MOF standard.

Concretely, we create a mapping between the ”Component” entity present in the OverSoC meta-model and the ”Service” entity present in the structural RTOS meta-model. After that, this rule is used by a *transformation engine* to convert a model instance of the OveRSoC meta-model into a model instance of the structural RTOS meta-model, as illustrated in Figure 5.11.

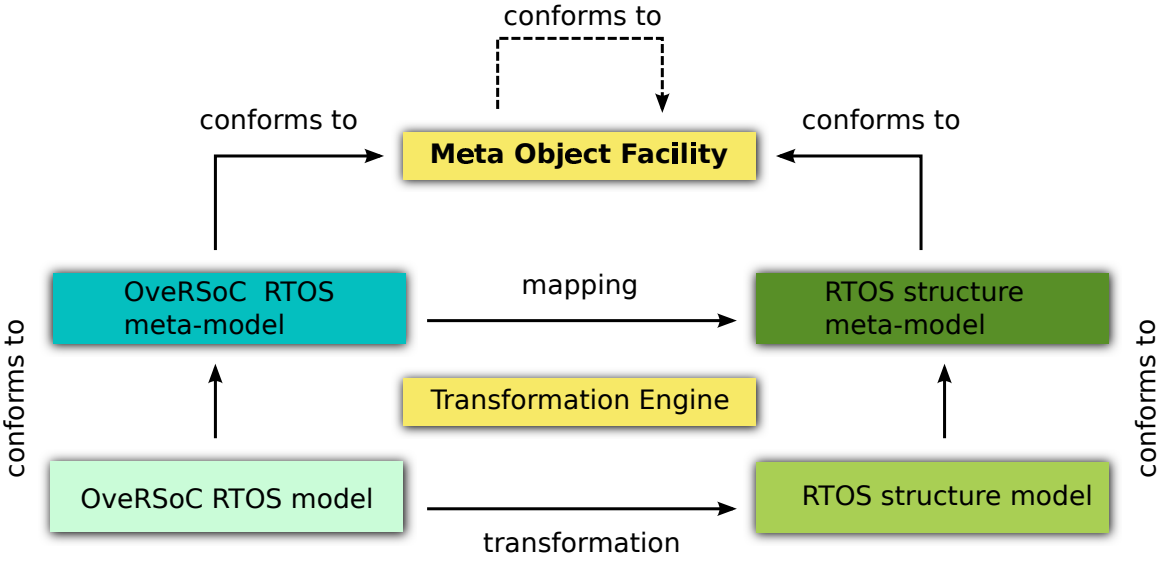


Figure 5.11: Model-to-Model transformation process.

This process could be automatically applied on any simulation model instance of the OveRSoC meta-model. Giving this model as an input to the *transformation engine*, produce as an output a model instance of the structural RTOS meta-model. After that, it is possible to use the developed *code generator* to automatically produce the final executable programs.

5.4.6 Limitation of the Approach

One major limitation of the presented approach is related to the modification of the source code of the existing RTOS (in our case μcOS-II). This modification is necessary because the final source

code should be able to provide any property that was configured in the OveRSoC RTOS simulation model. For instance, if the designer selects a rate-monotonic scheduling policy in the RTOS simulation model, then the final executable source code should include the same policy, and if the designer selects an EDF policy then this policy should as well be supported by the final RTOS source code.

Changing the scheduling policy is a straightforward operation in the RTOS simulation model because the RTOS simulation model relies on the component-based design, where each component is independent from the others and communicates through a set of ports and interfaces, thus changing the internal implementation of one component does not break the whole system as long as the the interfaces are preserved. In contrast, this operation could be very complex in the case of a monolithic RTOS, first due to implementation problems such as the function calls dependency and data structures dependency between the modules, and second due to the fundamental problems that we mentioned in our reviewing of the state-of-the-art related to RTOS configuration through kernel level modification (see Section 2.4).

As a solution we proposed a method that avoids the modification of the RTOS internal kernel when it came to implement a new scheduling policy or synchronization protocol. The idea is to implement the RTOS features that are subject to configuration at middleware level without changing anything in the RTOS kernel.

5.5 Summary

We have revisited the OveRSoC methodology and proposed a method to transform OveRSoC simulation models into executable models on a real hardware. The method decomposes the OveRSoC simulation models into two separate parts: structural and behavioral. We reviewed the model-driven engineering approach and demonstrated how it could be used to easily transform the simulation models into executable models. We have further discussed the limitation of the proposed method and mentioned the potential solution, that we present in the next chapter.

CHAPTER 6

RTOS Configuration using User-Level Library

A limitation of the method presented in the previous chapter is that the configuration of the RTOS executable model required the modification of the RTOS internal kernel. As discussed in Chapter 2, this approach suffers from a set of drawbacks related to its adoption by industrial practitioner, and its maintenance and integration with the open source operating system projects. In this chapter we present a potential solution to overcome these problems.

The idea is to implement the scheduling algorithms and the synchronization protocols outside the kernel. This means that the kernel is no longer responsible for taking the decision related to selecting the next task to run, preempting the currently running tasks, or context-switching between them.

Fundamentally, the idea relies on the separation between the abstractions and the policies. That is, the RTOS kernel is responsible for providing abstractions of the hardware resources and the mechanisms to control them, in the opposite, a user-level library implements the policies to allocate and manage these resources. This design known as a *middleware* in the software engineering terminology.

Recently, Mollison and Anderson (2013) implemented a middleware that runs on top of POSIX RTOS allowing researchers and integrators to develop and evaluate new scheduling and locking techniques for multicore hardware.

The user-level library has been evaluated on top of a real-time Linux kernel configured by the PREEMPT_RT patch. The results of the evaluation showed that the overheads and latencies of the library were similar to the same measurements from a kernel-level approach. Furthermore, a robustness test was conducted in which a real-time application has been executed by the user-level

library. The experiment proved that the library was able to guarantee the respect of the application deadlines during twenty four hours of experiment.

This promising design corresponds to our requirement, first because it avoids any modification to the RTOS kernel, second it permits to build a library that assembles all the new developed resource allocation techniques and thereby facilitates their reuse and sharing across multiple projects.

However, this approach becomes more effective if it could be used on top of different RTOSes, or at least easily ported to new RTOSes and platforms. More specifically, we declared in Chapter 5 that our requirement is to use a RTOS to manage a heterogeneous hardware platform composed by CPUs, FPGAs, DSPs, *etc.* In practice such a platform is known as a *hybrid platform*, the Xilinx Zynq 7000 computing board is an example. It combines two ARM Cortex A9 CPUs and a reconfigurable hardware circuit (FPGA), as illustrated in Figure 6.1.

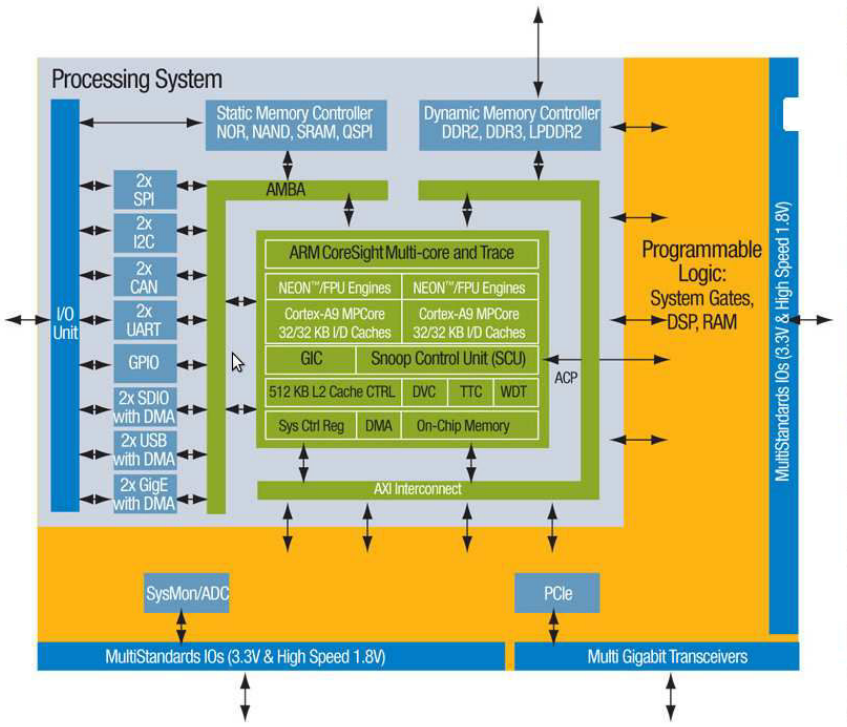


Figure 6.1: Block diagram of the Xilinx Zynq 7000 System-On-Chip.

Recently, Pham et al. (2013) proposed a solution based on a microkernel operating system to manage efficiently this hybrid platform. In the proposed solution, a microkernel has been adapted to manage the FPGA as a hardware accelerator able to execute concurrently multiple compute intense tasks in hardware. The framework schedules concurrently software tasks on the CPUs and hardware

tasks on the FPGA. The microkernel uses the *dynamic partial reconfiguration* property of the FPGA to adapt the hardware architecture at runtime to the hardware task selected for execution. This solution complies with our requirement, therefore it is necessary to investigate the portability of the user-level library not only on other commercial RTOSes but also on microkernel-based operating systems.

In this chapter, we present the adaptation of the user-level library (Mollison and Anderson, 2013) to a microkernel-based OS. In Section 6.1 we review other approaches to user-level scheduling on top of microkernels. In Section 6.2 we present the abstractions and mechanisms required by the user-level library. And we present how these abstraction are implemented by the Nova microkernel, then we describe the implementation of the user-level library on top of the Nova microkernel. Finally, we show the results of our experiments.

6.1 User-Level Scheduling on top of Microkernel

The user-level scheduling concept resembles the scheduling scheme of the hierarchical scheduling framework (HSF) (Deng and Liu, 1997). Recall that in the HSF model, there are two scheduling levels; a global scheduling is implemented at the operating system level wherein a set of components share the CPU resource according to a periodic resource model (PRM), defined by a budget and a period. And a local scheduling is implemented at component level, in which a set of real-time tasks are scheduled according to the component specific policy, as we have previously illustrated in Figure 2.6. Note that the scheduling policy used at each level is not necessarily the same, for instance it might be possible to schedule the components in round-robin manner, and let each component schedule locally its set of tasks according to its specific policy, *e.g.* RM or EDF.

Multiple works have focused on analyzing hierarchically scheduled systems using monolithic RTOSes. For instance, Behnam et al. (2008) implemented HSF on VxWorks, van den Heuvel et al. (2009) supported HSF on μcOS-II, and Inam et al. (2011) deployed it on the FreeRTOS. Here, we review some studies that are based on a user-level scheduling that are dependent on the underlying microkernel.

Recently, Åsberg and Nolte (2012) presented a user-mode approach to *partitioned scheduling* in the seL^4 microkernel without requiring kernel modifications. The proposed approach relies on the

microkernel provided API to implement mechanisms such as thread suspension (seL4_TCB_Suspend()) and thread context switch (seL4_TCB_ReadRegisters(), seL4_TCB_WriteRegisters()) at user-level. Also an efficient implementation of the EDF scheduling algorithm has been proposed. The results of the experiments of this approach showed that the performance are at the same order-of-magnitude as the performance from a kernel-level approach.

Stoess (2007) proposed a user-controlled scheduling for microkernel-based systems. The approach was implemented on top of the L^{4}Ka::Pistachio microkernel and required some kernel-level adaptation. Also, it used the ExchangeRegisters( ) system call that allows a thread to read or modify parts of the execution and communication state of another thread, provided both threads are executing within the same address space. To that end, it also allows the invoker to suspend or resume other threads. Ruocco (2006) proposed a user-level fine-grained adaptive real-time scheduling via temporal reflection. It was evaluated on top of a L^4-embedded microkernel without changing its implementation.

6.2 Tasks Model and Thread Mechanisms

In this section, we first revisit shortly the real-time task model used as an abstraction to create the application, second we define the standard operating systems abstractions and mechanisms necessary to build a user-level library.

6.2.1 Sporadic Task Model

The basic entity of computational work is the task, which is a series of sequential instructions. A task T is defined by its *worst case execution time* (WCET), T_e, its *period*, T_p, and its *deadline*, T_d. The processor *utilization* required by a task T is given by $T_u = \frac{T_e}{T_p}$. Each successive job of a *sporadic* task T is released at least T_p time units after its predecessor. If each successive job is released precisely T_p time units after its predecessor the sporadic task becomes a *periodic* task.

6.2.2 Thread library

A thread library provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library

entirely at user-level with no kernel support. Then, all code and data structures of the library exist at user-level. This means that invoking a function in the library results in a local function call at user-level and not a system call. Many such libraries are available, such as the GNU Portable Threads, ”Pthreads”, refers to the ”POSIX IEEE standard 1003.1” (IEEE, 2014) defining an API for thread creation and synchronization. A POSIX-compliant threading implementation can make use of either user-level threads or kernel-level threads:

- **User threads** are supported above the kernel and are managed without kernel intervention.
- **kernel threads** are supported and managed directly by the operating system.

Ultimately, a relationship must exist between user threads and kernel threads. The user-level library used in this paper make use of the many-to-many model to establish a relationship.

The many-to-many model, shown in Figure 6.2, multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or particular machine.

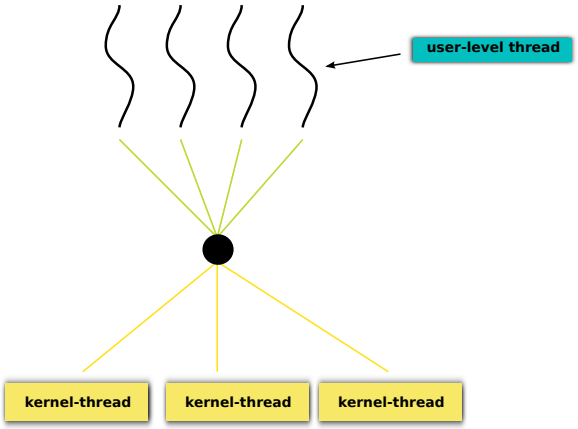


Figure 6.2: Many-to-Many model.

Beside the thread concept, two essential building blocks are required by the user-level library:

User-Context. The processor user-context is a data structure that contains the thread’s machine registers, the current execution stack, and in some operating systems the signal mask (see Figure 6.3). It allows threads to be preempted and switched among in an arbitrary order. The user-level library that we present here relies on the POSIX API ”ucontext_t ” (POSIX, 2014) to implement the user-level threads. In our adaption of the user-level library to the Nova microkernel, we ported the GNU C Library (glibc)’s ucontext_t implementation to the Nova microkernel.

Signal Handling. A signal is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending on the source and the reason of the event being signaled. Every signal has a default signal handler that is ran by the kernel. This default action can be overridden by a user-defined signal handler. In the case of the Nova microkernel, the signal mechanisms are yet not supported. Instead, we used the Inter-Process Communication (IPC) mechanisms as a substitution for the signal handling.

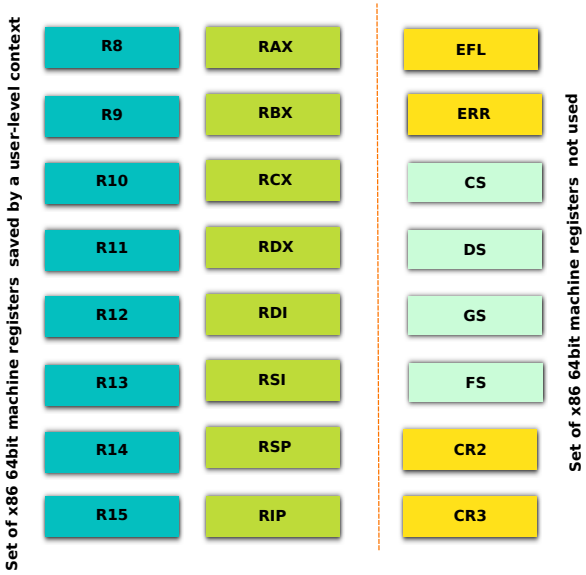


Figure 6.3: Set of registers that constitute the CPU user-context.

6.3 Nova Microkernel and Runtime Environment

Prior to presenting the implementation of the user-level library on top of the microkernel, we define the programming abstractions provided by the Nova microkernel used as ”bricks” to build the library.

Protection-Domain. It is the kernel object that implements the spatial isolation. Each protection-domain consists of three spaces: the *memory space* handles the page table, the *I/O space* handles the I/O permission bitmap, and the *capability space* controls access to kernel objects.

Execution-Context. A process in a protection domain is called an execution-context. An execution-context executes program code, manipulates data and uses *portals* to send messages to other execution-contexts. Each execution-context has its own CPU/FPU registers state.

Scheduling Context. In addition to the spatial isolation implemented by protection domains, the Nova microkernel enforces temporal isolation through the scheduling contexts. This entity combines a time quantum with a priority to ensure that no execution-context can monopolize the CPU for more than its allocated time share.

Portal. Communication between protection domains is controlled by portals. Each portal represents a dedicated entry point into the protection domain in which the portal was created.

Global-Thread. In the Nova microkernel an execution-context object could not be instantiated directly by a user application. Instead, a Global-Thread object defined by the Nova Runtime Environment (NRE) should be used. The Global-Thread associates an instance of an Execution-Context to an instance of Scheduling-Context. It is the entity used by a Nova application to execute code.

Local-Thread. A Local-Thread is also an execution-context but without a scheduling-context. It is used to execute code when there is a communication between threads through the portals.

6.4 Library Implementation

In this section, we describe the blocks on which we constructed the user-level library. First, we present how the Nova objects are used, then we present how mechanisms such as the preemption, task context-switch, time management and interrupt handling are supported at user-level.

Kernel-level Elements. The user-level library creates a Global-Thread instance that will act as a "virtual CPU". Note that this Global-Thread is a kernel-level thread that is managed by the Nova microkernel. In the library's terminology we refer to this instance as a *worker-thread*. To create the worker-thread, we used the GlobalThread::create( ) system call, it is equivalent to the pthread_create() in a POSIX RTOS.

The worker-thread is scheduled natively by the underlying microkernel scheduler. To ensure that the worker-thread will always be scheduled over the rest workload of the microkernel it is assigned the highest priority in the system.

The Nova microkernel schedules the Global-Threads in a round-robin fashion. Each Global-Thread has a priority and a time-slice. Thus, a Global-Thread is executed until it finishes its time-slice, the default time-slice in Nova is $10ms$. A Global-Thread could be preempted by a highest priority Global-Thread even if its time slice is not consumed. When a Global-Thread finishes its

budget it is inserted at the end of the ready-queue, and a Global-Thread that is at the same priority level is picked and dispatched on the CPU.

User-level Elements. The user-level library uses a set of user-level threads. A user-level thread is an entity that is not scheduled by the Nova microkernel. The user-level thread is the basic unit of the real-time computational work, and represents a real-time task of the application.

As mentioned earlier, the library implements the user-level thread by the mean of the POSIX ucontext_t data structure. This data structure is used to store the current execution context of a CPU to memory, and to load the previously-stored context onto CPU. The ucontext_t object is created using the POSIX getcontext( ) and makecontext( ) functions. Figure 6.4 illustrates the overall architecture of the library.

Scheduling. The scheduling algorithm of the library is defined in the schedule( ) function. It selects the highest priority real-time task from the ready queue and binds it to the worker-thread. When the microkernel schedules the worker-thread, the processor of the machine will subsequently executes one job of the real-time task. This level of indirection between the kernel and the real-time task allows to change the scheduling algorithm of the underlying microkernel without modifying its internal kernel.

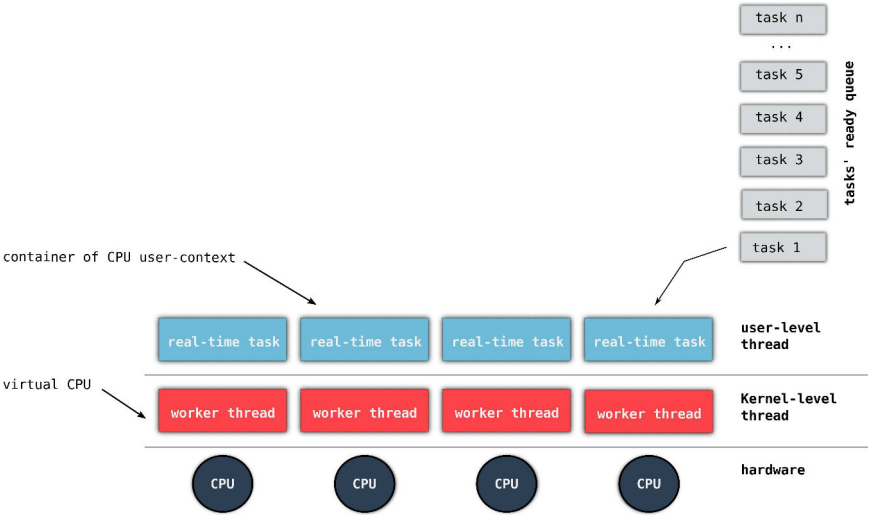


Figure 6.4: Schematic of overall architecture. The worker-thread acts as a ”virtual CPU” for the real-time task.

Preemption mechanism. The Global-Thread created by the library and used as a worker-thread is executed endlessly by the microkernel. To interrupt this worker-thread, the library uses the timer

interrupt. At system initialization, the library arms a timer to fire at the time of the earliest job release. Each time the timer fires, it is re-armed to fire again at the time of the subsequent job release.

Given that the worker-thread executes continuously the real-time workload, or just idle if there is no task ready to run, it cannot listen to the timer interrupt. Thus, the library creates a second Global-Thread instance that is used to endlessly listen to timer interrupts and to asynchronously suspends the worker-thread. We refer to this second Global-Thread as the *listener-thread*.

When a timer fires, a message is sent to the listener-thread. And when the listener-thread receives the timer signal, then it uses a Portal object to send a message to the worker-thread which will be interrupted. The transmission of this preemption signal causes the library-defined function exception-handler( ) to run asynchronously forcing the worker-thread to interrupt the currently running real-time task and jump to the scheduler( ) function.

Exception handling. Since the POSIX signal handling was not supported by the Nova microkernel, we implemented an equivalent mechanism that allows to asynchronously suspend the worker-thread from the listener-thread. To that end, we used a Local-Thread object and a Portal object in order to send an IPC message from the listener-thread to the worker-thread.

When the message is received by the worker-thread, its execution flow is suspended and redirected to the exception-handler( ) function. This function reads the machine registers state using a UtcbExcFrameRef data structure provided by Nova, which was previously filled by Nova when the exception occurred, stores it into memory, and loads the machine instruction pointer register (EIP) with the address of the scheduler( ) function and the stack pointer register (ESP) with a new address in order to redirect the execution of the worker-thread and forces it to do a rescheduling if it is necessary.

Context-Switch. In order to initialize the contexts used by the real-time tasks, the library relies on the POSIX getcontext( ) and makecontext( ) functions which create and initialize the u_context data structure. The x86 assembly function, fast_swapcontext( ), is called by the schedule() function to perform a context switch at user-level.

Time Measurement. The library relies on the x86 per-processor register known as the *timestamp counter* (TSC) to measure the time in order to make scheduling decisions. The TSC register records the number of CPU cycles that have elapsed since the processor is initialized at boot time.

6.5 Experiments

Our evaluation of the library consists in measuring a set of overheads. The measurement aims to demonstrate the performance of the user-level library on top of the microkernel. In the following subsections, we describe the overheads and latencies that are of interest, then we present the measured results. In addition, we present the hardware platform, and experimental workload.

6.5.1 Overheads and Latencies

We evaluated the library using the same set of overheads and latencies that we used to evaluate a guest RTOS running in a virtual machine presented in Chapter 3:

- **Event Latency** is the amount of time that elapses between the periodic release time of a real-time task and the corresponding invocation of the release handler.
- **Release Overhead** is the time taken to execute the release handler.
- **Scheduling overhead** is the duration of the schedule() function.
- **Context switch overhead** is the time taken to make a context switch.

6.5.2 Experiment Setup

We ported the user-level library to the x86 64-bit architecture. We evaluated the library in the Quick EMUlator (Qemu). Qemu is a machine emulator that emulates real hardware accurately down to CPU cycle level. We configured Qemu to emulate an Intel Nehalem Core i7 processor with SMP 1. The Nova microkernel was configured to run only the drivers that are required by the user-level library notably, the advanced programmable interrupt controller, the timer device, and the serial port for debugging and retrieving experiment results.

6.5.3 Experimental Workloads and Execution Trace

We tested a synthetic workload to observe how the user-level library controls the physical CPU. We analyzed the correctness of the scheduling using the Grasp visualization tool (Holenderski et al.,

2006). The Grasp tool takes as an entry the scheduling events recorded during the experiments. The real-time task set used as test case is defined in Table 6.1:

Task	Te_{i} (ms)	Tp_{i} (ms)	Tu_{i}	$Tprio_{\mathrm{i}}$
T_1	10	150	0.06	2
T_2	10	140	0.07	1
T_3	20	400	0.05	3
T_4	50	560	0.08	4

Table 6.1: Example of real-time task set schedulable under RM scheduling.

Figure 6.5 shows the execution trace under the *rate-monotonic fixed-priority* scheduling algorithm. All the tasks are simultaneously released. First, task T_2 starts executing before the others because it has the highest priority. Second, task T_1 starts executing followed by task T_3. Task T_4 is executed after T_3 but it is preempted by T_2, then it continues executing until its completion. After that, the worker-thread becomes idle because there is no ready task to run. This simple test case allows to verify the correctness of the library's behavior, that the preemption mechanism and task context-switches are performed correctly, and that the interrupts are handled correctly as well.

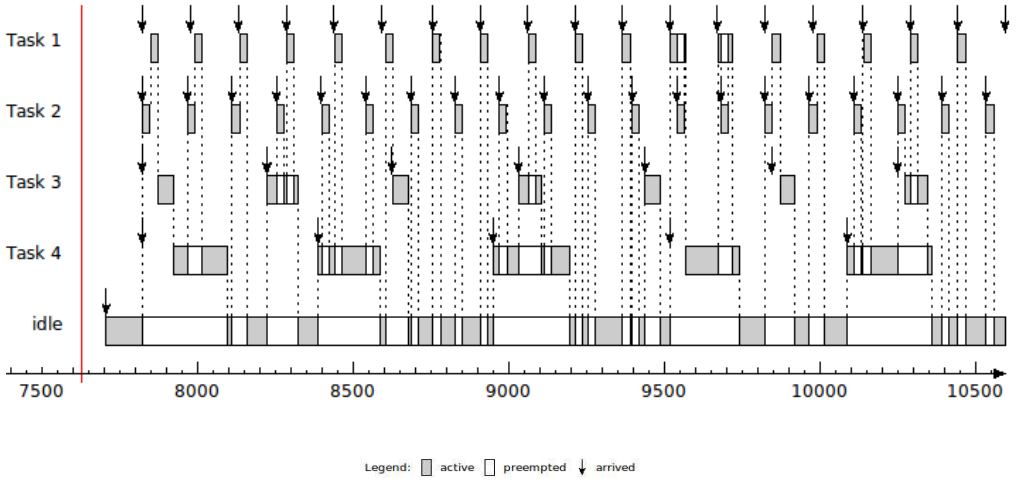


Figure 6.5: Scheduling trace of a task set according to the RM algorithm.
Execution trace of the task set presented in Table 6.1 scheduled according to the RM algorithm using the user-level library on top of Nova.

6.5.4 Measured Results

In order to measure the overheads and latencies of the library, we recorded every scheduling events that happened during the execution of a synthetic task sets system. The task sets system we used is a composition of 20 task sets, each task set has a size n, where n ranges from $n = 2$ to $n = 20$ in step of 2.

The task sets are generated by randomly choosing the utilization of each task it includes until the CPU utilization capacity is reached. The utilization of each task is randomly generated using one of the following distributions: *light uniform*, *light bimodal*, *light exponential*, *medium uniform*, and *medium bimodal*. Task periods are generated using a uniform distribution with range [$100ms$, $800ms$]. Then, the utilization and the period values are used to calculate the execution cost of each task. Each task executes the same function that performs a set of arithmetic operations on a large array in memory during 30 seconds.

After the termination of the execution, we applied a statistic analysis to extract the average- and the worst-case overheads. We presented the measured overheads as a function of number of tasks in Figure 6.8, Figure 6.6, and Figure 6.7.

Figure 6.6 shows the measured context-switch triggered by the library to switch between two user-level threads after a scheduling decision has been made. In the average case the context-switch is constant relative to the number of tasks.

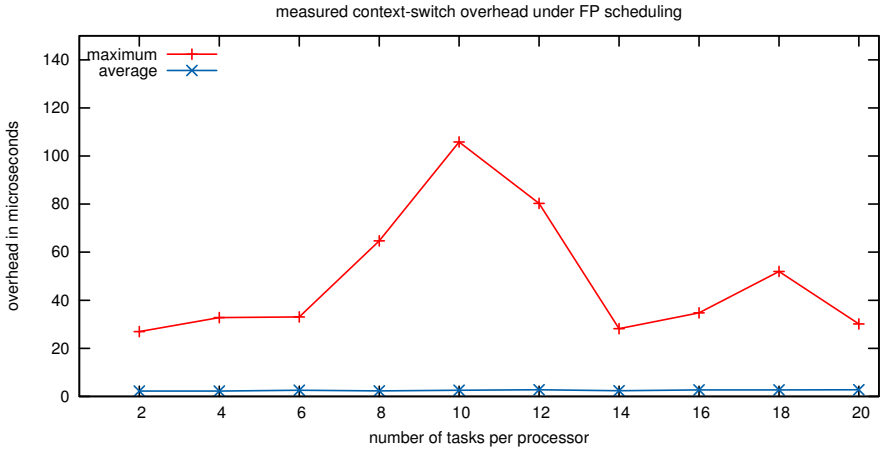


Figure 6.6: Context-Switch overhead of the FP scheduler.

Figure 6.7 shows the measured scheduling overhead under the *fixed-priority* (FP) algorithm, and Figure 6.8 shows the scheduling overhead under the *earliest-deadline-first* (EDF) algorithm. We compared the FP and the EDF scheduler because in our first prototype we implemented the FP scheduler using a sorted linked list, where the dequeuing of the highest priority job from the ready queue requires $O(n)$ time. And we implemented the EDF scheduler using a binomial heap, which results in $O(log(n))$ when dequeuing the highest priority job from the ready queue.

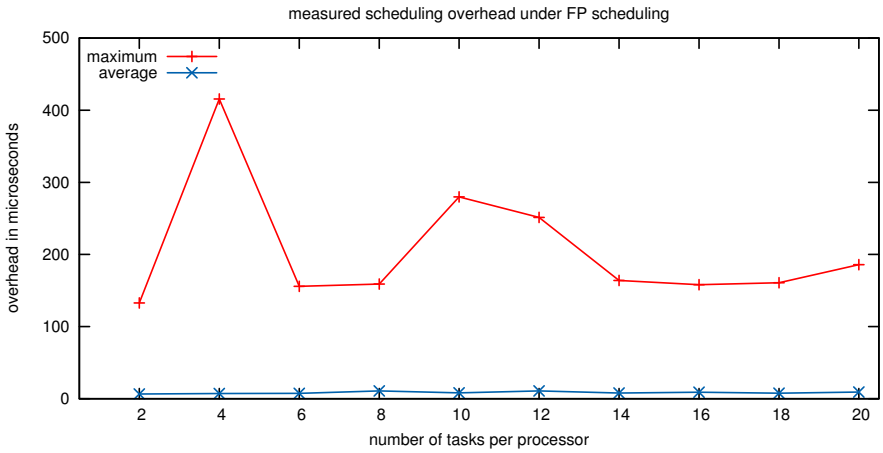


Figure 6.7: Scheduling overhead of the FP scheduler.

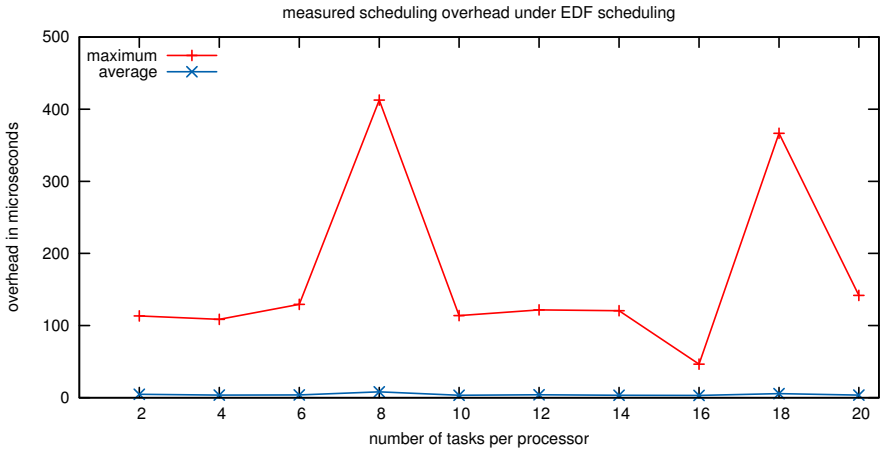


Figure 6.8: Scheduling overhead of the EDF scheduler.

As we can see, in practice we do not observe a significant difference between both implementations. This could be explained by the fact that the more-frequent invocation of the scheduler likely results in an increased cache hit rate, which lowers the cost of scheduler invocation on average.

The observation of the overheads in the average shows that the performance of user-level library are in the same order of magnitude as the overheads from a kernel-level approach. However, the overheads in the worst-case are generally far from the average-case. We suspect that the maximum overheads observed are caused by the preemption of the worker-thread when executing the switch-context and scheduling code executed by other workload running the microkernel. Also more investigations are required to determine with certainty the cause of the delays.

6.5.5 Comparison with Similar Approaches

As a comparison, we evaluated the library on top of a Linux kernel configured with PREEMPT_RT real-time patch and we tested it on a real hardware Intel Core i7 2.6GHz. The measurement showed that the context switch overhead is equal to $0.50\mu s$ in the average-case, and $21.34\mu s$ in the worst-case, the scheduling overhead is equal to $0.89\mu s$ in the average-case and to $21.71\mu s$ in the worst-case.

We also compared the library to a native RTOS, LITMUS$^{\mathrm{RT}}$, which we installed on an Intel Core 2 2.4GHz real hardware, we observed that the context-switch overhead is equal to $1.83\mu s$ in average, and $8.07\mu s$ in the worst-case. The average scheduling overhead in LITMUS$^{\mathrm{RT}}$ is around $3.78\mu s$ and the maximum is equal to $18.36\mu s$.

These comparisons show that the user-level library performed competitively to the kernel-based approach. The reason for this similarity is related to the fact that in the average-case, the execution cost of the scheduling function and context-switch is independent from the level of privilege that these functions are executed at. For instance, the scheduling function is whether executed at privileged-level or unprivileged-level does not impact its execution cost because the code runs at the same processor speed rate in the both privilege-levels.

However, the difference emerges in the worst case because the user-level library's functions are more subject to interruption than the kernel-level functions due to the preemption caused by the execution of other threads. For example, in our experiment the timer driver thread was assigned the same priority as the worker-thread of the library.

As a comparison with other user-level implementation on other microkernel version, we can mention the user-level scheduling on the seL^4 microkernel (Åsberg and Nolte, 2012), that was tested

Measurement	Platform	Time (μs)
library Scheduling	Intel Core 2 2.4GHz (Nova)	8.01
library Context switch	Intel Core 2 2.4GHz (Nova)	2.72
library Scheduling	Intel Core i7 2.6GHz (Linux/PREEMPT_RT)	0.89
library Context switch	Intel Core i7 2.6GHz (Linux/PREEMPT_RT)	0.50
Scheduling	Intel Core 2 2.4GHz (LITMUS$^{\mathrm{RT}}$)	1.83
Context switch	Intel Core 2 2.4GHz (LITMUS$^{\mathrm{RT}}$)	3.78
PS Scheduling (Åsberg and Nolte, 2012)	Intel P3 533MHz (seL^4)	213
PS Context switch (Åsberg and Nolte, 2012)	Intel P3 533MHz (seL^4)	109
Set timer (Yang et al., 2011)	AMD Athlon 2GHz (L^{4}Fiasco)	236
System call (Blackham et al., 2011)	ARM-A8 800MHz (seL^4)	20
IPC (Hessel et al., 2008)	ARM-11 416MHz (L^{4}Fiasco)	35/54

Table 6.2: Overheads comparison.

on a Qemu emulator of the Intel P3 533MHz, where the overhead of invoking the scheduler was equal to 213μs, and the context-switch equals to 109μs.

In the L^{4}Fiasco microkernel (Yang et al., 2011), the overhead of setting a timer in a hierarchical scheduling framework takes in average 236μs on 2GHz AMD Athlon processor. The measured time of a system call (seL4_Send(), seL_Wait(), seL4_ReplyWait()) in seL^4 on an ARM Cortex-A8 800MHz takes approximately 20μs. And the duration of an IPC in the L^{4}Fiasco microkernel (Hessel et al., 2008) running on an ARM1176 416MHz processor varied between 35μs and 54μs. Table 6.2 summarizes all these comparisons.

One critical metric for our user-level library performance is the signal latency, which is the duration of the transmission of the preemption signal from the listener-thread to the worker-thread. We observed that the signal latency takes 89.42μs in average case and a maximum of 445.66μs. The maximum signal latency observed could be explained by the fact that, in a microkernel-based OS, the operation of sending a message from one thread to another involves two IPC calls, plus one invocation of the kernel scheduler and one context-switch.

Based on these measurements, the overheads and latencies of the user-level library do not seem overwhelming, i.e., the overheads are at least not orders of magnitude larger than general system overheads in kernel-level implementation and other user-level implementations.

6.6 Summary

The Nova microkernel is a high performance kernel, its small *trusted computing base* makes it suitable for safety-critical systems. That means, if an application behaves correctly, in terms of functionality, then it will never be disrupted or fail due to a faulty kernel or other error-prone applications. However, in order to make it more suited for real-time systems, new resources allocation methods and techniques need to be supported by the microkernel.

We have ported a user-level library to implement new resource allocation techniques at user-level on top of the Nova microkernel. This solution also simplifies the configuration of the operating system by avoiding the changing of its internal kernel, and thus resolves the problem defined in Section 5.4.6 that raised after the transformation from a component-based RTOS model to an executable RTOS model.

The evaluation of the user-level library on top of the Nova microkernel revealed that the average-case overheads of a scheduler invocation and an execution of a context-switch were similar to the same overheads from kernel-level implementation.

However, worst-case observed overheads and latencies are higher than the same metrics from a kernel-level implementation. Further investigations need to be pursued in a future work as well as contributing to the development of the Nova microkernel to allow its deployment on a real hardware. After that, more engineering effort is needed to support multicore hardware platform and enable all the features offered by the library.

CHAPTER 7

Conclusions

The research work presented in this dissertation attempted to answer two main questions, first how to co-locate a real-time OS and a general-purpose OS on a common hardware platform, second how to transform a component-based RTOS simulation model into an RTOS executable model without loosing the degree of configuration offered by the initial design.

The availability of new hardware platforms that support virtualization oriented our work towards the use of a virtual machine system as an answer to the first question. But it created a new subsequent question regarding the overhead of running an RTOS on a virtual machine. In Chapter 3 we conducted two evaluations in order to measure the overhead of a guest RTOS running on a virtual machine.

These evaluations focused on analyzing how the virtualization of the main three hardware resources notably the processor, the memory management unit, and the I/O impacted the performance of the guest RTOS.

In the first evaluation we measured the global scheduling latency of a guest RTOS, and compared it to a native RTOS. The result showed that the scheduling latency of the guest RTOS incurred a slight increase in the average case comparing to a native RTOS. However we observed a maximum scheduling latency on the guest RTOS that is two orders of magnitude higher than the native RTOS. Although the probability of such a delay is low, it does not represent a confident worst-case result. This leads us to conduct a second evaluation in which we decomposed the global scheduling latency in a set of fine-grained overheads and latencies internal to the RTOS kernel, and we measured each overhead individually.

The results of this evaluation showed that all kernel overheads of a guest RTOS are comparable to a Native RTOS in the average-case, except for the event-latency, where we observed a slight in-

crease comparing to the same latency from a native RTOS. This observation explained the difference between the global scheduling latencies measured in the first evaluation. An increase that is related to the virtualization of I/O interrupts, and could be alleviated with an assistant from the underlying hardware.

We observed that some worst-case values of the kernel overheads were very far from the average-case values, and their probability was also very low, this result resembles the result from the first evaluation.

While it is difficult to state with certainty that there is an upper limit for the overhead of a guest RTOS, we are able to state that a real-time application that is running on a guest RTOS should expose the same performance as if it was running on a native RTOS, except in extreme rare case where its timing requirement may not be respected.

This suggested that beside the overhead of the virtualization, the scheduling of the virtual machines by the host system is involved as well in guaranteeing the quality of service required by a guest RTOS and the real-time application running on top.

In Chapter 4 we analyzed how the scheduling of virtual machines impacts a guest real-time application, we analyzed a scheduling technique based on the periodic resource model (PRM) that guarantee the temporal isolation among the virtual machines hosted by the system, and we reviewed a method to calculate the optimal scheduling parameters to allocate efficiently the CPU resource for all the hosted virtual machines.

Furthermore, we proposed an extension of this scheduling technique by integrating the overheads measured in our evaluation, and we showed that it is possible to guarantee the temporal isolation between the virtual machines.

Based on these results, our answer to the first question is as follows: the hardware support for the virtualization of the CPU and the memory management unit permit to run an RTOS on a virtual machine at the same speed rate as on a real hardware. The performance of a guest RTOS could also be improved if the I/O are virtualized efficiently. It is also necessary to configure carefully the scheduling of virtual machines to avoid any overhead from affecting the predictability of the system.

In our work pertaining to the transformation from an RTOS simulation model to an RTOS model executable on a real hardware, we proposed a method based on a model-driven engineering technique. We showed how a model-to-model transformation is used to extract the structural part

of a component-based RTOS model, then we used a model-to-code transformation to automatically generate the source code of the executable programs.

We analyzed the limitation of the proposed method concerning the support of configuration in the RTOS executable model. Then we proposed a solution based on the adoption of a middleware design in which the resource allocation policies could be supported at user-level library that could be reused across a variety of operating systems.

7.1 Open Question and Future Works

First, our research team is currently porting the Nova microkernel to the ARM architecture, thereafter the periodic resource model should be implemented on the x86 version and the ARM version in order to make the microkernel suitable for use in real-time systems.

Second, more engineering effort is required to enable the full-support of the multi-core hardware by the microkernel version of the user-level library.

One open research problem concerns the major limitation of the user-level library, this limitation is related to providing strong memory isolation to the user-level threads managed by the library. In the current version of the library, the user-level threads created by the library are in the same memory address space, this means that there are no physical barriers between the tasks that could prevent a ”misbehaving” task from corrupting the memory of another task. While, this problem could be alleviated by partitioning the application using multiple group of tasks, wherein the tasks in one group trust each others, and let the library schedules the task groups. It would be more secure to make the underlying operating system support the memory isolation at user-level. This could be achieved by providing system calls that could be used by a user-level library to ensure spatial isolation between its managed threads.

CHAPTER 8

Résumé de la thèse

Nos travaux de recherche explorent les solutions qui permettent de faire co-habiter sur une même unité de calcul plusieurs systèmes d'exploitation. Dans le domaine des serveurs d'entreprise, il existe une solution qui a fait ses preuves, la *virtualisation*. Elle consiste à "faire croire" au système d'exploitation qu'il contrôle réellement le matériel, mais en réalité il ne contrôle qu'une machine *virtuelle*. Cette solution a récemment sucité l'intérêt de la communauté des chercheurs dans le domaine des systèmes temps-réel embarqués. Notamment, le support de la virtualisation par des processeurs historiquement destinés à être utilisés dans des systèmes-sur-puce a permis de nouvelle applications dans les domaines de l'automobile, de l'avionique et des télécommunication mobiles.

Utiliser la virtualisation pour faire co-habiter plusieurs systèmes d'exploitation pour des applications temps-réel embarquées semble être une solution valide. La question que cette thèse cherche à résoudre est : quel est le surcoût de la virtualisation sur un système d'exploitation temps-réel ?

Le deuxième axe de recherche de nos travaux concerne l'exploration et le co-design logiciel/-matériel pour des systèmes-sur-puces reconfigurables. En particulier, notre travail vise à proposer une méthode qui permet de générer automatiquement des programmes executables sur une cible matérielle à partir de modèles de simulation. Ces modèles de simulation ont été développés pour explorer les differents choix de conception logiciel et matériel à haut-niveau, c'est à dire, avant l'implementation sur une puce de silicium. Une fois que le choix de conception validé au niveau simulation, une solution doit être implémentée. Après partitionnement, une partie de la solution sera implémentée en logiciel, et une partie sera implémentée en matériel sur une puce de silicium.

C'est la partie logicielle qui nous intéresse dans cette thèse. Notamment, dans la phase d'exploration tous les modèles ont été décrits dans un langage de simulation appelé SystemC, y compris les modèles qui répresentent le logiciel parce qu'à ce stade nous ne savons pas quel composant allait

être implémenté en logiciel ou en matériel. Le but de notre travail dans cette thèse est de proposer une méthode qui permet de générer automatiquement à partir des modèles de simulation les programmes qui seront exécutés sur la puce électronique. Pour cela, nous avons adopté une technique issue de l'ingénierie-dirigée par les modèles.

8.1 Etat de l'art sur la virtualisation

Il existe une multitude de solutions qui proposent l'utilisation de machines virtuelles pour faire co-habiter sur une même platforme matérielle plusieurs systèmes d'exploitation. Chaque solution adopte une architecture logicielle différente, la Figure 8.1 récapitule toutes les architectures.

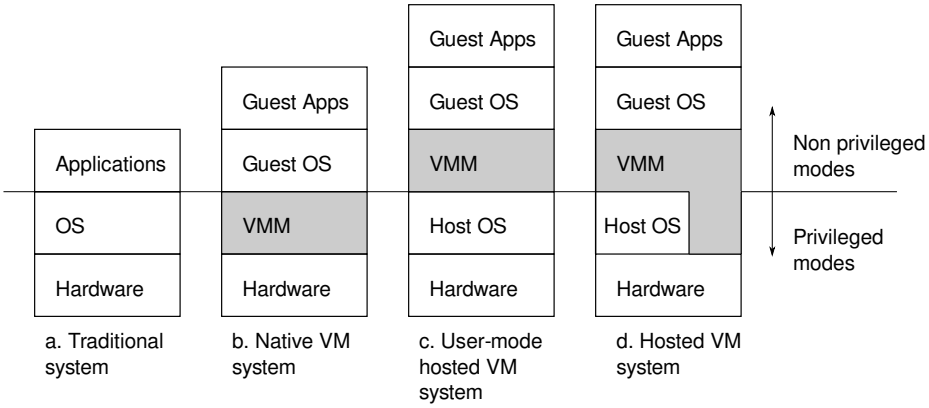


Figure 8.1: Architecture logicielle des systèmes supportant des machines virtuelles.

Parmi les architectures présentées sur la Figure 8.1, deux sont utilisées pour rendre compatible la virtualisation avec les contraintes d'un système temps-réel. Notamment le système de machine virtuelle natif (Figure 8.1b) et le système de machine virtuelle *hybride* (Figure 8.1d). Dans le premier cas, toutes les machines virtuelles sont contrôlées par un *hyperviseur* (appelé aussi *Virtual Machine Monitor*), il s'agit d'un composant logiciel installé directement sur le matériel. Dans le second cas, un système d'exploitation standard est installé sur le matériel puis étendu par une partie de l'hyperviseur, l'autre partie de l'hyperviseur est implémentée sous forme d'application "utilisateur classique".

Nous allons dans les sections suivantes examiner des implémentations de ces deux architectures et voir comment elles ont été adaptées aux contraintes d'un système temps-réel.

8.1.1 Linux Kernel Virtual Machine

Le système d'exploitation Linux a été étendu pour pouvoir supporter l'exécution de plusieurs machines virtuelles. Il s'agit d'un hyperviseur de type *hybride*, où la partie qui se charge de contrôler la virtualisation du matériel (appelée kvm) est intégrée au noyau Linux, et la partie qui se charge de *virtualiser* les périphériques d'entrée/sortie (I/O) est implémentée par l'émulateur Qemu. Le module kvm est responsable de la virtualisation du processeur, de l'unité de gestion mémoire (MMU) et du timer. Qemu s'occupe d'émuler le disque, la carte réseau, l'écran VGA, *etc.*

Le noyau Linux est responsable de l'ordonnacement des machines virtuelles ainsi que des allocations mémoires demandées par les systèmes d'exploitation *invités*, c.-à-d. fonctionnant sur les machines virtuelles.

Deux principales méthodes ont été utilisées pour rendre Linux-kvm compatible avec les contraintes temps-réel. D'une part, le noyau Linux a été configuré par le *patch* temps-réel, "PREEMPT_RT", afin d'améliorer sa "préemptabilité" et réduire sa latence d'ordonnancement (*scheduling latency*).

D'autre part, l'ordonnancement dans le noyau Linux a été modifié (Cucinotta et al., 2009a) afin d'intégrer l'activation périodique des machines virtuelles selon l'algorithme CBS (*Constant Bandwidth Server*), chaque machine virtuelle se voit attribuer un couple appelé *periodic resource model* (PRM) défini par un budget et une période, (Θ, Π). Ensuite l'ordonnanceur alloue le processeur à la machine virtuelle pour une durée égale à son budget à chaque activation de sa période.

Plusieurs études (Bing, 2010; Kiszka, 2010; Zhang et al., 2010b; Zuo et al., 2010) ont évalué la première méthode en utilisant le benchmark cyclictest pour mesurer la latence d'ordonnancement d'un système d'exploitation *invité*. Les résultats de ces évaluations ont montré que l'utilisation d'un noyau Linux temps-réel a considérablement amélioré cette latence.

Cependant, la majorité de ces analyses n'expliquent pas clairement comment la virtualisation du processeur, la virtualisation de la MMU ainsi que des interruptions influencent le surcoût subi par l'OS invité.

D'autre part, l'évaluation de la méthode qui repose sur un ordonnancement de type CBS a montré que ce type d'ordonnancement est nécessaire pour garantir le respect des contraintes tem-

porelles d'une application. Mais, cette étude n'a malheureusement pas mesuré le surcoût subi par un système d'exploitation temps-réel (RTOS).

Dans notre étude (voir Section 8.3), nous avons cherché à determiner comment les mécanismes de virtualisation offerts par le matériel sous-jacent, notamment la virtualisation du CPU, de MMU, et des périphériques d'entrée/sortie, impactent le surcoût subi par l'OS dû à la virtualisation. Notre méthode vise à concilier les deux méthodes, c.-à-d. investiguer le rôle de l'ordonnancement des machines virtuelles dans le surcoût subi par l'OS en tenant compte de l'influence des mécanismes matériels liés à la virtualisation.

8.1.2 Virtualisation basée sur le Micro-noyau

Plusieurs systèmes d'exploitation de type micro-noyau ont été utilisés en tant que hyperviseur. Le microkernel OKL^4 développé par Open Kernel Labs a été porté sur le nouveau jeu d'instruction ARMv8 afin de pouvoir supporter l'exécution de systèmes d'exploitation invités sans aucune modification à leur code source (Varanasi and Heiser, 2011). L'évaluation des fonctionalités unitaires de ce prototype en utilisant un modèle SystemC du processeur ARM Cortex A15 a donné des performances jugées intéressantes ce qui a encouragé le développement d'un produit commercial à partir de ce prototype. Cependant, aucune mesure du surcoût de la virtualisation sur un RTOS n'a été effectuée.

Le *microhypervisor* Nova est la 3ème generation des microkernel L^4 (Liedtke, 1996), sa particularité est qu'il intégre le support de la virtualisation dès sa phase de conception. Nova exploite les mécanismes matériels offerts par la récente architecture x86 pour proposer une virtualisation efficace du processeur et de la MMU. Une comparaison de la compilation du code source du noyau Linux faite sur une machine virtuelle exécutée par Nova, par Linux-kvm, et par Xen a montré que Nova permet d'atteindre les meilleures performances. Ici aussi aucune évaluation du surcoût de la virtualisation sur un RTOS n'a été effectuée.

Le microkernel L^{4}Fiasco a été adapté par Yang et al. (2011) afin de le rendre compatible avec les contraintes d'un système temps-réel. Le framework HSF (*Hierarchical Scheduling Framework*) a été implémenté. Il consiste à proposer deux niveaux d'ordonnancement, un premier niveau implémenté par l'hyperviseur afin d'allouer les ressources CPU aux machines virtuelles, et un sec-

ond niveau d'ordonnancement implémenté par chaque système d'exploitation invité pour arbitrer l'exécution des tâches temps-réel.

L'ordonnancement des machines virtuelles dans le framework HSF utilise le modèle PRM, le même qui est utilisé par le framework CBS, où chaque machine virtuelle se voit attribuer un couple $\Gamma = (\Theta, \Pi)$. Le calcul du PRM est réalisé en fixant d'abord la période Π et en calculant la valeur de Θ en utilisant le théorème 2.1 dans le cas d'un ordonnancement de type *rate-monotonic*.

L'évaluation de l'ordonnancement HSF et sa comparaison avec l'ordonnancement *rate-monotonic* et *round-robin* a montré que le HSF est plus apte à garantir le respect des échéances des tâches temps-réel.

Cependant, le test des propriétés temps-réel de l'hyperviseur L^{4}Fiasco repose sur l'utilisation d'une version particulière de Linux en tant que système d'exploitation invité, c'est le L^{4}Linux. Le L^{4}Linux est un Linux *para-virtualisé*, c.-à-d. qui a été modifié afin de remplacer certaines fonctionalités "sensibles" de sa "HAL" (hardware abstraction Layer) par des appels systèmes à l'hyperviseur (*hyper-calls*). Ces fonctionalités sensibles sont des opérations qui nécessitent un privilège, celui-ci est accordé au système d'exploitation uniquement lorsqu'il a le droit de contrôler le matériel. Ce n'est pas le cas lorsque le système d'exploitation s'exécute sur une machine virtuelle, car seul l'hyperviseur a ce privilège.

Dans notre étude, nous nous sommes plus intéressés à évaluer des systèmes d'exploitation non modifiés et voir comment les avancés technologiques proposées par les récentes architectures matérielles influencent les propriétés temps-réel d'un hyperviseur.

8.1.3 Xen

Xen est un hyperviseur de type natif déstiné initialement à être utilisé sur des machines-serveurs d'entreprise, il est de plus en plus utilisé dans le domaine des systèmes temps-réel, notamment sa version temps-réel, RT-Xen (Xi et al., 2011), et embarqué, Xen-ARM (Yoo and Yoo, 2013).

Dans le projet RT-Xen, quatre nouveaux algorithmes ont été implémentés, le *Defferable Server*, le *Periodic Server*, le *Polling Server* et le *Sporadic Server*. Ces algorithmes reposent tous sur la théorie d'ordonnancement hiérarchique que nous avons déjà rencontré lorsque nous avons décrit les frameworks HSF et CBS (voir Section 8.1.1 et Section 8.1.2).

Dans RT-Xen, le mot *server* est synonyme du mot composant utilisé par la théorie, et du mot machine virtuelle utilisé en pratique. Chaque composant géré par RT-Xen est défini par un budget, une période, et une priorité. La différence entre tous les algorithmes mentionnés est la façon dont le budget de chaque composant est depensé ou renouvellé. Dans le *Periodic Server* le budget du composant est consommé lorsque le composant en question est ”idle”, c.-à-d. lorsqu'il n'exécute aucune tâche, alors que dans le *Polling Server* il est préservé. Dans le *Defferable Server*, le budget est utilisé par un autre composant moins prioritaire mais qui en a réellement besoin, c.-à-d. qu'il a des tâches à exécuter mais sa période n'est pas encore arrivée. Alors que tous ces algorithmes renouvellent les budgets des composants de façon périodique (à intervalle fixe), le *Sporadic Server* les renouvelle au fur et à mesure qu'ils sont épuisés.

Les composants sont triés par ordre de priorité fixe selon une politique de type *rate-monotonic*, le composant ayant la plus haute priorité est celui qui a la plus petite période.

Une évaluation empirique a montré que le *Defferable Server* permet d'atteindre les meilleures performances en terme de respect des échéances à cause de sa meilleure gestion du budget non consommé. Les résultats ont montré qu'il est meilleur que le *Credit Scheduler* et le SEDF, les deux algorithmes d'ordonnancement par défaut dans Xen. En contre partie, la mesure du surcoût de ces algorithmes a montré qu'il est supérieur à celui de Credit Scheduler et SEDF parce que la gestion des listes des composants telle que *run-queue*, *ready-queue*, et *replenishment-queue*, est plus couteuse que la gestion d'une liste simple selon une politique *round-robin* ou *rate-monotonic* dans le cas de Credit Scheduler par exemple. Mais, ce surcoût varie entre 0.21% et 0.23% du temps CPU ce qui est favorable à l'utilisation de ces algorithmes.

Une seconde étude a complété ces premiers travaux dans RT-Xen en proposant deux nouveaux algorithmes, le *work-conserving periodic server* (WCPS) et le *capacity reclaiming periodic server* (CRPS) pour améliorer l'algorithme *Periodic Server*.

L'idée consiste à ne pas gaspiller le budget d'un composant lorsqu'il est ”idle” en le donnant à un autre composant. Le CRPS diffère du WCPS dans le fait que lorsqu'un composant prête son budget non utilisé à un autre, seul le budget qui a été donné est consommé alors que dans le cas du WCPS le budget de celui qui donne et le budget de celui qui reçoit sont consommés.

De plus, une méthode analytique a été proposée pour calculer le modèle de ressources, PRM, définit par $\Gamma = (\Theta, \Pi)$, pour chaque composant. Cette méthode permet d'assurer que ces paramétres

garantissent une utilisation efficace des resources de calculs (CPU) et "l'ordonnançabilité" des composants et de leurs tâches temps-réel, alors que dans les premiers travaux le modèle PRM est fixé manuellement par le developpeur dans la phase de conception.

L'évaluation empirique a montré que le CRPS est meilleur que le WCPS et le *Periodic Server* dû à sa politique efficace pour la consommation du budget non utilisé.

D'autre part, l'utilisation de Xen sur des systèmes embarqués comme les smartphones a révélé un problème lié au quantum d'ordonnancement, qui est une valeur entière par défaut égale à $10ms$.

Lorsque cette valeur est augmentée, elle risque de rallonger le temps de réponse d'un système d'exploitation invité car le temps de ré-ordonnancement est plus long, et lorsqu'elle est diminuée elle risque d'allourdir le surcoût dû à la virtualisation car il y a plus de changement de contexte entre les machines virtuelles. Elle intérfère également avec le système de gestion d'énergie puisque à chaque "tick" d'ordonnancement le système est réveillé ce qui peut nuire à la batterie.

Dans le cadre d'un ordonanncement hiérarchique, un composant se voit attribué un budget qui doit avoir une valeur entière multiple du quantum d'ordonnancement. Or les méthodes théoriques qui calculent le PRM (Θ, Π) d'un composant supposent que cette valeur soit réelle, ce qui oblige à l'arrondir pour qu'elle soit multiple du quantum d'ordonnancement, la quantité ajoutée est considérée alors comme un surcoût appelé "quantization overhead", définie par :

$$\Delta(\Pi) = \frac{\Theta'}{\Pi} - \frac{\Theta}{\Pi} = \frac{\Theta' - \Theta}{\Pi} \tag{8.1}$$

Comme on peut le voir dans l'équation 8.1, augmenter la période Π implique la diminution du "quantization overhead" :

$$\forall \alpha \geq \alpha *, \Delta(\Pi) > \Delta(\Pi + \alpha) \tag{8.2}$$

D'autre part, le PRM donné à un composant doit être supérieur à la charge totale demandée par les tâches exécutées par le composant :

$$U_W = \sum_i \frac{e_i}{p_i} \tag{8.3}$$

$$\frac{\Theta}{\Pi} \geq U_W \tag{8.4}$$

La différence entre le ratio du PRM et la charge totale est aussi considérée comme un surcoût appelé ”abstraction overhead” :

$$\Psi(\Pi) = \frac{\Theta}{\Pi} - U_W \tag{8.5}$$

Comme on peut le voir augmenter la période Π implique l'augmentation du ”abstraction overhead” :

$$\forall \alpha \geq \alpha *, \Psi(\Pi) > \Psi(\Pi + \alpha) \tag{8.6}$$

Seehwan Yoo and Chuk Yoo ont proposé l'algorithme nommé *SH-Quantization* (Yoo and Yoo, 2013) qui permet de calculer la période et le budget d'un composant de façon optimale par rapport aux deux contraintes ”quantization overhead” et ”abstraction overhead”.

8.1.4 Virtualisation pour les systèmes critiques

Plusieurs travaux de recheche ont considéré l'utilisation d'un hyperviseur dans le domaine des systèmes critiques. L'un des principaux besoins concerne les systèmes avioniques et le respect de la norme ARINC 653 qui régit le développement logiciel de ces systèmes. Cette norme préconise le partitionnement temporel et spatial entre les différentes applications exécutées sur une seule plateforme matérielle. Ce qui correspond naturellemet aux spécifications d'un hyperviseur puisqu'il est capable d'isoler les machines virtuelles en utilisant une MMU, et garantir pour chaque machine virtuelle les ressources CPU qui lui sont allouées et empêcher tout dépassement par exemple en spécifiant pour chaque machine virtuelle un PRM (Θ, Π).

XtratuM est un hyperviseur de type natif destiné à être utilisé dans des applications aérospatiales (Masmano et al., 2009, 2010; Carrascosa et al., 2013). La dernière version de XtratuM est disponible sur l'architecture Sparc V8. Elle bénéficie de la MMU et du multicoeur offert par cette architecture. Cependant, le système d'exploitation invité qui est géré par XtratuM est *para-virtualisé* afin qu'il puisse utiliser les *hyper-calls* proposés par l'hyperviseur pour demander l'exécution d'une

opération ”privilégiée” sur le matériel. Cette modification du système d'exploitation est inévitable car l'architecture Sparc V8 n'a que deux niveaux de privilège : ”user” et ”supervisor”, forçant le système d'exploitation invité à s'exécuter au niveau ”user” et l'hyperviseur au niveau ”supervisor”.

Un autre hyperviseur destiné également pour une application aérospatiale a été dévelopé par Tavares et al. (2012). Il est compatible avec l'architecture PowerPC 405 disponible sur un FPGA Xilinx. Contrairement à XtratuM, il bénéficie d'une propriété offerte par cette architecture matérielle; les instructions ”sensitive”[1] du jeu d'instruction sont aussi des instructions privilégiées, ce qui permet à l'hyperviseur d'intercepter toutes les instructions ”sensitive” exécutées par un système d'exploitation invité parce qu'une instruction privilégiée génére une exception lorsqu'elle est exécutée dans un niveau de privilége différent du niveau ”supervisor”, ce qui est le cas de l'OS invité.

L'évaluation des deux hyperviseurs s'est limitée à la mesure de certaines opérations de bas-niveau tel que le changement de contexte, ou la prise en compte d'interruptions, mais aucune évaluation de l'impact de la virtualisation sur un RTOS ou une application temps-réel n'a été conduite en utilisant ces hyperviseurs.

8.2 Etat de l'art sur la configuration des systèmes d'exploitation

Notre intérêt pour la configuration d'OS émerge d'un besoin industriel qui vise à transformer le système d'exploitation temps-réel développé dans le projet OveRSoC (Miramond et al., 2009), depuis sa ”forme” simulable vers une ”forme” exécutable sur une plate-forme réelle. Le modèle représentant le système d'exploitation temps-réel (RTOS) ainsi que les modèles qui représentent la plate-forme SoC reconfigurable (RSoC) sont décrits dans un langage de simulation (logiciel/-matériel) appelé SystemC.

La particularité de l'OS ”OveRSoC” réside dans son architecture basée sur l'approche composant. Spécifiquement, l'OS est composé d'un ensemble de services où chaque service est un composant indépendant des autres composants. Les composants sont connectés entre-eux par l'intermédiaire d'interfaces et de ”ports”. Ceci permet de remplacer le comportement d'un composant par un autre sans ”casser” tout l'ensemble à condition de ne pas modifier les interfaces.

[1] Une instruction qui manipule l'état du matériel.

Cette facilité de modification des services de l'OS lui confère une grande capacité d'adaptation notamment par rapport au besoin de l'application. L'objectif de notre travail serait donc de transformer automatiquement le modèle d'OS simulable vers un modèle exécutable sans perdre en capacité d'adaptation.

Plusieurs travaux de recheche ont étudié la configuration, par exemple Composite (Parmer, 2010) qui est un OS basé sur l'approche composant. Composite repose sur un module intégré au noyau Linux appelé $\text{Hijack}^{\text{COS}}_{\text{Linux}}$ (Parmer et al., 2012) afin de "pirater" les fonctionalités clés du noyau et prendre le contrôle de la machine. Il exporte ensuite au niveau utilisateur (user-level) un certain nombre de services qui permettent à des composants définis au niveau utilisateur de prendre des décisions sur l'allocation des ressources matérielles.

Un deuxième exemple est le système d'exploitation $\text{LITMUS}^{\text{RT}}$ qui est une modification (patch) du noyau Linux. Il modifie le noyau Linux afin de détourner la gestion d'interruption et l'ordonnancement vers ses propres fonctions. Ces fonctions appellent ensuite des fonctions qui sont définies dans des extensions (plug-ins) au niveau utilisateur. Les plug-ins décident alors de la politique de gestion des ressources en utilisant des algorithmes d'ordonnacement temps-réel pour le multicoeur ou bien des protocoles de synchronisation des ressources partagées.

Un troixième exemple est le framework ExSched qui est une extension (module) ajouté au noyau Linux. Il permet également de remplacer des fonctionalités clés du noyau Linux comme l'ordonnancement sans modifier le noyau. Il dépend des fonctions offertes par Linux comme par exemple la fonction schedule(), à l'aide de cette fonction il est capable de demander un ordonnancement d'un jeu de tâches qui a été préalablement trié par l'un de ses plug-ins suivant une politque d'allocation de ressources propre à chaque plug-in. ExSched a également été porté pour être compatible avec le RTOS VxWorks.

Dans notre étude, nous avons décidé d'implémenter la configuration en utilisant une approche de type *middleware*, c.-à-d. en n'utilisant que des mécanismes (compatibles avec le standard POSIX) accessibles depuis le niveau-utilisateur (user-level) et ceci dans le but d'éviter toute dépendance avec le noyau du système d'exploitation sous-jacent.

8.3 Impact de la Virtualisation sur les Systèmes Temps-Réel

Notre étude de l'impact de la virtualisation nous a amené à conduire trois évaluations progressives. Dans une première évaluation, nous avons mesuré la *latence d'ordonnancement* qui est un délai séparant le déclenchement d'une interruption (par exemple suite à un évenement issu d'un capteur, ou une interruption timer) et le début de l'exécution du "job" de la tâche qui lui correspond. Ensuite nous avons décomposé ce délai en un ensemble de surcoûts (overheads) et de latences (latencies) internes au système d'exploitation, afin de determiner l'opération la plus pénalisée par la virtualisation. Enfin nous avons conduit deux études de cas en utilisant une applications temps-réel simple pour valider les conclusions déduites des deux premières évaluations.

La conclusion que nous avons retenue des deux premières évaluations est qu'un système d'exploitation doit exposer les même performances lorsqu'il exécuté sur une machine virtuelle que lorsqu'il est exécuté sur une machine réelle. Seulement dans des cas rares il est susceptible de subir une dégradation dans ses performances.

La troixième expérience a corroboré cette conclusion, le test de plusieurs applications temps-réel a démontré que les échéances ont été respectées aussi bien en exécutant les applications sur une machine réelle que sur une machine virtuelle.

8.4 Ordonnancement Temps-Réel des Machines Virtuelles

Nous avons analysé comment l'ordonnancement des machines virtuelles peut causer le non respect des échéances des tâches temps-réel. Nous avons alors étudié la méthode qui permet d'éviter ce problème et qui repose essentiellement sur l'utilisation du modèle PRM (periodic resource model) afin d'attribuer à chaque machine virtuelle un couple (Θ, Π) définissant le partage des ressources matérielles et force chaque machine virtuelle à garantir le respect des contraintes temporelles des tâches qu'elle exécute.

8.5 Transformation d'un modèle d'OS Temps-Réel

La transformation d'un modèle d'OS simulable vers un modèle d'OS exécutable est basée sur une technique issue de l'ingénieurie-dirigée par les modèles. Cette technique fait appel à une double transformation : une "Model-to-Model" et une "Model-to-Code".

Dans un premier temps, le modèle d'OS simulable est transformé vers un modèle OS abstrait qui reflète uniquement la structure de l'OS, ensuite le modèle abstrait est transformé en un code source qui sera par la suite compilé en programmes qui s'exécuteront sur la plate-forme matérielle.

La transformation "Model-to-Model" a nécessité la création d'un meta-modèle, c.-à-d. un langage de modélisation (grammaire + syntaxe) pour décrire la structure d'un OS. Ceci nous a permis d'extraire les informations nécessaires sur les composants de l'OS depuis le modèle simulable.

Le modèle d'OS qui représente la structure de l'OS est ensuite transformé automatiquement en un code source, la génération de code source repose sur un ensemble de code source existant et paramétré (template). Le code source existant vient d'un RTOS existant qui propose les même fonctionalités que celles qui sont proposées par les composants formant le modèle d'OS simulable.

La limite de notre méthode vient du fait que le RTOS choisi comme base pour créer les *templates* utilisés par le générateur de code est un RTOS monolithique. Or, à l'origine le modèle d'OS simulable utilisé en entrée est un modèle basé sur les composants, offrant par conséquent une grande facilité d'adaptation, ce qui résulte en une perte de degré de configuration dans la transformation d'un modèle d'OS basé sur les composants vers un modèle d'OS monolithique.

Pour remédier à ce manque nous avons utilisé un middleware qui sera déployé sur le RTOS monolithique afin de le doter d'une capacité de configuration.

8.6 Utilisation d'une Libraire pour la Configuration d'OS

Doter un OS d'une capacité d'adaptabilité sans modifier son code source interne peut être réaliser en utilisant un middleware. Ce middleware s'interpose entre l'OS et l'application afin de connaître le besoin en termes de ressources de l'application et de demander à l'OS l'allocation des ces ressources selon une politique qui est défini par le middleware et non pas par l'OS.

En utilisant les abstractions des ressources matérielles (processus) fournies par l'OS et en donnant une priorité maximale au processus qui constitue le middleware par rapport au reste des programmes exécutés par l'OS, le middleware est capable de prendre le contrôle de la machine. Une fois que le programme middleware a pris le contrôle de la machine, il est capable d'exécuter n'importe quel travail suivant sa propre "politique".

Pour exécuter plusieurs tâches, le middleware utilise une structure de données connue sous le nom de "u_context" dans la norme POSIX, et qui sert de "recipient" pour sauvegarder le contexte processeur (registres) d'une tâche exécutée par le processus middleware.

La norme POSIX prévoit un ensemble de fonctions qui permettent de créer des structures "u_context" et de faire des changements de contexte processeur (registres) depuis le niveau-utilisateur. En utilisant ces fonctions, un middleware peut multiplexer sur un processus plusieurs tâches (multi-tâches). Par la suite, lorsque le processus est exécuté par l'OS sous-jacent, c'est en réalité le code de la tâche élue par le middleware qui est exécuté.

Pour interrompre (de manière asynchrone) le travail d'un processus, le middleware utilise les interruptions logicielles d'un "timer". Lorsqu'un "timer" se déclenche, il envoie un signal au processus middleware pour l'interrompre. A ce moment, l'exécution du processus est dirigée vers une fonction particulière du middleware timer_handler() qui va gérer la suite de l'exécution. Cette fonction re-programme d'abord le timer pour un prochain réveil ensuite appelle la fonction schedule() du middleware pour choisir la prochaine tâche à exécuter.

Pour contrôler le temps d'exécution de chaque tâche, le middleware peut utiliser un registre particulier de la machine, par exemple sur l'architecture x86 le registre TSC compte le nombre de cycle processeur depuis le démarrage de la machine.

En étant capable de réaliser la préemption, la mesure du temps, le changement de contexte processeur au niveau utilisateur, un middleware peut implémenter n'importe quelle politique d'allocation de ressources.

Une implémentation de ce middleware a été proposée par Mollison and Anderson (2013). Dans sa version initiale, le middleware a été testé sur un système d'exploitation Linux temps-réel pour une architecture x86 32-bit. Les performances de son évaluation nous ont incité à le considérer comme une solution potentielle à notre problème de configuration d'OS. Cependant, il était nécessaire de vérifier la portabilité de cette approche sur un autre OS et notamment sur un micro-noyau afin que

cette solution soit optimale pour notre besoin. Nous avons implémenté le middleware sur le micro-noyau Nova pour une architecture x86 64-bit. L'évaluation des performances du middleware sur le micro-noyau a démontré la validité de l'approche.

8.7 Conclusion et futurs travaux

Dans cette thèse nous avons répondu à la question de l'impact de la virtualisation sur les propriétés d'un système temps-réel. L'utilisation des mécanismes matérielles pour virtualiser le processeur, la MMU et les périphériques d'entré/sortie permet de garantir à un système d'exploitation les même performances d'une machine réelle. Ces performances peuvent être altérées dans des cas extrêmement rares. Pour éviter cette dégradation dans les performances et garantir le respect des contraintes temps-réel il est nécessaire d'adopter un ordonnancement temps-réel au niveau de l'hyperviseur.

Nous avons également montré comment il est possible de transformer un modèle de RTOS simulable vers des programmes exécutables sur une cible matérielle. Et nous avons proposé une méthode qui évite la perte de la capacité de configuration d'un modèle de RTOS lorsqu'il est transformé vers un autre modèle.

Il serait aussi intéressant dans le futur d'améliorer l'implémentation du middleware sur le micro-noyau afin de proposer un framework complet capable de supporter un ensemble riche de politique d'allocation de ressources pour les architectures multicoeurs.

BIBLIOGRAPHY

Abeni, L. and Buttazzo, G. (1998). Integrating multimedia applications in hard real-time systems. *Real-Time Systems Symposium*, pages 4–13.

Accellera (2014). IEEE 1666-2011: SystemC. http://www.accellera.org/home/.

Aichouch, M., Miramond, B., and Huck, E. (2008). Oversoc platform design simulator. *Conference on Design and Architectures for Signal and Image Processing (DASIP)*.

ARM (2014). Fast Models. http://www.arm.com/products/tools/models/fast-models/.

Åsberg, M., Forsberg, N., Nolte, T., and Kato, S. (2011). Towards real-time scheduling of virtual machines without kernel modifications. *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–4.

Åsberg, M. and Nolte, T. (2012). Towards a user-mode approach to partitioned scheduling in the sel4 microkernel. *5th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'12)*.

Åsberg, M., Nolte, T., Kato, S., and Rajkumar, R. (2012). Exsched: An external cpu scheduler framework for real-time systems. *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*, pages 240–249.

Audsley, N., Burns, A., Richardson, M., Tindell, K., and Wellings, A. J. (1993). Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292.

Baker, T. P. (2005). A comparison of global and partitioned edf schedulability tests for multiprocessors. Technical report, In International Conf. on Real-Time and Network Systems.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. *Proceedings of the 19th ACM symposium on Operating systems principles*.

Bechennec, J.-L., Briday, M., Faucou, S., and Trinquet, Y. (2006). Trampoline an open source implementation of the osek/vdx rtos specification. *Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference on*, pages 62–69.

Behnam, M., Nolte, T., Shin, I., Åsberg, M., and Bril, R. J. (2008). Towards hierarchical scheduling on top of vxworks. pages 63–72.

Bellard, F. (2005). Qemu, a fast and portable dynamic translator. *Proceedings of the Linux Symposium*.

Bing, Z. (2010). Scheduling policy optimization in kernel-based virtual machine. *Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on*.

Blackham, B., Shi, Y., Chattopadhyay, S., Roychoudhury, A., and Heiser, G. (2011). Timing analysis of a protected operating system kernel. pages 339–348.

Brandenburg, B. and Anderson, J. (2007). Feather-Trace: A lightweight event tracing toolkit. *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 19–28.

Brandenburg, B., Block, A., Calandrino, J., Devi, U., Leontyev, H., and Anderson, J. (2007). LITMUSˆRT: A Status Report. *Proceedings of the 9th Real-Time Linux Workshop*, pages 107–123.

Brandenburg, B. B. (2011). *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill.

Carrascosa, E., Masmano, M., Balbastre, P., and Crespo, A. (2013). XtratuM hypervisor redesign for LEON4 multicore processor. *Workshop on Virtualizatio for Real-Time Embedded Systems*.

Cloutier, P., Mantegazza, P., Papacharalambous, S., Soanes, I., Hughes, S., and Yaghmour, K. (2008). Diapm-rtai position paper. *In Real-Time Linux Workshop*, 3.

Cook, S., Jones, G., Stuart, K., and Cameron Wills, A. (2007). *Domain-Specific Language Development with Visual Studio DSL Tools*. Addison-Wesley.

Cucinotta, T., Anastasi, G., and Abeni, L. (2009a). Respecting temporal constraints in virtualised services. *Computer Software and . . . .*

Cucinotta, T., Palopoli, L., Marzario, L., and Lipari, G. (2009b). Aquosa – adaptive quality of service architecture. software – practice and experience. Technical report.

Delange, J. and Lec, L. (2011). POK, an ARINC653-compliant operating system released under the BSD license. *13th Real-Time Linux Workshop*.

Deng, Z. and Liu, J. W. S. (1997). Scheduling real-time applications in an open environment. pages 308–319.

Faggioli, D., Checconi, F., Trimarchi, M., and Scordino, C. (2009). An edf scheduling class for the linux kernel.

Forsberg, N. (2011). Evaluation of real-time performance in virtualized environment. Technical report.

Garcia, P., Gomes, T., Salgado, F., Monteiro, J., and Tavares, A. (2013). Towards Hardware Embedded Virtualization Technology: Architectural Enhancements to an ARM SoC. *Workshop on Virtualizatio for Real-Time Embedded Systems*.

Gerum, P. (2008). *The Xenomai real-time system*. O'Reilly Media, 2nd edition.

Gleixner, T. (2013). The future of realtime Linux. `http://lwn.net/Articles/572740/`. Fifteenth Real-Time Linux Workshop.

Gleixner, T. and Niehaus, D. (2006). Hrtimers and beyond: Transforming the linux time subsystems. *In Proceedings of the 2006 Linux Symposium*, pages 333–346.

Gronback, R. (2009). *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Pearson.

Heiser, G. (2011). Virtualizing embedded systems: Why bother? *Proceedings of the 48th Design Automation Conference*, pages 901–905.

Hessel, S., Bruns, F., Bilgic, A., Lackorzynski, A., Härtig, H., and Hausner, J. (2008). Acceleration of the l4/fiasco microkernel using scratchpad memory. pages 6–10.

Holenderski, M., van den Heuvel, M. M. H. P., Bril, R. J., and Lukkien, J. J. (2006). Grasp: Tracing, Visualizing and Measuring the Behavior of Real-Time Systems. *Proceedings of the International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*.

Hwang, J.-y., Suh, S.-b., Heo, S.-k., Park, C.-j., Ryu, J.-m., Park, S.-y., Kim, C.-r., and History, A. V. (2008). Xen on ARM : System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones. pages 257–261.

IEEE (2014). Pthreads. http://standards.ieee.org/findstds/standard/1003.1c-1995.html.

Inam, R., Maki-Turja, J., Sjodin, M., Ashjaei, S. M. H., and Afshar, S. (2011). Support for hierarchical scheduling in freertos. In *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–10.

Intel (2012). *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation.

Joseph, M. and Pandya, P. (1986). Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395.

Kiszka, J. (2010). Towards linux as a real-time hypervisor. *Proceedings of the 11th Real-Time Linux Workshop*.

Kiszka, J. (2011). Using kvm as a real-time hypervisor. http://www.linux-kvm.org/wiki/images/0/03/KVM-Forum-2011-RT-KVM.pdf.

Kivity, A., Kamay, Y., Laor, D., and Lublin, U. (2007). kvm: the linux virtual machine monitor. *Proceedings of the Linux Symposium*.

Lackorzynski (2014). Running Linux on top of L4. http://os.inf.tu-dresden.de/L4/LinuxOnL4/overview.shtml.

Lee, J., Xi, S., Gill, C., Chen, S., Lee, I., Chen, S., Phan, L. T., and Sokolsky, O. (2011). Realizing compositional scheduling through virtualization. *Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*.

Lehoczky, J., Sha, L., and Ding, Y. (1989). The rate monotonic scheduling algorithm: exact characterization and average case behavior. pages 166–171.

Lelli, J., Lipari, G., Faggioli, D., and Cucinotta, T. (2011). An efficient and scalable implementation of global edf in linux. pages 6–15.

Lemerre, M., Ohayon, E., Chabrol, D., Jan, M., and Jacques, M.-B. (2011). Method and tools for mixed-criticality real-time applications within pharos. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on*, pages 41–48.

Liedtke, J. (1996). Toward real microkernels. *Commun. ACM*, 39(9):70–77.

Lin, B. and Dinda, P. A. (2005). Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. page 8.

Lip6 (2014). SocLib. http://www.soclib.fr/trac/dev.

Liu, C. L. (1969). Scheduling Algorithms for Multiprocessors in a Hard Real-Time Environment. *JPL Space Programs Summary 37-60*, II:28–31.

Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61.

LWN (2014). Deadline Scheduler Merged for 3.14. `http://lwn.net/Articles/581491/`. Linux Weekly News.

Masmano, M., Ripoll, I., and Crespo, A. (2009). Xtratum: a hypervisor for safety critical embedded systems. *Proceedings of the 11th Real-Time Linux Workshop*.

Masmano, M., Ripoll, I., Peiró, S., and Crespo, A. (2010). Xtratum for leon3: an open source hypervisor for high integrity systems. *Embedded and Real-Time Software and Systems.*

Masrur, A., Drossler, S., Pfeuffer, T., and Chakraborty, S. (2010). Vm-based real-time services for automotive control applications. *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, pages 218–223.

Masrur, A., Pfeuffer, T., Geier, M., Drossler, S., and Chakraborty, S. (2011). Designing vm schedulers for embedded real-time applications. *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2011 Proceedings of the 9th International Conference on*, pages 29–38.

Miramond, B., Huck, E., Verdier, F., Benkhelifa, A., Granado, B., Lefebvre, T., Aïchouch, M., Prevotet, J. C., Oliva, Y., Chillet, D., and Pillement, S. (2009). OveRSoC: A Framework for the Exploration of RTOS for RSoC Platforms. *International Journal of Reconfigurable Computing*, 2009:1–22.

Mok, A. (1983). *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology.

Mollison, M. S. and Anderson, J. H. (2013). Bringing Theory Into Practice : A Userspace Library for Multicore Real-Time Scheduling. *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*.

Molnar, I. (2004). CONFIG_PREEMPT_REALTIME, Fully Preemptible Kernel, VP-2.6.9-rc4-mm1-T4. `http://lwn.net/Articles/105948/`.

OMG (2014). Object Managemet Group. http://www.omg.org.

Ongaro, D., Cox, A. L., and Rixner, S. (2008). Scheduling i/o in virtual machine monitors. *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 1–10.

Parmer, G. (2010). *Composite: A Component-Based Operating System for Predictible and Dependable Computing*. PhD thesis, Boston University.

Parmer, G., Wang, Q., Song, J., Hossain, T., and Wu, Y. Y. (2012). Hijack$^{\mathrm{COS}}_{\mathrm{Linux}}$: Practical, Predictable, and Efficient OS Co-Location using Linux. *Proceedings of the 14th Real-Time Linux Workshop*.

Pham, K. D., Jain, A., Cui, J., Fahmy, S., and Maskell, D. (2013). Microkernel hypervisor for a hybrid arm-fpga platform. *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pages 219–226.

Phan, L. T. X., Xu, M., Lee, J., Lee, I., and Sokolsky, O. (2013). Overhead-aware compositional analysis of real-time systems. *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 237–246.

Popek, G. J. and Goldberg, R. P. (1974). Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421.

POSIX (2014). POSIX user context API. http://pubs.opengroup.org/onlinepubs/009695399/basedefs/ucontext.h.html.

Ramachandran, M. (2013). Challenges in virtualizing real-time systems using kvm/qemu solution. *Proceedings of the 14th Real-Time Linux Workshop*.

Rose, R. (2013). ELC: SpaceX lessons learned. `http://lwn.net/Articles/540368/`. Embedded Linux Conference.

Ruocco, S. (2006). User-level fine-grained adaptive real-time scheduling via temporal reflection.

Sekiyama, T. (2012). KVM: x86: CPU isolation and direct interrupts handling by guests. `https://lkml.org/lkml/2012/6/28/30`.

Shin, I. and Lee, I. (2003). Periodic resource model for compositional real-time guarantees. *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 2–13.

Srinivasan, B., Pather, S., Hill, R., Ansari, F., and Niehaus, D. (1998). A firm real-time system implementation using commercial off-the-shelf hardware and free software. *Real-Time Technology and Applications Symposium, 1998. Proceedings. Fourth IEEE*, pages 112–119.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2008). *EMG Eclipse Modeling Framework*. Pearson.

Steinberg, U. and Kauer, B. (2010). NOVA: a microhypervisor-based secure virtualization architecture. *Proceedings of the 5th European conference on Computer systems, ser. EuroSys '10. ACM*.

Stoess, J. (2007). Towards effective user-controlled scheduling for microkernel-based systems. *SIGOPS Oper. Syst. Rev.*, 41(4):59–68.

Tavares, a., Carvalho, a., Rodrigues, P., Garcia, P., Gomes, T., Cabral, J., Cardoso, P., Montenegro, S., and Ekpanyapong, M. (2012). A customizable and ARINC 653 quasi-compliant hypervisor. *2012 IEEE International Conference on Industrial Technology*, pages 140–147.

van den Heuvel, M. M. H. P., Holenderski, M., Cools, W., Bril, R. J., and Lukkien, J. J. (2009). Virtual timers in hierarchical real-time systems. *Work-in-Progress (WiP) session of the 30th IEEE Real-time Systems Symposium (RTSS)*, pages 37–40.

Varanasi, P. and Heiser, G. (2011). Hardware-supported virtualization on ARM. *Proceedings of the Second Asia-Pacific Workshop on Systems*.

Xi, S., Wilson, J., Lu, C., and Gill, C. (2011). RT-Xen: Towards Real-time Hypervisor Scheduling in Xen. *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 39–48.

Yang, J., Kim, H., Park, S., Hong, C., and Shin, I. (2011). Implementation of compositional scheduling framework on virtualization. *ACM SIGBED Review*, 8(1):30–37.

Yoo, S. and Yoo, C. (2013). Real-time Scheduling for Xen-ARM Virtual Machines. *IEEE Transactions on Mobile Computing*, pages 1–1.

Zhang, B., Wang, X., Lai, R., Yang, L., Wang, Z., Luo, Y., and Li, X. (2010a). *Evaluating and Optimizing I/O Virtualization in Kernel-based Virtual Machine (KVM).*, volume 6289 of *Lecture Notes in Computer Science*. Springer.

Zhang, J., Chen, K., Zuo, B., Ma, R., Dong, Y., and Guan, H. (2010b). Performance analysis towards a KVM-Based embedded real-time virtualization architecture. *5th International Conference on Computer Sciences and Convergence Information Technology*, pages 421–426.

Zijlstra, P. (2008). Linux-rt: Turning a general purpose os into a real-time os. `http://www.test.org/doe/`. Keynote, Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications.

Zuo, B., Chen, K., Liang, A., Guan, H., Zhang, J., Ma, R., and Yang, H. (2010). Performance Tuning Towards a KVM-Based Low Latency Virtualization System. *2010 2nd International Conference on Information Engineering and Computer Science*, pages 1–4.

INSA de RENNES
Service des Formations

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Titre de la thèse:
Evaluation of a mixed-critically real-time virtual machine system and configuration of an RTO's resources allocation techniques

Nom Prénom de l'auteur : AICHOUCH MOHAMED EL MEHDI

Membres du jury :
- Monsieur Pautet Laurent
- Monsieur PREVOTET Jean-Christophe
- Madame NOUVEL Fabienne
- Monsieur VERDIER François
- Madame Puaut Isabelle
- Monsieur Béchennec Jean-Luc

Président du jury : Isabelle Puaut

Date de la soutenance : 28 Mai 2014

Reproduction de la these soutenue

☞ Thèse pouvant être reproduite en l'état
~~Thèse pouvant être reproduite après corrections suggérées~~

Fait à Rennes, le 28 Mai 2014

Signature du président de jury

Le Directeur,

M'hamed DRISSI

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES - RENNES -

Résumé

L'utilisation de la virtualisation dans le domaine des serveurs d'entreprise est aujourd'hui une méthode courante. La virtualisation est une technique qui permet de faire fonctionner sur une seule machine réelle plusieurs systèmes d'exploitation.

Cette technique est train d'être adoptée dans le développement des systèmes embarqués suite à la disponibilité de nouveaux processeurs classiquement destiné à ce domaine.

Cependant, il y a une différence de contraintes entre les applications d'entreprise et les applications embarquées, celle-ci doivent respecter des contraintes de temps-réel en réalisant leurs tâches.

Dans nos travaux de recherche nous avons étudié l'impact de la virtualisation sur un système d'exploitation temps-réel. Nous avons mesuré le surcoût et la latence des fonctions internes du système d'exploitation déployé sur une machine virtuelle, et nous les avons comparés à celles du système installé sur une machine réelle. Les résultats ont montré que ces métriques sont plus élevées lorsque la virtualisation est utilisée.

Notre analyse a révélé que la puce électronique doit inclure des mécanismes matériels qui assistent le logiciel de contrôle des machines virtuelles afin de réduire le surcoût de la virtualisation, mais il est aussi essentiel de choisir une politique d'allocation des ressources efficace afin de garantir le respect des contraintes de temps-réel demandées par les machines virtuelles.

Notre second axe de recherche concerne la transformation d'un modèle de simulation d'un système d'exploitation vers des programmes exécutables sur un système-sur-puce. Cette transformation doit également préserver une caractéristique offerte par ce modèle qui est la facilité de configuration des techniques d'allocation de ressources.

Pour transformer le modèle de système d'exploitation nous avons utilisé des techniques de l'ingénierie-dirigée par les modèles. Où dans un premier temps le modèle initiale est transformé vers un autre modèle, ensuite ce second modèle est à son tour transformé automatiquement en un code source.

Pour assurer la configuration du système d'exploitation finale nous avons utilisé une librairie placée entre le système d'exploitation et l'application afin d'identifier les besoins de celle-ci en termes de ressources et adapter le système à ces besoins. L'évaluation des performances de la librairie a démontré la viabilité de l'approche.

Abstract

In the domain of server and mainframe systems, virtualizing a computing system's physical resources to achieve improved sharing and utilization has been well established for decades.

Full virtualization of all system resources makes it possible to run multiple guest operating systems on a single physical platform. Recently, the availability of full virtualization on physical platforms that target embedded systems creates new use-cases in the domain of real-time embedded systems.

In this dissertation we use an existing "virtual machines monitor" to evaluate the performance of a real-time operating system. We observed that the virtual machine monitor affects the internal overheads and latencies of the guest OS.

Our analysis revealed that the hardware mechanisms that allow a virtual machine monitor to provide an efficient way to virtualize the processor, the memory management unit, and the input/output devices, are necessary to limit the overhead of the virtualization. More importantly, the scheduling of virtual machines by the VMM is essential to guarantee the temporal constraints of the system and have to be configured carefully.

In a second work and starting from a previous project aiming at allowing a system designer to explore a software-hardware co-design of a solution using high-level simulation models, we proposed a methodology that allows the transformation of a simulation model into a binary executable on a physical platform.

The idea is to provide the system designer with the necessary tools to rapidly explore the design space and validate it, and then to generate a configuration that could be used directly on top of a physical platform.

We used a model-driven engineering approach to perform a model-to-model transformation to convert the simulation model into an executable model. And we used a middleware able to support a variety of the resources allocation techniques in order to implement the configuration previously selected by the system designer at simulation phase. We proposed a prototype that implements our methodology and validate our concepts. The results of the experiments confirmed the viability of this approach.

N° d'ordre : 14ISAR11 / D14-11

Institut National des Sciences Appliquées de Rennes

20, Avenue des Buttes de Coëmes • CS 70839 • F-35708 Rennes Cedex

Tel : 02 23 23 82 00 - Fax : 02 23 23 83 96