UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

A thesis submitted to the
Faculty of Computer Science,
Electrical Engineering and Mathematics
of the
University of Paderborn
in partial fulfillment
of the requirements for the degree of Dr. rer. nat.

submitted by

Dipl.-Inform. Timo Kerstan

Paderborn, 21. März 2011

TOWARDS FULL VIRTUALIZATION OF EMBEDDED REAL-TIME SYSTEMS

Supervisor____
Prof. Dr. rer. nat. Franz Josef Rammig

Referees____
Prof. Dr. rer. nat. Franz Josef Rammig
Prof. Dr. rer. nat. Marco Platzner

ABSTRACT

The growing complexity and the need for high level functionality in embedded hard real-time systems lead to conflicting goals for the design of the underlying system software. Adding the required high level functionality endangers the properties of being small, robust, efficient, safe and secure typically stated for embedded real-time operating systems. On the other side, the capability of handling hard real-time loads in a general purpose OS is often not compatible due to strong non determinisms in general purpose OSes. Top of the line cars already house more than 70 interconnected embedded control units (ECUs). The question is how to cope with this additional complexity. Will the next generation of cars house even more ECUs or will there be powerful general purpose ECUs executing more than one dedicated task to reduce the number of ECUs, as the growing number of ECUs is not adequately manageable any more? When looking towards the direction of using more powerful ECUs, the conflicting goals of general purpose OSes and real-time OSes arise again. Is it suitable to apply the paradigm of virtualization to realize this integration, still ensuring the requirements for both types of OSes? What virtualization paradigm is suitable for the domain of embedded real-time systems? How to guarantee real-time properties when applying virtualization to a real-time system with other real-time system or general purpose systems? All these questions need to be answered when thinking of virtualization as an upcoming paradigm for designing complex distributed embedded hard real-time systems.

ZUSAMMENFASSUNG

Die zunehmende Komplexität und die Forderung nach Schnittstellen auf höchster Ebene bei eingebetteten harten Echtzeitsystemen führt zu gegensätzlichen Zielen bei der Entwicklung der unterliegenden Systemsoftware. Das Hinzufügen weiterer Funktionalität auf höchster Ebene gefährdet die typischen Eigenschaften eingebetteter harter Echtzeitsysteme. Auf der anderen Seite ist die Implementierung von Funktionalität zum Ausführen von Systemen unter harter Echtzeit in nicht echtzeitfähigen Betriebssystemen nicht möglich, da diese aufgrund ihrer vorhandenen Implementierung oft nicht deterministisches Verhalten aufweisen. Heutige Oberklassenfahrzeuge enthalten allerdings mittlerweile mehr als 70 eingebettete Steuereinheiten (ECUs). Es stellt sich also die Frage, wie man dieser zunehmenden Komplexität Herr wird. Werden aus diesem Grund zukünftige Fahrzeuge noch mehr ECUs enthalten oder werden leistungsfähigere ECUs die Funktionalitäten mehrere ECUs in sich vereinen, weil die wachsende Zahl von ECUs nicht mehr adäquat handhabbar ist. Blickt man in Richtung des Einsatzes leistungsfähigerer ECUs, so tauchen die Konflikte der Designziele von eingebetteten harten Echtzeitsystemen und Betriebssystemen mit Funktionalität auf höchster Ebene wieder auf. Denkt man an den Einsatz von Virtualisierung, ergeben sich daraus interessante Fragestellungen. Ist der Einsatz von Virtualisierung in der Lage, die widersprüchlichen Ziele in ein System zu integrieren, wobei die Anforderungen beider Systeme noch immer erhalten bleiben? Eine weitere Frage ist, welches Paradigma der Virtualisierung am besten für eingebettete Echtzeitsysteme geeignet ist. Sämtliche Fragen bedürfen einer Antwort, wenn Virtualisierung als mögliches Lösungsparadigma in Betracht gezogen wird, um dem Design komplexer verteilter eingebetteter harter Echtzeitsysteme Herr zu werden.

Auf der Familie ruht die Kunst, die Wissenschaft, der menschliche Fortschritt, der Staat.

—Adalbert Stifter *23.10.1805 - †28.01.1868

To my children, my wife, my parents and my parents in law.

OWN PUBLICATIONS

[BK09] Baldin, Daniel and Timo Kerstan: *Proteus, a hybrid virtualization platform for embedded systems*. In Rettberg, Achim and Franz Josef Rammig (editors): *Analysis, Architectures and Modelling of Embedded Systems*. IFIP WG 10.5, Springer-Verlag, September 2009.

[GK08] Groesbrink, Stefan and Timo Kerstan: *Modular paging with dynamic tlb partitioning for embedded real-time systems*. In *SIES '08. Third International Symposium on Industrial Embedded Systems*, La Grande Motte, France, 2008.

[KBG10] Kerstan, Timo, Daniel Baldin, and Stefan Groesbrink: *Full virtualization of real-time systems by temporal partitioning*. In Petters, Stefan M. and Peter Zijlstra (editors): *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 24–32. ArtistDesign Network of Excellence on Embedded Systems Design, ArtistDesign Network of Excellence on Embedded Systems Design, July 2010. in conjunction with the 22nd Euromicro Intl Conference on Real-Time Systems Brussels, Belgium, July 7-9, 2010.

[KBS09] Kerstan, Timo, Daniel Baldin und Gunnar Schomaker: *Formale Bestimmung von Systemparametern zum transparenten Scheduling virtueller Maschinen unter Echtzeitbedingungen*. In: *Informatik aktuell (Tagungsband Echtzeit 2009)*. Fachausschuß Echtzeitsysteme der Gesellschaft für Informatik und der VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik (GMA), Springer-Verlag, November 2009.

[KDMK10] Klobedanz, Kay, Bertrand Defo, Wolfgang Müller, and Timo Kerstan: *Distributed coordination of task migration for fault-tolerant flexray networks*. In *Proceedings of the fifth IEEE Symposium on Industrial Embedded Systems (SIES2010)*. IEEE, IEEE, July 2010.

[KO10] Kerstan, Timo and Markus Oertel: *Design of a real-time optimized emulation method*. In *Proceedings of*

DATE 2010, Dresden, March 2010. IEEE Computer Society, IEEE Computer Society Press.

[RDJ$^+$09] Rammig, Franz Josef, Michael Ditze, Peter Janacik, Tales Heimfarth, Timo Kerstan, Simon Oberthür, and Katharina Stahl: *Hardware-dependent Software Principles and Practice*, chapter Basic Concepts of Real Time Operating Systems, pages 15–45. Springer, January 2009.

[SBTKS09] Samara, Sufyan, Fahad Bin Tariq, Timo Kerstan, and Katharina Stahl: *Applications adaptable execution path for operating system services on a distributed reconfigurable system on chip*. In *Proceedings of International Conference on Embedded Software and Systems, 2009. ICESS '09*, May 2009.

[SHK$^+$10] Schäfer, Wilhelm, Christian Henke, Lydia Kaiser, Timo Kerstan, Matthias Tichy, Jan Rieke und Tobias Eckardt: *Der Softwareentwurf im Entwicklungsprozess mechatronischer Systeme*. In: Gausemeier, Jürgen, Franz Josef Rammig, Wilhelm Schäfer und Ansgar Trächtler (Herausgeber): *7. Paderborner Workshop Entwurf mechatronischer Systeme*. Heinz Nixdorf Institut, HNI Verlagsschriftenreihe, Paderborn, März 2010.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Contents

Virtualization has been a key technology in the desktop and server market for a fairly long time. Numerous products offer hardware virtualization at the bare metal level or at the host level. They enable system administrators to consolidate whole server farms and end users to use different operating systems concurrently. In the case of server consolidation, virtualization helps to improve the utilization or load balance which facilitates a reduction of costs and energy consumption. Distributed embedded systems used in automotive and aeronautical systems consist of multitudinous microcontrollers, each executing a dedicated task to guarantee isolation and to prevent a fault from spreading over the whole network. In addition, the utilization of a single microcontroller may be very low. Applying virtualization to distributed embedded systems can help to increase the scalability while preserving the required isolation, safety, and reliability. It is not possible to apply the virtualization solutions for server and desktop systems one-to-one to embedded systems. The inherent timing constraints of embedded systems preclude this. These timing constraints add temporal isolation as a requirement to virtualized embedded real-time systems. Especially the classical approach of schedulability analysis [LL73] is no longer applicable to virtualized environments.

One of the main problems of building complex embedded systems is the integration of software components to a big integrated system.The automotive domain can be use as an example for a complex embedded system. When considering a top line car, there are about 70 ECUs[1] installed with different tasks, from hard real-time tasks for actuating elements to soft real-time for multimedia devices to non real-time elements like the ECU for the window lifter. Those traditionally quit unrelated tasks start to interact in such complex embedded systems, and the problem of unintentional feature interaction becomes an extremely important issue to be handled safely as the verification of a complex integrated system is an extremely hard task. Thus, there is the trend to have much less ECUs in favor of more centralized multi-functional multipurpose hardware, less communication lines and less dedicated sensors and actuators. The trend towards more centralized multi-functional hardware boosts the problem of unintentional interaction of software components as they share the processor, memory and I/O devices in this case. Thus, the task of the system software is to prevent unintentional interactions which are not based on the communication between the components such as the domination of hardware resources like the processor or memory, faulty implementations allowing for buffer overflows, heap overflows, stack overflows, race conditions and so on, as these unintentional interactions endanger all components running on the same hardware. This is a typical task for system software and is normally covered by using virtual memory isolating the tasks from each other, but there is a new demand for high-level functionality in such complex embedded systems. Reconsidering the automotive domain, one can easily see that the modern multimedia, infotainment and other comfort functions require a lot of high-level APIs to like sophisticated GUI libraries. This is typically not a task of embedded RTOSs[2]. This is the point where virtualization can show its strength as virtualization allows to isolate the high-level tasks into a VM[3] together with a GPOS[4] providing all of the high-level functionality needed by the cognitive, while isolating the hard real-time tasks in RTVMs[5], which are executing an RTOS. Thus, virtualization helps to simplify the integration process of compo-

1 Electronic Control Unit (ECU) is a generic term for any embedded system that controls one or more of the electrical systems or subsystems in a motor vehicle.

2 Real-Time Operating Systems (RTOS) are a class of operating systems being used in computing systems that must react within precise time constraints to events in the environment [Buto4]

3 Virtual Machines (VMs) are containers that provide subsets of the underlying hardware to the guest operating systems executed in this container

4 General Purpose Operating Systems (GPOSs) are the class of operating systems typically used on desktop computers

5 Real-Time Virtual Machines

nents with different requirements to their operating systems as it allows to run multiple operating systems while spatially isolating them and preventing unintentional interactions like resource domination or attacks resulting from faulty implementations. A big advantage of virtualization against the use of a single operating system providing all functionality is that VMMs are by design very small and are easier to verify than a big operating system full of high-level functionality.

1.1 PURPOSE OF THE THESIS

The growing complexity of embedded real-time systems and their demand for high-level functionality typically provided by GPOSs like Linux, Windows and Mac OS X is the main motivation of this thesis. This growing complexity of distributed embedded real-time systems needs to be handled. Considering a top of the line car, one can observe that there are more than 70 ECUs used for the realization of safety and comfort functions. So will there be more ECUs used due to the growing complexity in future? Prof. Dr. Manfred Broy of the Technical University Munich, who is a leading scientist in the domain of software engineering, doubts this trend and formulated the following hypothesis:

"The car of the future will certainly have much less ECUs in favor of more centralized multi-functional multipurpose hardware, less communication lines and less dedicated sensors and actuators. Arriving today at more than 70 ECUs in a car, the further development will rather go back to a small number of ECUs by keeping only a few dedicated ECUs for highly critical functions and combining other functions into a small number of ECUs, which then would be rather not special purpose ECUs, but very close to general-purpose processors. Such a radically changed hardware would allow for quite different techniques and methodologies in software engineering." [Bro06]

The requirement for High-Level API functionality for comfort functions and hard real-time capabilities for safety functions is not addressed in state of the art real-time operating systems, as their focus is to be small and efficient while general purpose operating systems are not able to handle hard real-time loads, besides the demand for High-Level APIs. As both domains have different requirements virtualization is a promising approach for integrating both kinds of operating systems into a single virtualized system. The purpose of this thesis is to provide a virtualization environment being capable of handling multiple real-time guests together with multiple general purpose guests, while preventing the necessity of paravirtalization. The virtualization platform shall be able to support paravirtualization, but it shall not require it, as licensing restriction may eliminate the possibility of paravirtualizing a specific given operating system.

1.2 CURRENT STATUS

Up to the point in time where the work on this thesis started there, was no available virtualization platform providing the support for paravirtualization and full virtualization to address the problem of providing a hybrid virtualization interface being tailored to the applications needs. The trend towards virtualization has raised the interest of industry, and some products offering full virtualization for High-Level guests and paravirtualization for real-time guests have been released on the market in the meantime. Unfortunately, the support for multiple real-time VMs has been covered sparely in industry by restricting the execution of real-time virtual machines either to a dedicated cpu core or by restricting the number of real-time virtual machines to only one with an arbitrary number of non real-time virtual machines. Academia in contrast offers different hierarchical scheduling approaches especially suited for paravirtualization that allow for the presence of multiple real-time virtual machines, but there is also a lack of approaches using full virtualization.

1.3 STRUCTURE OF THE THESIS

To provide a general common understanding of the underlying real-time and virtualization techniques, chapter 2 will provide a short overview of basic real-time and virtualization terms and techniques. Chapter 3 will clearly define the problems being addressed in detail within this thesis. The problems being defined will be covered in chapter 3 and 4. The problem of providing a configurable hybrid virtualization interface will be covered in chapter 3, and an overview of the corresponding related work is given in the beginning of this chapter. Afterwards, the design of such a virtualization platform is presented and evaluated. In addition, the worst case execution times are determined in order to provide the possibility to determine the deterministic overhead induced by the virtualization. Finally, chapter 4 covers the problem of deriving the cpu requirements and feasible schedule of given real-time virtual machines in order to guarantee their execution in a full virtualized environment to eliminate the necessity of paravirtualization. Therefore, the related work relevant for this topic is presented in the beginning of this chapter, before these problems are modeled and a solution is presented. The final step of the thesis is an evaluation based on a real execution on a PowerPC hardware platform. Within this evaluation, the approach presented in this thesis will be compared to the state of the art approach and a final resume on the evaluation is given. To complete this thesis, a final conclusion is given summing up the reached goals and discussing the possible future work.

2 VIRTUALIZING EMBEDDED REAL-TIME SYSTEMS

Contents

Today's computer systems are extremely complex and are designed as hierarchies with well-defined interfaces that separate levels of abstraction. This allows the independent development of subsystems by hardware and software design teams. Low-level implementation details are hidden by the simplifying abstractions. In contrast to abstraction, virtualization does not necessarily hide or simplify details. Instead, virtualization provides different resources at the same abstraction level. A simple example is providing two virtual network adapters while having only one physically available.

System virtualization has become a key technology in the enterprise and personal computing spaces and is recently gaining significant interest in the domain of embedded systems [Hei08b, Hei07]. After introducing the key characteristics of enterprise and embedded systems, the difference in motivation for the use of virtualization and the resulting differences in requirements will be presented. These requirements will be used to identify the flaws of current virtualization technologies in the context of embedded hard real-time systems under the assumption that the software of the embedded systems is not modified to support virtualization as this may be prohibited by licensing restrictions.

When looking at modern data centers today, virtualization is a hot topic. The decoupling of physical and virtual execution platforms by System VMs enables a variety of aspects which are of importance for enterprise data centers.

- Service consolidation: Services being executed on single machines can be integrated into a single virtualized system using system virtualization to reach a better utilization of the system.
- Load balancing: Through the decoupling of physical and virtual execution platform, it is possible to migrate VMs between different virtualization hosts depending on their load.
- Heterogeneity: The use of System VMs enables the execution of different operating systems (OSs) on a single machine. This is mostly relevant for personal desktops.
- Power management: This is closely related to load balancing, but with the optimization goal to minimize the power consumption of the data center.
- Spatial Isolation: The criticality of services may differ extremely, so in general it is necessary to isolate services from the rest of the system not allowing them to compromise the whole system when they are compromised or fail.

The main characteristic of these aspects is the fact that the VMs execute GPOSs, providing roughly the same kind of of capabilities and similar abstraction levels. Another characteristic of those scenarios is that VM communication is closely related to the communication of physical machines using network interfaces [Hei07, Hei08b, SN05a, SN05b].

The characteristics of embedded systems have changed dramatically over the last two decades. Especially the complexity of embedded systems has increased tremendously. They changed from relatively simple single purpose devices to extremely complex distributed systems with millions lines of code (Mloc). Top of the line cars consist of approximately 70 embedded control units (ECUs) and gigabytes of software. A funny rumor is that it takes longer to download the software than building the vehicle physically.

When taking a look at modern smart phones like the Apple iPhone, a new characteristic of modern embedded systems can be noticed. Increasingly, embedded systems run applications originally developed for GPOSs like Linux, Windows and Mac OS

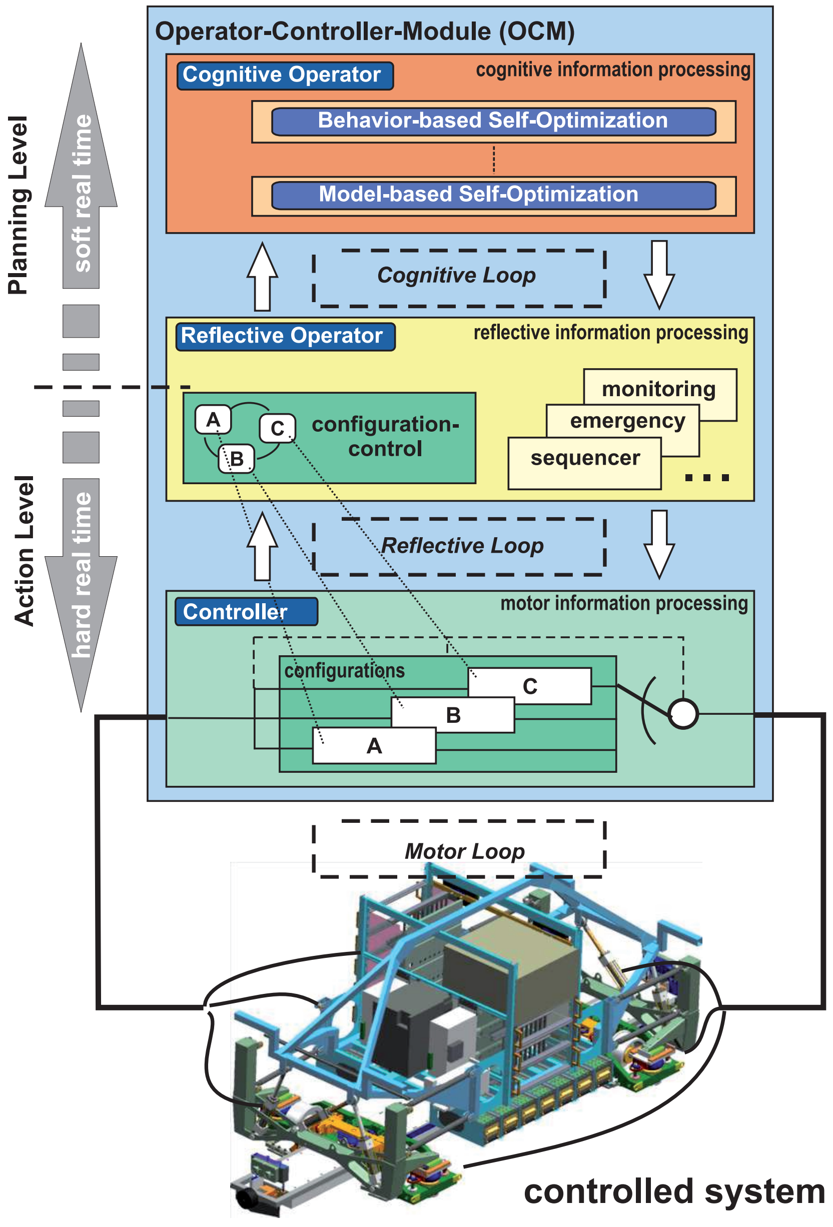
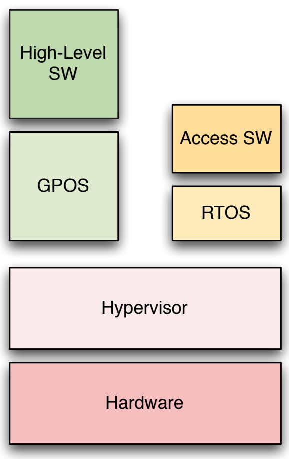


Figure 1: Operator Controller Module

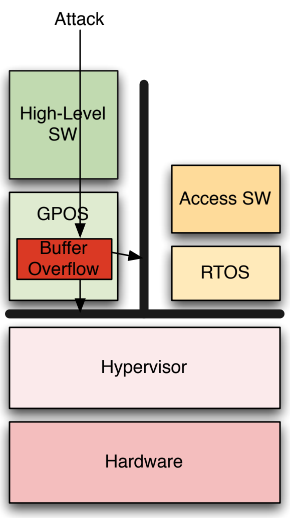
X. There is also a trend that programers develop programs for embedded system platforms without any experience in this area. Furthermore there is a strong trend towards openness. The owners of a device want to load their own applications on the embedded systems and run them there. To enable such an openness, it is necessary to provide an open API introducing all the security challenges known from GPOSs. However, embedded systems are still subject to real-time and resource constraints. In addition they are often used in safety critical mechatronical systems like planes, cars and trains leading to very high requirements on safety, reliability and security. The OCM[1] structure (depicted in figure 1) of the CRC 614[2] is an example for such a safety critical system where the concurrent demands of safety critical systems and the need of High-Level APIs exist. [But06, Hei07, Hei08b]. The OCM is divided into 3 layers:

1. Controller: A closed loop system that controls the sensors and actuators of the mechatronical system. The sensing, calculation of the control signals and the output of the control signal need to be performed in hard real-time.
2. Reflective operator: A monitoring and controlling layer above the controller. It has no direct access to the hardware, but modifies the controller by parameter or structure modifications. Typically, the reflective operator is time-critical in terms of soft real-time, but it may also operate in hard real-time.
3. Cognitive operator: The top layer of the OCM is responsible for using the information of the reflective operator as input for its cognition to perform self-optimization of the whole mechatronical system. This is realized by using a diversity of methods like machine learning, model based optimization or knowledge based systems. The application of these methods demand the use of High-Level APIs implementing those methods. The cognitive operator is not time critical and thus does not need to be executed under real-time constraints. Furthermore the programers at this layer do not need necessarily the knowledge of programing embedded systems.

1 Operator Controller Module
2 Collaborative Research Center for Self-Optimizing Mechatronical Systems



(a) Heterogenous OS environments



(b) Security

Figure 2: Use Cases for the application of virtualization to embedded systems[Hei08b]

The new trend for GPOS properties and the demands of complex embedded real-time systems like the self-optimizing mechatronical systems developed within the CRC 614 make virtualization a very promising technique in this area, as virtualization provides the following interesting aspects:

- Heterogeneity: The use of System VMs enables the execution of different operating systems (OSs) on a single embedded system, as depicted in figure 2a to address the conflicting requirements of high-level APIs, real-time support and legacy support.
- Spatial isolation: When building heterogenous OS environments on a single embedded system it is of indispensable importance to spatially isolate the different OSs from each other in a manner that it is not possible for a fault or an attack to spread to the other systems as depicted in figure 2b. Spatial isolation is typically realized by assigning different virtual address spaces to the different VMs. Virtualization fulfills this requirement and therefore increases the security and dependability significantly.

- Architectural abstraction: The decoupling of physical and virtual execution platform enables the possibility to abstract from the instruction set architecture (ISA). This results in the possibility to migrate VMs unchanged to hosts with the same ISA and to hosts with a different ISA using emulation. The same ISA migration covers especially the case of distributing VMs on multiple processor systems on chip (MPSoC) which are also an upcoming trend in embedded systems.
- Legacy software support: In contrast to personal desktops and enterprise systems, embedded systems are built on a broad diversity of different μCs[3]. This diversity implies the existence of different ISAs which makes it necessary to port embedded system software to run on a different ISA when changing the μC of an embedded system. This becomes problematic if the source code of this software is not available or accessible due to licensing restrictions. The advantage of virtualization now comes into play as virtualization allows the usage of emulation techniques to make this software run on a non supported ISA.
- Service consolidation: Services being executed on single ECUs can be integrated into a single virtualized system using system virtualization to lower the complexity of large distributed embedded systems.

These aspects are closely related to the aspects of enterprise system virtualization. However, the inherent timing requirements of embedded systems makes the application of existing virtualization techniques difficult as they have been developed for enterprise system architectures. Temporal isolation is an essential requirement of RTVMs. This means they must not be interfered by other VMs to respect their own timing behavior. Specifically, there is a temporal isolation among VMs whenever the ability for a VM to respect its own timing constraints (e.g. terminating a computation within a specified time, a.k.a. deadline) does not depend on the temporal behavior of other unrelated VMs running on the same system, thus sharing with it a set of resources (e.g. the CPU or such devices as disk, network, etc...). Understanding the issues of temporal isolation requires a fundamental understanding of real-time systems and their parameters. Sec-

3 micro controllers

tion 2.1 clarifies the common notations and algorithms used for preserving the timing requirements of real-time systems.

The next essential requirement for virtualizing embedded real-time systems is the architecture of VMs enabling the spatial isolation being introduced in section 2.2. Especially the architecture of the software controlling virtual machine is addressed in this section. Besides the architecture of the VMs a brief introduction on common implementation methods for processor virtualization and memory virtualization will be given to show the issues of implementing VMs on a specific hardware.

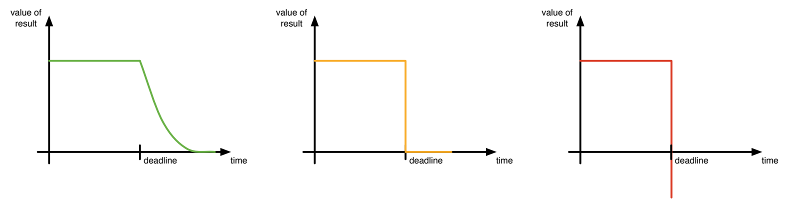
2.1 REAL-TIME SYSTEMS

In the introduction of this chapter, the term "real-time" was already used without clearly defining it. Most people think of real-time systems being extremely fast and performing without any noticeable lags. This understanding of real-time systems does not conform to the definition of real-time systems given by Kopetz:

Definition 2.1. *A real-time Computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computation, but also on the physical instant at which these results are produced.*

The definition implies that the value of a logical result depends on the time it is produced. Thus, a real-time operating system (RTOS) needs to be able to manage tasks with timing constraints. In addition, the RTOS may need to ensure the timing constraints of a task in the peak load (worst-case) situation. When this property is of indispensable importance, the real-time system needs to be entirely predictable to be able to determine the worst-case situation. Thus, a real-time system does not only need to be fast to fulfill the timing requirements, it especially has to be predictable to be able to determine whether it is possible to fulfill the timing constraints even in the worst-case situation. In a nutshell, a real-time system has to fulfill these properties:

- Timeliness
  - OS has to provide kernel mechanisms for time management



(a) soft real-time **(b)** firm real-time **(c)** hard real-time

Figure 3: Classes of real-time systems. [Buto4]

  - handling tasks with explicit time constraints
- Design for peak load
  - Predictability

The value of the logical result in real-time decreases at the time the deadline is reached. Imagine a video decoder who has to decode the video frames in time to guarantee lag free video display. When the video decoder exceeds the deadline, the result is not completely useless. The decoded frame can still be displayed when the deadline is exceeded within a pre-defined time interval to prevent the frame from being dropped. This leads to video jitter, but the video may still be viewed at an acceptable quality. This application is an example for the class of *soft real-time systems* where the value of the computed result decreases smoothly after the deadline is exceeded (see figure 3a). The second class of real-time systems is called *firm real-time systems*. In a firm real-time system, the value of the computed result is zero when the deadline is exceeded (see figure 3b). Examples for such a systems are decision support and value prediction systems such as stock exchange systems and weather forecast systems. The third and final class of real-time systems is called *hard real-time systems*. In hard real-time systems, the value of the computed result is negative when the deadline is exceeded (see figure 3b), because missing the deadline causes catastrophic damage to the controlled system. Consider an ABS[4] in a vehicle where the real-time task misses to compute the force control value of the brakes. Typical hard real-time activities are sensory data acquisition, actuator servoing and low-level control of critical system components. [Buto4]

4 Anti Lock-Braking System

2.1.1 Task Models

Real-time systems in general execute a set of so called real-time tasks under the constraint of fulfilling the time constraint of each real-time task. Tasks within a real-time system are characterized by computational activities within stringent timing constraints that must be met in order to achieve the desired behavior. A typical timing constraint on a task is the deadline. If a deadline is specified with respect to the arrival time, it is called a *relative deadline*, whereas if it is specified with respect to time zero, it is called an *absolute deadline*.

In general, a real-time task can be characterized by the following parameters (see figure 4 for a graphical description):

- Arrival time a_i: is the time at which a task becomes ready for execution; it is also referred to as *request time* (or release time) and indicated by r_i;
- Computation time C_i: is the time necessary to the processor for executing the task without interruption;
- Absolute deadline d_i: is the time before which a task should be completed to avoid damage (if hard) or performance degradation (if soft);
- Relative deadline D_i: is the difference between the absolute deadline and the arrival time: $D_i = d_i - r_i$;
- Start time s_i: is the time at which a task starts activation;
- Finishing time f_i: is the time at which the task finishes its execution;
- Response time R_i: is the difference between finishing time and the arrival time: $R_i = f_i - a_i$;
- Criticality: is a parameter related to the consequences of missing the deadline (typically, it can be hard or soft);
- Value v_i: represents the relative importance of the task with respect to other tasks in the system;
- Lateness L_i: $L_i = f_i - d_i$ represents the delay of a task completion with respect to its deadline;
- Tardiness or *Exceeding Time* E_i: $E_i = max(0, L_i)$ is the time a task stays active after its deadline;

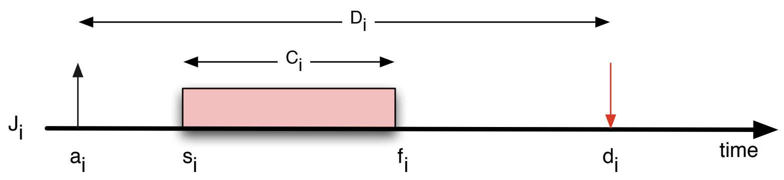


Figure 4: Common parameters of real-time tasks [Buto4].

- Laxity or *Slack time* X_i: $X_i = d_i - a_i - C_i$ is the maximum time frame a task can be delayed on its activation to complete within its deadline;

In real-time systems there are different classes of real-time tasks, namely *periodic tasks* (see figure 5a), *aperiodic tasks* (see figure 5b) and *sporadic tasks*. The exact definition of these classes is now given to clearly distinguish them from each other:

Definition 2.2. *A periodic task τ_i has an infinite sequence of identical activities, called instances or jobs, and is regularly activated at a constant rate. The activation time of the first periodic instance is called* phase ϕ_i. *If ϕ_i is the phase of the periodic task τ_i, the activation time of the k-th instance instance is given by $\phi_i + (k-1) \cdot T_i$, where T_i is called* period *of the task. [Buto4]*

Definition 2.3. *An aperiodic task J_i has an infinite sequence of identical activities, called instances or jobs, and is not regularly activated at a constant rate [Buto4].*

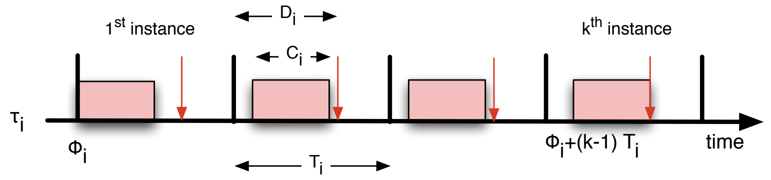
Definition 2.4. *A sporadic task is an aperiodic task where consecutive jobs are seperated by a minimum interarrival time [Buto4].*

2.1.2 Periodic task scheduling

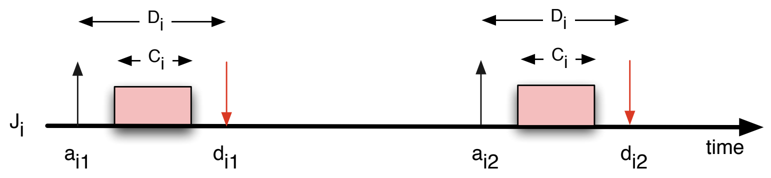
In many real-time control applications, like the controller module of the OCM described in the introduction of this chapter (see 2), periodic activities represent the major computational demand in the system. When a control application consists of n concurrent periodic tasks, building the taskset

$$\Gamma = \{\tau_i(\Phi_i, T_i, C_i) | i = 1...n\} \tag{2.1}$$

with $\tau_{i,j}$ being the j-th instance of task τ_i, and $a_{i,j}$ or $r_{i,j}$ being the release time of the j-th instance of task τ_i, the operating



(a) periodic task τ_i



(b) aperiodic task J_i

Figure 5: Classes of real-time tasks [Buto4].

system has to guarantee that each periodic instance is regularly activated at its proper rate and is completed within its deadline. Therefore, a brief introduction to timeline scheduling (TS) also known as cyclic scheduling (CS) or synchronous time division multiple access (TDMA), rate monotonic priority assignment (RM) and earliest deadline first scheduling policies are now given under the assumption of an independent taskset Γ and tasks τ_i having relative deadlines D_i equal to their period T_i.

Timeline scheduling

The main idea of timeline scheduling consists in dividing the temporal axis into slices of equal length in which one or more tasks can be allocated offline for execution, in such a way to respect the frequencies derived from application requirements. Figure 6 illustrates the timeline scheduling method for a taskset A, B and C with periods $T_A = 25ms$, $T_B = 50ms$ and $T_C = 100ms$. To meet the required periods, the GCD[5] of the periods can be used to determine the time slice length which is also called *minor cycle*. The minimum period after which the schedule repeats itself is called a *major cycle* or *hyperperiod* being in general equal to the least common multiplier (LCM) of the tasks periods. In the example shown in figure 6, the minor cycle is $25ms$ and

5 Greatest Common Divisor

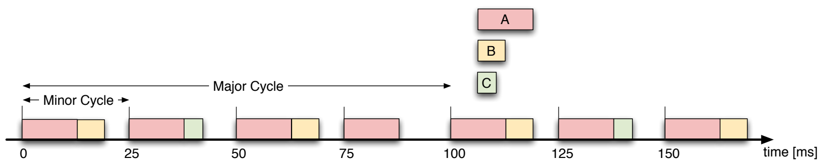


Figure 6: Example of timeline scheduling [Buto4].

the major cycle is $100ms$. Task A needs to be executed every minor cycle while Task B needs to be executed every two minor cycles and Task C needs to be executed every four minor cycles. A possible schedule is also depicted in figure 6.

Timeline scheduling is a very simplistic approach which can be easily implemented by programming a timer interrupt with a period equal to the minor cycle. Another advantage is that the tasks are not affected by jitter, because task start and response times are not subject to large variations. Nevertheless, timeline scheduling has some more drawbacks besides being an offline approach. Overhead condition handling is very problematic, because if a task does not terminate in time, it can either be aborted, leaving the system in an inconsistent state, or if the failing tasks continues execution, it can cause a domino effect on the other tasks, breaking the entire scheduler.

The timeline scheduling approach is similar to the *Time Division Multiple Access* multiplexing used in communications engineering.

Rate Monotonic Scheduling

The RM[6] scheduling algorithm, or more precisely the rate monotonic fixed priority assignment algorithm, is a simple algorithm assigning priorities to real-time tasks proportional to their execution rates or equivalently anti-proportional to their period T. Thus, tasks with a higher rate (shorter period) will receive higher priorities. As periods are not subject to change at runtime, RM is a fixed priority scheduling algorithm where priorities are assigned offline to all tasks. The scheduling itself is priority-based and intrinsically preemptive.

In 1973, Liu and Layland showed in [LL73] that RM is optimal among all fixed-priority assignments, in the sense that there is

6 Rate Monotonic

no other fixed priority algorithm that can schedule a taskset that cannot be scheduled by RM. Furthermore the least upper bound based on the processor utilization factor has been derived to test whether a taskset is schedulable by RM or not.

Definition 2.5. *The processor utilization factor* U *is the fraction of processor time spent on executing the tasks of a given taskset* Γ*. The time spent within a task* τ_i *is* $\frac{C_i}{T_i}$*. Thus the utilization factor* U *for* n *tasks is given by:*

$$U = \sum_{i=0}^{n} \frac{C_i}{T_i}$$

To decide whether a real-time taskset Γ is being schedulable by an arbitrary scheduling algorithm A, it has to be checked whether the processor utilization U is less or equal than the upper bound $U_{ub}(\Gamma, A)$, which depends on the taskset and the applied scheduling algorithm. In the case $U = U_{ub}(\Gamma, A)$ the processor is said to be fully utilized. To eliminate the dependency on the specific taskset to decide whether a taskset is schedulable by a scheduling algorithm A, the minimum of all upper bounds can be used as simplified schedulability check for a given taskset Γ. The minimum of all upper bounds is called *least upper bound* U_{lub}:

$$U_{lub}(A) = \min_{\Gamma}(U_{ub}(\Gamma, A)) \tag{2.2}$$

In the case of RM, the least upper bound $U_{lub}(n)$[7] of a taskset with n tasks can be calculated as:

$$U_{lub}(n) = n \cdot (2^{\frac{1}{n}} - 1) \tag{2.3}$$
$$\lim_{n\to\infty} U_{lub}(n) = \ln(2) \approx 0.69 = U_{lub}(RM) \tag{2.4}$$

Theorem 2.1. *Let* Γ *be an arbitrary set of periodic tasks with a processor utilization factor of* $U(\Gamma)$*. Then* Γ *is schedulable by RM if*

$$U(\Gamma) \leqslant U_{lub}(RM).$$

7 The derivation of $U_{lub}(n)$ can be found in [LL73] or [But04]

Theorem 2.2. *Let Γ be an arbitrary set of n periodic tasks with a processor utilization factor of $U(\Gamma)$. Then Γ is schedulable by RM if*

$$U(\Gamma) \leqslant n \cdot (2^{\frac{1}{n}} - 1).$$

Another approach in the same complexity class of $O(n)$ is the hyperbolic bound introduced by Bini et. al in [BB01]. The hyperbolic bound is less pessimistic than the original Liu and Layland bound.

Theorem 2.3. *Let Γ be an arbitrary set of n periodic tasks. Then Γ is schedulable by RM if*

$$\prod_{i=1}^{n} (\frac{C_i}{T_i} + 1) \leqslant 2.$$

All these schedulability tests presented up to now are sufficient tests. Audsley et. al presented in [Aud91] a sufficient and necessary schedulability test with complexity $O(n^2)$. This test is called *Response Time Analysis*. As the name implies, the test determines for every task the response time at the critical instant as the sum of its computation and the interference due to preemption by higher priority tasks.

Theorem 2.4. *Let Γ be an arbitrary set of n periodic tasks. Then Γ is schedulable by RM if and only if*

$$\begin{aligned} \forall R_i \leqslant D_i &: \\ \textit{with } R_i &= C_i + I_i, \\ \textit{where } I_i &= \sum_{j=1}^{i-1} \lceil \frac{R_j}{T_j} \rceil \cdot C_j \end{aligned}$$

As this is a recurrent equation, no simple solution exists. Thus, the test has to be performed by iteratively checking the equation for every task of the taskset Γ. The floor expression within the equation ensures the termination of the iteration when checking against the deadline of each task, as there is no asymptotic behavior.

Earliest Deadline First Scheduling

In contrast to RM, the EDF[8] scheduling algorithm is a dynamic scheduling algorithm that selects the next task to execute according to the absolute deadlines. Thus, tasks with earlier absolute deadlines have higher priorities and as the absolute deadline of a periodic tasks depends on the current instance the priorities of the periodic task change during execution. As RM, EDF is intrinsically preemptive, as it preempts the current running task when a task arrives with a shorter absolute deadline. EDF is not only able to handle periodic tasks, but also aperiodic tasks and is optimal in minimizing the maximum lateness on a given taskset.

To check whether a given taskset is schedulable by EDF the processor utilization factor introduced in definition 2.5 is used to check it against the least upper bound of EDF. Fortunately, the least upper bound of EDF is one, enabling a very simple schedulability test and allowing tasks to fully utilize the processor up to 100%.

Theorem 2.5. *Let Γ be an arbitrary set of periodic tasks, then Γ is schedulable by EDF if and only if*

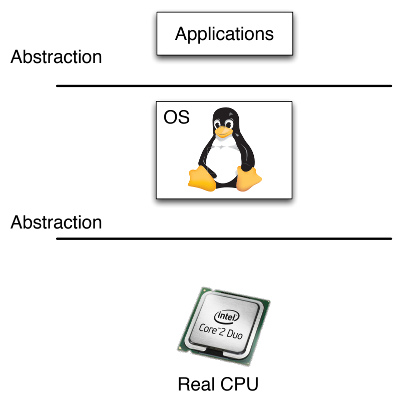
$$U(\Gamma) = \sum_{i=1}^{n} \frac{C_i}{T_i} \leqslant 1.$$

[But04]

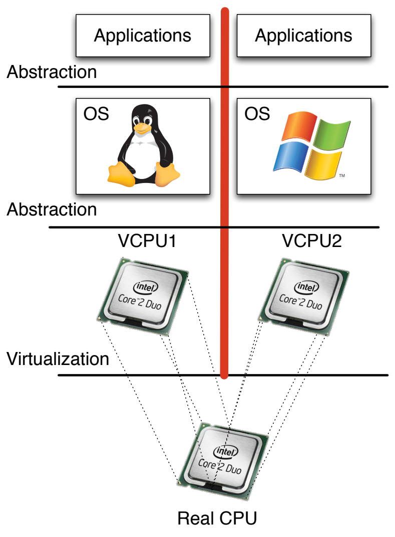
2.2 THE ARCHITECTURE OF VMS

Today's computer systems are extremely complex and are designed as hierarchies with well-defined interfaces that separate levels of abstraction. This allows the independent development of subsystems by hardware and software design teams. In addition low-level implementation details are hidden by the simplifying abstractions. In contrast to abstraction, virtualization does not necessarily hide or simplify details. This is depicted in figure 7. Figure 7a shows that the operating system is an abstraction of the hardware, in this case the CPU, and provides a simplified interface (Processes) to access the CPU, while figure 7b shows

8 Earliest Deadline First



(a) Abstraction



(b) Virtualization

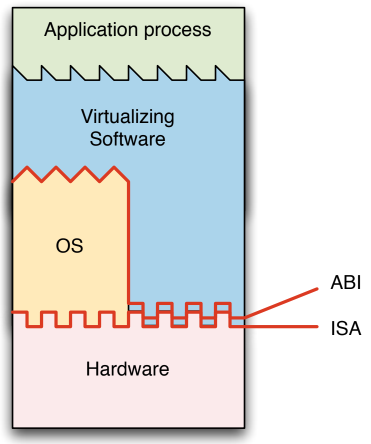
Figure 7: Abstraction vs. Virtualization

that virtualization provides different resources, in this case two virtual CPUs, at the same abstraction level.

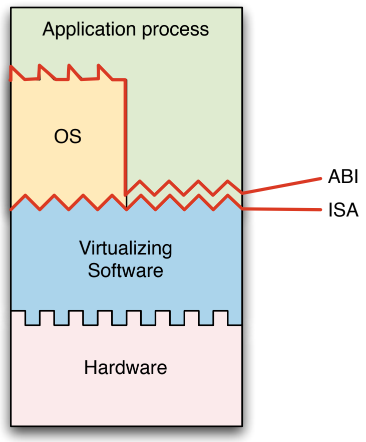
Virtualization can be applied at different interface levels of a computer system architecture. The Instruction Set Architecture (ISA) is the interface between soft- and hardware and is divided into user and system instructions. The system instructions are privileged instructions that are only accessible by software running in system mode to prevent user mode software from unauthorized access to the hardware.

A process virtual machine is executed in general on top of the OS and provides a uniform view to application processes independent of the underlying OS and hardware. A good example for this is the Java Virtual Machine (JVM). A system virtual machine is located directly at the ISA level and creates virtual instances of the underlying hardware which it can assign to different guest OSs.

In case of process VMs depicted in figure 8a, the operating system runs in system mode and the virtualizing software runs in user mode while in case of System VMs the virtualizing software depicted in 8b runs in system mode. So in case of system virtualization, the virtualizing software is put directly at the ISA level and is therefore in full control of the hardware, while in



(a) Process virtual machine



(b) System virtual machine

Figure 8: Process and System VMs

case of process VMs the virtualizing software is put in the ABI[9] level where the OS is in full control of the hardware. The ABI gives a program access to the hardware by the system call interface of the OS and direct access to the user mode ISA. Thus the difference of System VMs and process VMs is that a process VM is a virtual platform that executes an individual process while a System VM provides a complete persistent system environment that supports an operating system along with its many user processes. In addition, it provides access to virtual hardware resources like networking, I/O, memory and processors [BDF$^+$03, SN05a, SN05b].

Due to the demands of GPOS properties, openness, High-Level APIs, real-time behavior and strict spatial and temporal isolation, system virtualization is the most interesting VM architecture for embedded real-time systems, because a system virtual machine is able to provide a complete persistent system environment to host multiple OSs fulfilling these demands. These properties inspired the development of the system multiple independent layers of security architecture (MILS [Obj08]) for embedded systems being used in mission critical systems.

9 Application Binary Interface

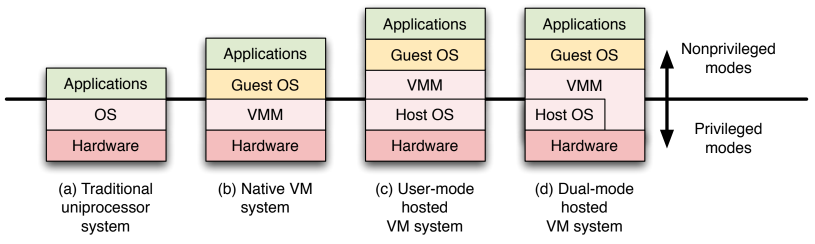


Figure 9: Different types of System VMs [SN05b].

2.2.1 System Virtual Machines

The VMM[10] is the core component in any System VM environment. It is responsible for scheduling and managing the allocation of hardware resources to various guest VMs. The VMM controls the physical resources and makes them available to the guest VMs by providing them as virtualized resources. The resources can be shared among the guest VMs, they can be partitioned with a partition being exclusively accessible to a guest VM or they can be exclusively assigned to a single guest VM. Such resources include the CPU registers, the real memory of the system and the various I/O devices attached to the system. To realize virtualization in an efficient manner, at least the VMM needs to be executed in an higher privileged "real CPU mode" than the code of the guest VMs. The real CPU mode denotes the CPU mode currently determined by the hardware. Usually, the hardware provides a system mode used by the OS and a user mode used by the applications. The real CPU mode can differ from the virtual CPU mode, as the CPU mode needs to be virtualized for the guest OS. Thus the guest OS executes virtually in a higher privileged CPU level than the guest applications, while the real CPU mode in hardware may not reflect this. With this knowledge, it is possible to distinguish between different types of System VMs.

Definition 2.6. *A virtual machine system in which the VMM operates in a privileged level higher than the level of the guest VMs is called* native VM system *(see figure 9) [SN05b].*

In such a system, the VMM has to be installed first on the system. The guest OSs are installed on top of the VMM. Thus, the

10 Virtual Machine Monitor

VMM always keeps full control over the hardware during the installation process. As the definition says the guest OS runs in a lower privilege level than the VMM, and thus the CPU privilege mode has to be emulated by the VMM. A well known example for native VM systems is XEN. Sometimes it is advantageous to run the VMM on top of a host OS for user convenience and implementation simplicity, as such a System VM can use the functionality provided by the host OS to implement the VMM functionality.

Definition 2.7. *A virtual machine system in which the VMM is executed on top of an existing OS is called* hosted VM system *(see figure 9) [SN05b].*

A hosted VM system can be implemented with the VMM running at a lower privilege level than the guest OS only when it is possible to modify the host OS. This is not always possible, as the source code may be unavailable or licensing restrictions prohibit such modifications. In these cases, the VMM may be implemented at the same privilege level of the guest OS.

Definition 2.8. *A virtual machine system in which the VMM is executed on top of an existing OS with a privilege level equal to the privilege level of its guests is called* user-mode hosted VM system *(see figure 9) [SN05b].*

User-mode System VMs suffer efficiency, because most of the code has to be scanned or emulated. To overcome these problems, an additional kind of hosted VM systems has been introduced.

Definition 2.9. *A virtual machine system in which the VMM is executed on top of an existing OS with a privilege level lower or equal to the privilege level of its guests is called* dual-mode hosted VM system *(see figure 9) [SN05b].*

Those systems can be implemented using well defined interfaces of the OS such as kernel extensions or device drivers. A well known example for such a dual-mode hosted system is the classic VMWare for Desktops.

To make the physical resources available to the guest VMs the VMM assigns these resources to the guest VM. It is very important for the VMM to be able to get the control of the resources back, so that they can be assigned to a different VM when the resource is shared between multiple VMs. Thus, the VMM must maintain the full control over all hardware resources, even in the case that they are temporarily assigned and used by the guest VM currently running. This problem already occurs in time-sharing system, where the OS needs to get the processor back to assign a new task to the processor. In this case. the resource controlled by the OS is the interval timer. It is not directly accessible for the tasks of the OS. Thus the OS can ensure its reactivation by setting a timer value equal to the time assigned to the task. After this time, the timer causes an interrupt guaranteeing that the OS gains back control of the processor. The situation is similar in a System VM environment. In this case the VMM controls the sharing of the resources between the different VMs. The VMM therefore emulates the access to privileged resources to prevent the VMs from directly accessing these resources. When considering interrupts as an example, the VMM would first handle the interrupt itself before it modifies the state of the guest VMs to emulate the incoming interrupt.

In the following section, the process of the control transfer from the guest VMs to the VMM will be formally derived to ensure virtualization with one of the main properties being to keep the VMM in full control of all system resources. Afterwards, the problems of virtualizing system memory will be introduced. [SN05b]

2.2.2 Processor Virtualization

Already in the very early stage of third generation computers, virtualization came up as a technology to realize multiple subsystems on a large system like a mainframe computer. In 1974, G.J. Popek and R.P. Goldberg presented the formal definitions of VMs and a simple condition which can be tested to determine whether an architecture can efficiently support virtualization. After presenting this formal requirements in the following subsection the different control transfer approaches are shortly introduced to pass the control from a VM to the VMM.

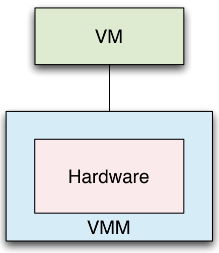


Figure 10: Virtual Machine Monitor Concept

Formal Requirements of Virtualizability

In the following section, it will be shown that system virtualization software needs to fulfill the three properties efficiency, resource control and equivalency to provide a correct and efficient virtualization of the underlying ISA. This will be described at a quite formal level which allows the exact classification of ISAs into two classes. The first class contains all ISAs fulfilling the formal requirements and are thus virtualizable, while the second class contains the ISAs not fulfilling these requirements. These ISAs are called to be not efficiently virtualizable.

To realize a virtual machine on top of the real machine, the concept of the virtual machine monitor (VMM), depicted in figure 10, is introduced. The three essential characteristics of a VMM are:

- The VMM provides an essentially identical environment for programs as the original machine would.
- Programs running in this environment show only minor decreases in speed.
- The VMM is in complete control of the system resources.

The first property is meant in terms of identical results when executing an arbitrary program with or without the existence of the VMM. *This does not cover identical in timing*. The second property demands for a statistically dominant subset of instructions of the virtual machine being executed natively. This rules out traditional emulators and software interpreters from the virtual machine umbrella. The third property ensures that an arbitrary program is not able to access any resource not allocated to it. Upon every attempt to access resources the VMM is invoked controlling the access to the resource. Now we can define what a virtual machine is:

Definition 2.10. *A virtual machine is the environment created by the VMM [PG74].*

To understand how the process of virtualization works, it is necessary to define a model of third generation architectures. The ISA provides two modes of execution: The user mode u and the system mode s. If the machine is in system mode, the complete set of instructions of the ISA is fully available while in user mode only a subset is available. A state of a third generation computer has four elements: executable storage E, processor mode M, program counter P, and relocation bounds register R. The relocation bounds register is a tuple with (l, b) where l defines the relocation base and b defines size of the relocated memory area. An instruction producing an address a which is out of the relocation bounds is called to *memorytrap*.

$$S =< E, M, P, R > \tag{2.5}$$

The triplet $< M, P, R >$ represents the PSW[11]. The memory location $E[0]$ is used to store the PSW which was in effect before a *trap* while the location $E[i]$ is used to store the new PSW to be in effect after the trap. The real machine can exist in any one of a finite number of states. This set is called C_r. The execution of an instruction transforms the state of the machine into another. Thus, the definition of an instruction can be formalized as:

Definition 2.11. *An instruction i is a function from C_r to C_r. $i : C_r \to C_r$. So, for example, $i(S_1) = S_2$ [PG74].*

Now the definition of the action of a trap can be given by:

Definition 2.12. *An instruction i is said to trap if $i(E_1, M_1, P_1, R_1) = (E_2, M_2, P_2, R_2)$ where $E_1[j] = E_2[j]$, $E_2[0] = (M_1, P_1, R_1)$ and $(M_2, P_2, R_2) = E_1[1]$. In addition the trap must activate the supervisor mode, thus $M_2 = s$, and the complete memory must be accessible by the VMM resulting in $R_2 = (0, q - 1)$.*

This is the formal description of a context switch from user mode to system mode caused by a trapping instruction user mode. The

11 The Program Status Word (PSW) reflects the current state of the processor

definition requires the memory to be untouched upon a trap, except for memory location 0, as there the PSW needs o be in effect after the trap is stored.

A *memory trap* is a special trap caused by an instruction that wants to access memory out of the bounds specified by the relocation register R. With this knowledge, it is now possible to classify the instructions of an ISA to determine whether the real machine is virtualizable.

Definition 2.13. *An instruction* i *is privileged if and only if for any pair of states* $S_1 =< e, s, p, r >$ *and* $S_2 =< e, u, p, r >$ *in which* $i(S_1)$ *and* $i(S_2)$ *do not memory trap:* $i(S_2)$ *traps and* $i(S_1)$ *does not [PG74].*

The difference between the two states S_1 and S_2 is the processor mode. When executing the instruction i in state S_1, the instruction is executed in system mode and does not trap, as the execution permissions are not restricted in this mode. In case of executing the instruction i in state S_2, the instruction is executed in an identical state with the difference that the instruction is executed in user mode. In user mode, privileged instructions are not executable and cause the system to trap, as the execution permissions are restricted to non privileged instructions only. Thus, an instruction is privileged when trapping in user mode and not trapping in system mode. Traps caused by a memory trap are not considered, as their origin is located in accessing unmapped or protected memory locations.

There are two types of sensitive instructions being introduced. The term sensitive instructions represents the set of instructions that are able to modify the state of the real hardware and instructions that depend on the processor mode or on the memory location where they are executed. At first the class of instructions that are able to modify the state of real hardware is introduced. This class is called *control sensitive*.

Definition 2.14. *An instruction* i *is control sensitive if there exists a state* $S_1 =< e_1, m_1, p_1, r_1 >$, *and* $i(S_1) = S_2 =< e_2, m_2, p_2, r_2 >$, *and* $r_1 \neq r_2$ *or* $m_1 \neq m_2$ *[PG74].*

If an instruction attempts to change the available memory by modifying the relocation register R or affects the processor mode without going through the memory trap sequence, it is called control sensitive. A very simple example of a control sensitive instruction is the *mtmsr*[12] instruction of the Power ISA. This instruction directly affects the machine state register (MSR) which is the PSW of the Power ISA. If this instruction is executed in user mode the instruction will cause a trap.

The second type of sensitive instructions is called *behavior sensitive*. The effect of a behavior sensitive instruction depends on the current processor mode m, on the memory location p where this instruction is executed, or on the relocation register r. Reading a special instruction cache line is an example for a behavior sensitive instruction Power Architecture. Dependent on the actual position, the returned value can either be an instruction executed before or the current instruction itself. This kind of behavior sensitivity is called *location sensitivity*. If the result of an instruction depends on the processor mode, the instruction is called *mode sensitive*.

To describe the behavior sensitivity formally, the operator $\oplus$ is defined, so that the relocation register r is modified in the way that its base value is shifted by the value of x. Thus, the new value is $r' = r \oplus x = (l + x, b)$. The notion $E|r$ describes the contents of E of the part of memory restricted by r. Combining these two notions $E|r = E'|r \oplus x$ means that the memory contents of part E restricted by r is equal to the memory contents of part E' restricted by $r \oplus x$. Now the definition for behavior sensitive instructions can be given:

Definition 2.15. *An instruction i is behavior sensitive if there exists an integer x and states: Let $S_1 =< e|r, m_1, p, r >$ and $S_2 =< e|r \oplus x, m_2, p, r \oplus x >$, where $i(S_1) =< e_1|r, m_1, p_1, r >$, $i(S_2) =< e_2|r \oplus x, m_2, p_2, r \oplus x >$ and neither $i(S_1$ or $i(S_2)$ memorytrap, while $e_1|r \neq e_2|r \oplus x$ and/or $p_1 \neq p_2$ [PG74].*

Please note that the definition requires the observation of possible combinations of S_1 and S_2. This especially includes the case of $m_1 \neq m_2$ representing the case required for mode sensitivity. Depending on the states S_1 and S_2, the instruction i shows different effects either on the executable storage $e_1|r \neq e_2|r \oplus x$ and/or on the next instruction being executed ($p_1 \neq p_2$). An example for mode sensitivity is the *POPF* instruction of the Intel IA-32 ISA. The instruction causes to pop a word or doubleword to be popped from the current stack into the flags registers of the

12 Move to MSR

cpu. The stack pointer is then decreased accordingly to the word or doubleword data type. This effect only occurs in system mode while the instruction has no effect in user mode. Thus $m_1 \neq m_2$ and $e_1|r \neq e_2|r \oplus x$.

Now it is possible to define two classes of instructions of an ISA:

Definition 2.16. *An instruction* i *is sensitive if it is either control or behavior sensitive. If* i *is not sensitive, then it is innocuous [PG74].*

The VMM mainly consists of three components, a dispatcher, an allocator and an interpreter. The task of the dispatcher is the heart of the VMM and decides which component to execute. The allocator decides what system resources are to be provided to the VMs. The allocator ensures the spatial isolation of the VMs and is called if a privileged instruction attempts to change the machine resources. The interpreter is responsible for all other instructions which trap and provides one interpreter routine per privileged instruction to emulate the effect of the instruction which trapped.

There are three properties of interest when the VMM is executing an arbitrary VM:

1. Efficiency: All innocuous instructions are natively executed without intervention of the VMM.
2. Resource Control: For an arbitrary VM it is not possible modify the availability of system resources. The allocator has to be invoked upon any attempt.
3. Equivalency: Any VM executed by a VMM performs in a manner indistinguishable from the case when the VMM did not exist, with the exception of a different timing behavior.

Now it is possible to state the following Theorem for a VMM:

Theorem 2.6. *For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions [PG74].*

To proof this theorem, it has to be shown that a VMM fulfills the three properties of efficiency, resource control and equivalency. The first two properties are guaranteed by the definition of sensitive instructions. The only thing left to show is equivalency. This is done by defining a virtual machine map (VM map)

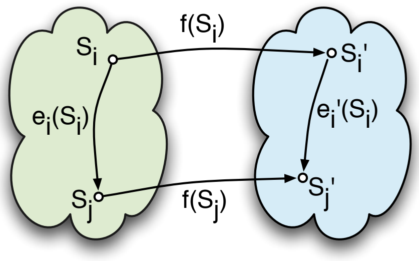


Figure 11: Virtual Machine Map

$f : C_r \rightarrow C_v$ where C_r is the set of machine states without a VMM being present in the memory and C_v is the set of machine states where the VMM is present in the memory.

Definition 2.17. *A virtual machine map is a ring homomorphism with respect to all the operations e_i in the instruction sequence set I. That is for any state $S_i \in C_r$ and any instruction sequence e_i, there exists an instruction sequence e_i' such that $f(e_i(S_i)) = e_i'(f(S_i))$. This corresponds to figure 11 [PG74].*

It is shown in [PG74] that the execution of arbitrary instruction sequences fulfill the equivalence requirement considering an example VMM and an example VM map being a one-one homomorphism with respect to all operations of the ISA.

This section clearly defines the formal requirements of the ISA to be efficiently virtualizable by the concept of a VMM. Therefore, efficient emulation of sensitive instructions is necessary to implement the interpreter of the VMM, which is responsible to emulate the trapping instructions of the ISA. The key aspect of efficient emulation is the control transfer from the VM to the VMM. This property is defined in theorem 2.6 demanding a sensitive instruction to trap what immediately transfers execution control to the VMM. The following sections will discuss the control transfer paradigm from the VM to the VMM for the case of full virtualization, paravirtualization and hybrid virtual machine systems.

Full virtualization

Full virtualization is a virtualization technique used to provide a certain kind of virtual machine environment, namely one that is a complete emulation of the underlying hardware. The executed VMs, further referred as guests, are not aware of the virtualization software executed on the host system. That especially means that the guests are not modified to support virtualization. So any software capable of running on the bare metal can be run in the virtual machine and, in particular, any operating systems. The key challenge of full virtualization is the interception and emulation of sensitive instructions. The requirements of full virtualizability have been introduced in detail in section 2.2.2. The interception of such sensitive instructions is quite complex as they cause an interrupt at any time, and the VMM has to determine which instruction caused this interrupt and what are the parameters of the instruction to emulate the instruction correctly.

Paravirtualization

Paravirtualization is a virtualization technique that presents a software interface to VMs that is similar but not identical to that of the underlying hardware. The intent of paravirtualization is to reduce the emulation overhead caused by emulating sensitive instructions in the domain of the VMM. Therefore, paravirtualization provides a so-called "hypercall interface". Hypercalls are system calls the virtual machine has to use to simplify virtualization tasks like emulating sensitive instructions. As an example, the emulation of an access to a sensitive register like the program status word (PSW) is more complicated in case of full virtualization than in the case of paravirtualization, because in case of full virtualization, a trap is caused by the sensitive instruction and the instruction causing this trap has to be identified including its parameters by the VMM while in case of paravirtualization the hypercall already contains this information and the VMM simply has to extract this from the hypercall. Besides the hypercall interface, which performs instruction level modifications, paravirtualizing a guest OS also means to perform structural modifications which require intimate knowledge of the guest kernel. Such structural changes may modify for example the address space layout to achieve fast VM communication. Despite this advantages, paravirtualization has a major drawback, because the usage of hypercalls and/or structural changes require the modification of the guest OS to support the hypercalls. This may be restricted due to licensing restrictions of the OS developer making a paravirtualization of that OS impossible.

Hybrid virtual machine system

Up to now, virtualization has been discussed for the case that the ISA fulfills theorem 2.6, but how to implement system virtualization when this property does not hold. The Intel IA-32 instruction *POPF* is behavior-sensitive, but does not cause a trap in user mode and thus violates theorem 2.6. Nevertheless, the ISA is virtualizable, but there should be spent some more effort on implementation. The naive solution would be to interpret every single instruction, but this of course degrades the performance dramatically. A pragmatic solution to this problem is to scan for critical instructions in advance and patch them. The patched instruction transfers the control to the VMM, allowing for the emulation of the non trapping sensitive instruction.

Definition 2.18. *A virtual machine system executing code of the VMs natively and in which the control transfer of some sensitive instructions is not caused by a trap caused by this instruction is called* hybrid virtual machine system*.*

Depending on the number of necessary control transfers, the approach of scanning and patching can still degrade the performance dramatically. Thus some enhanced methods have to be used to improve performance. It is possible to apply the scan and patch methodology to common techniques like binary translation, which is discussed in general in section 2.3.2. [SN05b]

2.2.3 Memory Virtualization

One of the properties of a VMM is resource control. Thus, the VMM has to ensure spatial isolation between the different VMs to prevent them from manipulating the memory of each other. When considering embedded systems without MMU[13], the performance of virtualization decreases extremely, as every memory access needs to trap and the VMM has to check whether the accessed memory location is protected or not. In reality, this situation is even worse, as in most embedded processors the memory

13 Memory Management Unit

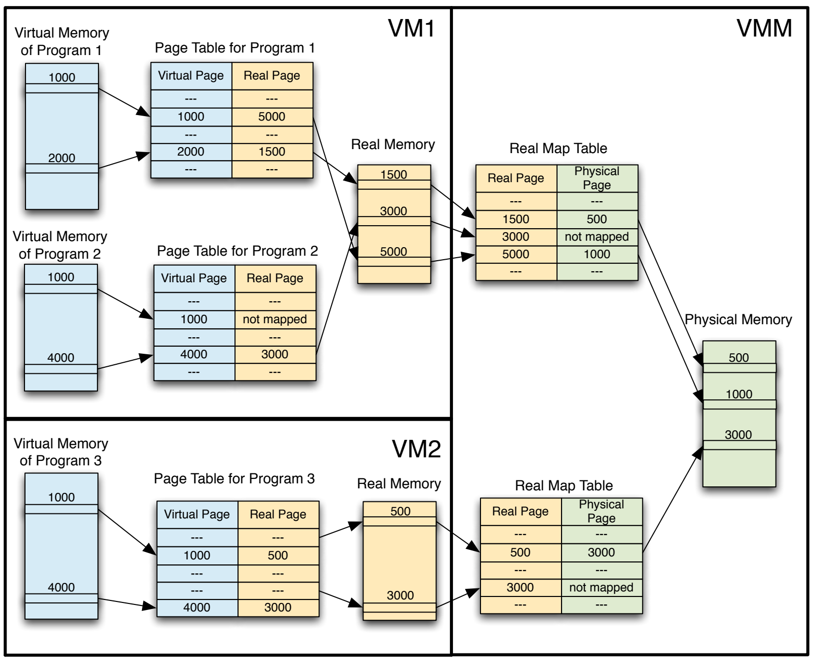


Figure 12: Memory Virtualization

accessing instructions do not trap. This results in the demand for MMU support. The concept of virtual memory can be seen as a special case of virtualization, as the concept of virtual memory defines a clear distinction between the logical view of memory as seen by an application programmer and the actual hardware memory resource as managed by the operating system. In a System VM environment, each guest VM has its own set of virtual memory tables. The MMU translates the virtual addresses to "real" memory addresses. The "real" memory addresses correspond to physical addresses in the case of non-virtualized systems. In a virtualized environment, the mapping of virtual addresses to "real" addresses is maintained by the guest OS, but the physical memory is controlled by the VMM and thus, another mapping from "real" memory to physical addresses has to be performed to guarantee spatial isolation, as different VMs may want to map a virtual address to the same "real" address. The process of mapping virtual addresses to "real" addresses and "real" addresses to physical addresses is depicted in figure 12.

Nowadays, the address translation from virtual to physical addresses is realized using architected page table or architected

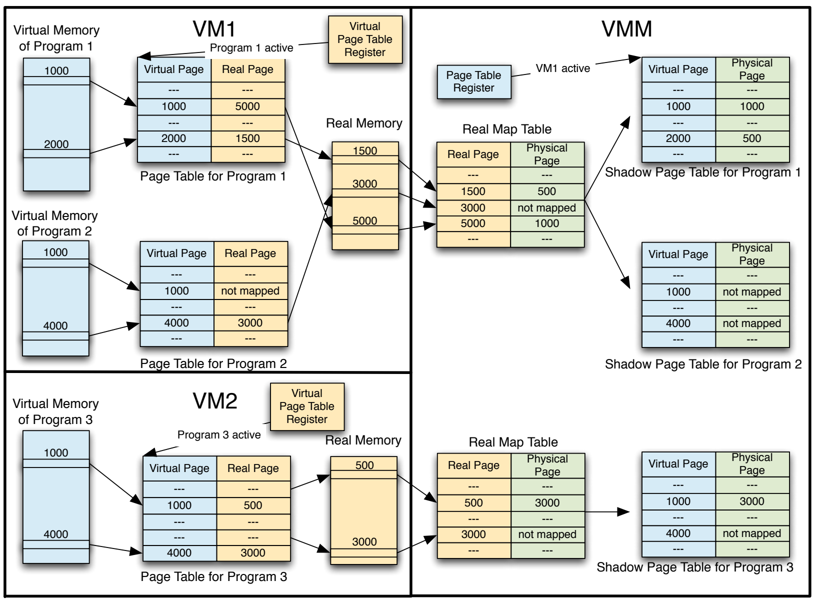


Figure 13: Virtualizing architected page tables

TLBs[14]. The virtualization of these two address translation architectures require different approaches that will be introduced in the following two sections. [SN05b]

Table 1: Actions of the VMM on triggered page faults

Mapped in page table	Mapped in shadow page table	VMM Action
Yes	Yes	No VMM activation
Yes	No	Handle page fault silently
No	Yes	Should not occur
No	No	Transfer page fault handling to guest OS

Architected Page Tables

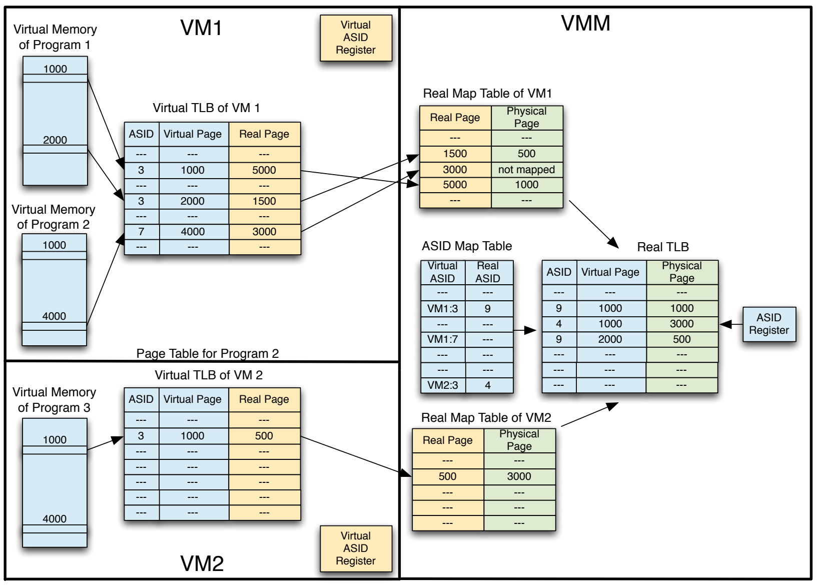
When an architecture provides architected page tables, the OS maintains its own page tables and the hardware is aware of this page tables by providing a page table pointer register. An address space switch is realized by changing the page table pointer to the page table of the corresponding address space. The page tables have to be in a format defined by the architecture. When a virtual address needs to be translated, the MMU walks the page table using the page table pointer and performs the translation when the corresponding entry has been found. If there is no entry found, a page fault is triggered by the MMU notifying the OS that the address translation cannot be performed for the given virtual address. In a virtualized environment, the page table register pointer needs to be virtualized. If a guest tries to access the page table pointer, and to either read or write it, a trap is triggered to activate the VMM. The page tables maintained by the guest OS do not contain virtual to physical mappings but virtual to real mappings. Thus, the VMM maintains shadow page tables to map real addresses to physical addresses as depicted in figure 13.

The maintenance of the page tables by the guest OS and the maintenance of the shadow page tables by the VMM introduces the problem of different VMM actions depending on the state of the mappings in the guest OS page tables and the shadow page tables. The case of having a page mapped in the shadow page table but not in the guest OS page table should never occur, because the modification of the page table needs to trap to the VMM. Otherwise, the page table and the shadow page table may get into inconsistent states. The two other cases are described in table 1. [SN05b]

Architected TLBs

In the case of architected TLBs, the TLB table itself needs to be virtualized together with the ASID[15] register. The ASID is a special tag field in the TLB associated with every entry in the TLB. The ASID describes the address space associated with the entry. Thus, it is possible to keep translations of different virtual

14 Translation Look-Aside Buffers
15 Address Space Identifier



Virtual TLB of VM 1

ASID	Virtual Page	Real Page
---	---	---
3	1000	5000
---	---	---
3	2000	1500
---	---	---
7	4000	3000
---	---	---

Real Map Table of VM1

Real Page	Physical Page
---	---
1500	500
3000	not mapped
5000	1000
---	---

ASID Map Table

Virtual ASID	Real ASID
---	---
VM1:3	9
---	---
VM1:7	---
---	---
---	---
VM2:3	4

Real TLB

ASID	Virtual Page	Physical Page
---	---	---
9	1000	1000
4	1000	3000
9	2000	500
---	---	---
---	---	---
---	---	---

Virtual TLB of VM 2

ASID	Virtual Page	Real Page
---	---	---
3	1000	500
---	---	---
---	---	---
---	---	---
---	---	---
---	---	---

Real Map Table of VM2

Real Page	Physical Page
---	---
500	3000
---	---
---	---
---	---

Figure 14: Virtualizing architected TLBs

address space simultaneously in the TLB to prevent flushing the TLB on an address space switch. The ASID register holds the current active ASID that enable translation of all entries in the TLB having the same ASID as in the ASID register. Thus, an address space switch is realized by writing to the ASID register. The advantage of an architected TLB is that the system software is able to define its own policy for replacement in the TLB, which is not possible in the case of architected page tables. The disadvantage is the complexity added to the system software to implement this policy.

When virtualizing architected TLBs, the instructions modifying the ASID register and the TLB entries need to trap to the VMM. The guest OS operates on virtual TLBs which can be reconstructed by the VMM using the ASID map table, the real map table, and the real TLB maintained by the VMM. The main difference to architected page tables is the additional ASID register and ASID tag. The virtual ASIDs, need to be mapped to real ASIDs, as the different VMs may use the same virtual ASIDs independently of each other. As already described in section 2.2.3, the VMM needs to manage the handling of page faults depending on the mapping state in the guest OS page tables and the shadow page tables. This problem is identical for the virtual TLBs of the guest OS and the real TLB. Table 2 shows the actions of the VMM

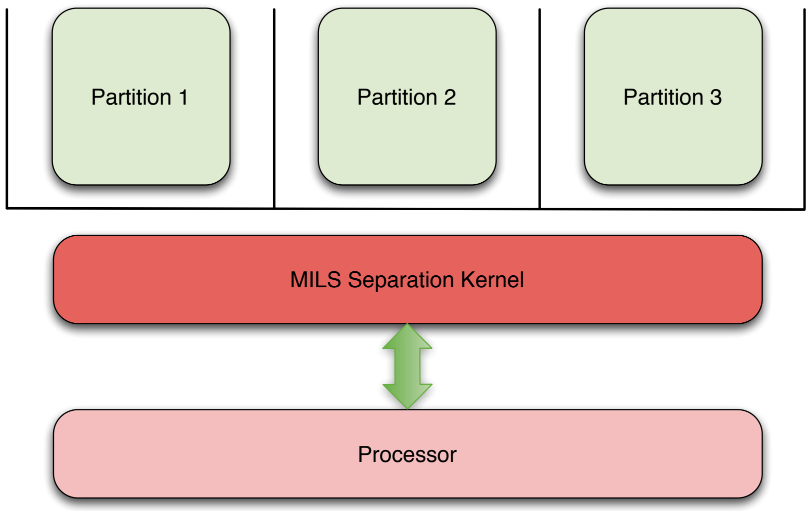


Figure 15: MILS architecture.

performed for the different cases. As already desribed for architected page tables, the case of having a TLB entry mapped in the real TLB but not in the guest OS virtual TLB table should never occur, because the modification of the virtual TLB needs to trap to the VMM. [SN05b]

Table 2: Actions of the VMM on triggered TLB misses

Mapped in virtual TLB	Mapped in TLB	VMM Action
Yes	Yes	No VMM activation
Yes	No	Handle TLB miss fault silently
No	Yes	Should not occur
No	No	Transfer TLB miss handling to guest OS

2.2.4 Multiple independent layers of security

The multiple independent layers of security architecture (MILS) satisfies highest security requirements. The main idea of this approach is the structuring of the participating components in a secure system to achieve extremely high security. This structuring is depicted in figure 15. The components of the system, like Processes or Applications, are isolated through the usage of different partitions that encapsulate these components. A single partition consists of the executable code, the data and the used system resources of the component. This partitioning allows for the possibility to separately verify each partition. For realizing this partitioning a software component called "separation kernel" is introduced. Its tasks are to isolate the partitions and to control the information flow between the partitions. The information flow control mechanism is the main addition that delimits the MILS architecture from general VMM architectures which are introduced in the next section. All components not responsible for partitioning and information flow control have to be placed in a separate partition. Keeping the separation kernel free from complex tasks allows for the general possibility for mathematical verification of the separation kernel, which is the overall goal of the MILS architecture. As an upper bound for verifiability of the separation kernel, a total number of about 5000 LOC is given nowadays. The verifiability of the separation kernel is given by the four aspects ensured by the MILS architecture:

1. Information Flow: The inter-partition communication needs the authorization of the separation kernel.
2. Data Isolation: No private data of a partition is accessible by any other partition.
3. Periods Processing: When a switch of the partitions occur, no information about the existence of other partitions are available.
4. Damage Limitation: A fault occurring in a partition has no impact on any other partition.

[Obj08]

2.3 EMULATION

In general, the term emulation is defined as the process of implementing the interface and functionality of one system or subsystem on a system having a different interface and functionality. Thus, emulation is a key aspect to implement virtualization, as instructions have to be emulated correctly to ensure the equivalency property of VMs. In case the VM uses the same ISA as the host system, all trapping instructions have to be emulated (see section 2.2.2). This is referred to as "same ISA virtualization". However virtualization may also provide an ISA different from the hosts ISA to the VMs. This is referred to as instruction set emulation allowing a machine implementing one instruction set, the target instruction set, to reproduce the behavior of software compiled to another instruction set, the source instruction set. To distinguish between instruction set emulation and complete virtual machine environments, the terms *guest* and *host* refer to complete virtual machine environments, while the terms *source* and *target* specifically address instruction set emulation. Instruction set emulation is a key feature for consolidating old hardware that is not available anymore, and for heterogeneous environments, as instruction set emulation allows for the migration of VMs using an ISA different from the host ISA [SN05b]. For many virtual machine applications, it is of great importance that the emulation of the instruction set is performed efficiently. This of course holds for embedded real-time systems. In the following, a short and abstract overview of the two basic emulation methods interpretation, covered in section 2.3.1, and binary translation, covered in section 2.3.2, is given [SN05b].

2.3.1 Interpretation

An interpreter program emulates and operates on the complete architected state of a machine implementing the source ISA, including all architected registers and main memory as depicted in figure 16. To execute a program of the source ISA, the interpreter has to manage the code and the program data which is maintained in the *source memory state* by the interpreter. To reflect the current state of the source machine, the interpreter holds a *source context block* which contains the various components of

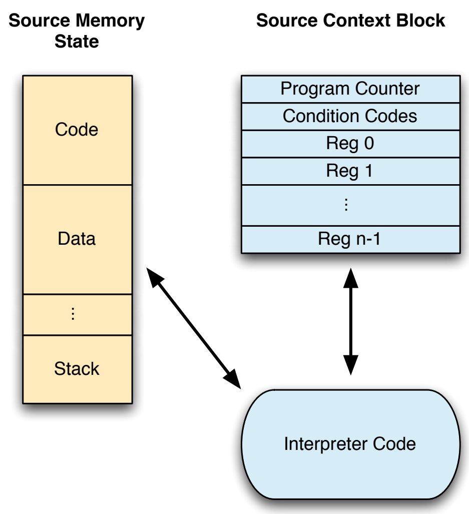


Figure 16: Overview of an interpreter [SN05b].

the source's architected state, such as general-purpose registers, the program counter, condition codes, and miscellaneous control registers.

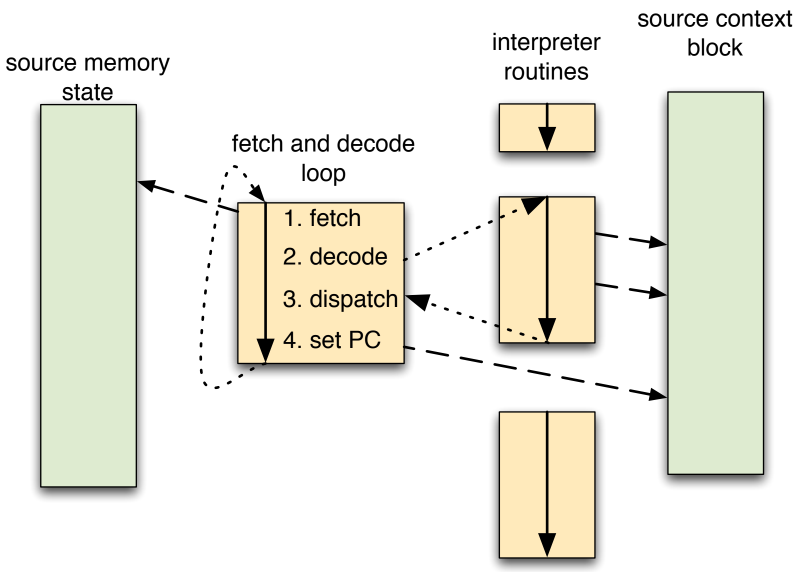
Basic interpretation is implemented as fetch and decode loop (see figure 17a). The program counter register stored in the source context block points to the address of the source memory state, where the next instruction is being fetched by a simple memory operation. After the instruction has been fetched, the instruction has to be decoded to decide how to emulate the instruction. Therefore, the instruction itself and the parameters have to be extracted. This can be very expensive, as the opcode needs to be identified first to decide how the parameters have to be extracted. This is all based on bit masking and shifting operations. After the instruction and its parameters have been extracted the fetch and decode loop dispatches to the responsible emulation method. After the emulation method has finished a branch back to the end of the main loop to unroll the call stack is performed. After this, the program counter is modified and a branch to the beginning of the loop is performed where the interpreter program is ready to fetch and decode the next instruction of the memory source state. To sum up, the fetch and decode loop introduces three branches which degrades the pipeline performance enormously [SN05b]. To address this problem, the fetch and decode loop can be integrated at the end of every interpretation routine increasing the code size but reducing the overhead branches to only one branch (see figure 17b) [Bel73, SN05b]. Obviously, interpretation is a complex process and introduces a lot of run-time overhead to the execution of a program of the source ISA [EG01, Bel73, EW01, EGKP02, Pit87, SCEG08, CEG07, PR98, BVZB05].

2.3.2 Binary translation

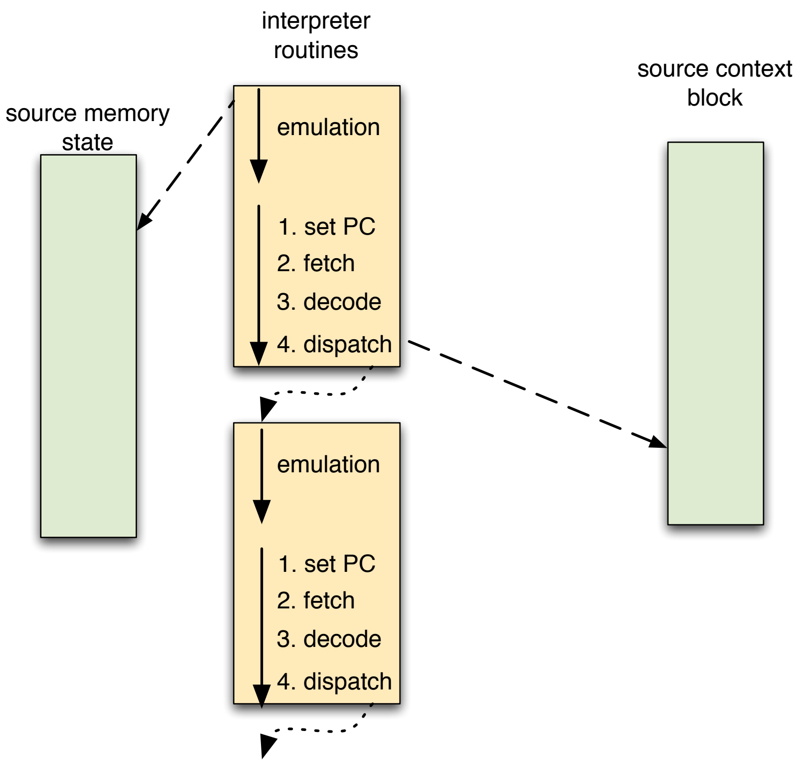
As already discussed in section 2.3.1, the run-time overhead of interpretation techniques is quite huge and a promising approach to reduce this overhead, called "Binary Translation",is a suitable method where the "source program" is translated into a program of the "target ISA", which is depicted in figure 18a. Due to the fact that instruction translation is customized, state mapping can be used to map registers of the source ISA directly to the target ISA registers (see figure 18b. This saves memory access to the context block and accelerates the execution.

A distinction is made between dynamic and static binary translation. In the case of dynamic binary translation, the translation of the program depends on the execution flow and is performed on the fly. If a certain part in the execution flow of the source program, called dynamic basic block, is not available as translated code, it will be compiled at runtime. Dynamic basic blocks are determined by the actual flow of a program as it is executed. A dynamic basic block always begins at the instruction executed immediately after a branch or jump, follows the sequential instruction stream, and ends with the next branch or jump. Jump or branch targets within this flow do not terminate the dynamic basic block. This is of course not suitable for real-time embedded systems, as the number of possible dynamic basic blocks is very high and a caching of all blocks is therefore not possible due to the resource limits. Furthermore, if the blocks cannot be completely cached, the worst case execution time of a dynamic basic block always has to consider the time for the translation from the source ISA to the target ISA destroying the advantage of the translation.

In the case of static binary translation, it is tried to offline translate the whole program from the source ISA to the target ISA. In this case, the execution flow cannot be considered as in the case of dynamic binary translation. Because of this, the source code

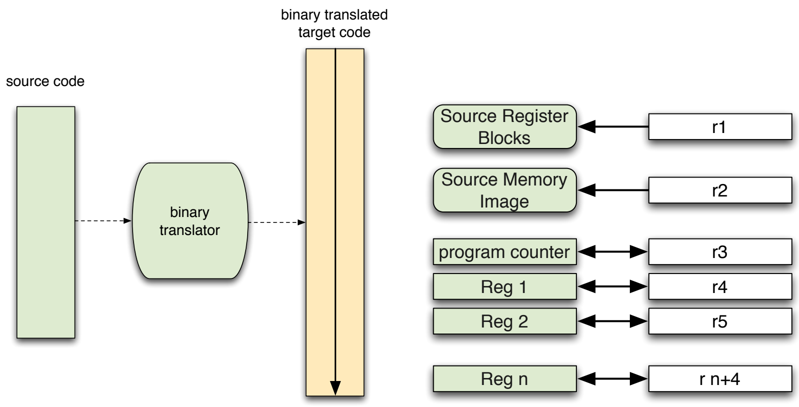


(a) Basic Interpretation



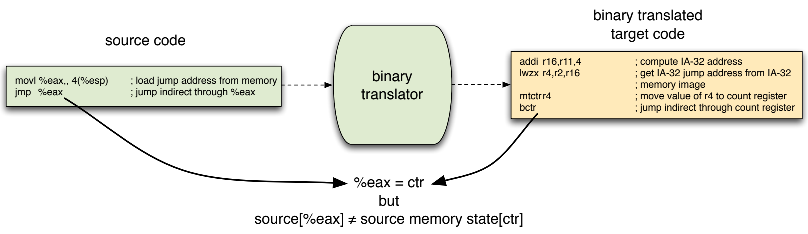
(b) Threaded Interpretation

Figure 17: Basic and threaded interpretation



(a) Binary Translation [SN05b]. **(b)** State Mapping [SN05b].

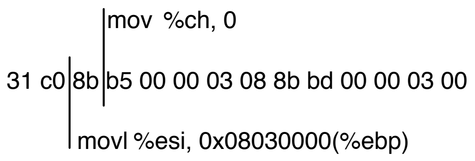
Figure 18: Binary Translation


source code
binary translator
binary translated target code
%eax = ctr
but
source[%eax] ≠ source memory state[ctr]

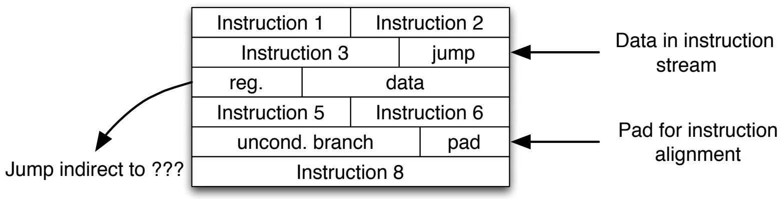
```
movl %eax,, 4(%esp)      ; load jump address from memory
jmp  %eax                ; jump indirect through %eax
```

```
addi r16,r11,4           ; compute IA-32 address
lwzx r4,r2,r16           ; get IA-32 jump address from IA-32
                         ; memory image
mtctr r4                 ; move value of r4 to count register
bctr                     ; jump indirect through count register
```

Figure 19: Code Location Problem [SN05b].


mov %ch, 0
31 c0 8b b5 00 00 03 08 8b bd 00 00 03 00
movl %esi, 0x08030000(%ebp)

(a) Finding IA-32 boundaries in an instruction stream [SN05b].



(b) Causes of the Code Discovery Problem [SN05b].

Figure 20: Code Discovery Problem

is divided into static basic blocks. In essence, static basic blocks begin and end at all branch or jump instructions and all branch or jump targets. Static basic blocks are the biggest atomic instruction sequences and can be translated in advance. Now one might think it is possible to completely translate the program, but the problem is that the binary code does not contain any information on where possible jump targets and instructions are located within the binary. The first problem of determining jump targets comes up when a relative jump instruction shall be translated. If the contents of the registers used to perform the relative jumps are calculated, it is in general impossible to perform the translation offline. This is due to the fact, that the result of the calculation is an address within the source program what has to be mapped to an address within the target program. This problem is called the "code location problem" (see figure 19).

The second problem is to identify instructions within the binary as compilers may intermix code and data due to padding for alignment reasons or to provide a mask that specifies which registers have been saved by a procedure caller at the time of the call. Whatever the reason for interspersing data in code sections is, it poses difficulties in statically identifying the starting points of all instruction sequences in a given region of memory. It becomes more problematic if the computer architecture is a CISC

[16], where the length of the instruction varies in contrast to RISC[17] architectures, where the length of an instruction is fixed (see figure 20a). This problem of identifying instructions within the binary is referred to as "code discovery problem". Hoorspool and Marovac proved in [HM80] that the code location and the code discovery problem are not solvable in general and if they are solvable they are NP-Hard problems.

As stated at the beginning of this section, emulation is a key aspect to implement virtualization, as instructions have to be emulated correctly to ensure the equivalency property of VMs. Beside same-ISA emulation, emulation offers the possibility to execute programs written for a source ISA. Interpretation is the straightforward approach to emulate a source program on a target ISA, but it suffers from its bad performance primarily caused by the overhead of the fetch, decode and dispatch steps. Nevertheless, interpretation has the big advantage of being robust against the code location and the code discovery problem, because interpretation in contrast to static binary translation, has knowledge of the execution flow. Binary translation comes in two flavors namely dynamic and static binary translation. Dynamic binary translation creates blockwise translated binary code at runtime depending on the execution flow and is as interpretation robust against the code location and code discovery problem, as interpretation can be integrated as fallback mechanism. The problem of dynamic binary translation is the huge overhead of caching dynamic basic blocks what is not possible to carry out completely and therefore has to be considered in the WCET. Static binary translation tries to translate the whole source program of the target ISA, but suffers from the code location and code discovery problem eliminating the possibility of translating the whole program in the general case [Hei08a, SCK+93, CM96, SBR05, GL94].

2.4 SUMMARY

At the beginning of this chapter, the properties of real-time systems have been introduced together with the common terms be-

16 Complex Instruction Set Computer
17 Reduced Instruction Set Computer

ing used in this area. This terms will be used throughout this document to be in line with this common understanding of real-time systems. To understand the common terms of virtualization, the different approaches of process and System VMs have been introduced. As this thesis aims at providing a system virtual machine approach for embedded hard real-time systems based on full virtualization, an in-depth look at the basics of System VMs was performed. Especially the formal requirements were highlighted, as these formulate the properties a VMM has to fulfill for realizing a full virtualization of third generation computer architectures in an efficient and secure manner. To realize a VMM, a memory protection mechanism is necessary to isolate the VMM and the VMs from each other. Todays systems mostly provide either an architected TLB or a software-managed TLB. For both architectures, the basic approaches to successfully virtualize a system with one of those architectures is presented. Finally an overview of emulation techniques, which are required by the VMM was given. With this knowledge in mind, it is now possible to address the problem of designing a system virtual machine for embedded hard real-time systems being based on full virtualization, which is the topic of the following chapter.

3 A VIRTUAL MACHINE MONITOR FOR EMBEDDED REAL-TIME SYSTEMS

Contents

The growing complexity of embedded real-time systems and their demand for high-level functionality typically provided by GPOSs like Linux, Windows and Mac OS X is the main motivation of this chapter. The OCM hierarchy already briefly introduced in chapter 2 is a perfect example for this demand of modern embedded systems.

The controller module of the OCM depicted in figure 21 operates on a closed loop called *motor loop* which needs to be controlled in hard-real time, as missing a deadline may destroy the controlled motor. Thus, the controller module needs an RTOS as execution platform. The reflective operator module controls and monitors the controller module. The different configurations for the controller module encapsulate different control algorithms, which may differ in quality of control, energy consumption, fail safe behavior and so on. These configurations can be selected by the reflective operator and assigned to the controller module to use this configuration. This process can be performed either in hard real-time or in soft real-time depending on the current situation. The cognitive operator is responsible for using the data provided by the reflective operator for self-optimization processes like machine learning approaches, model-based optimization or knowledge-based systems. Depending on the result of the self-

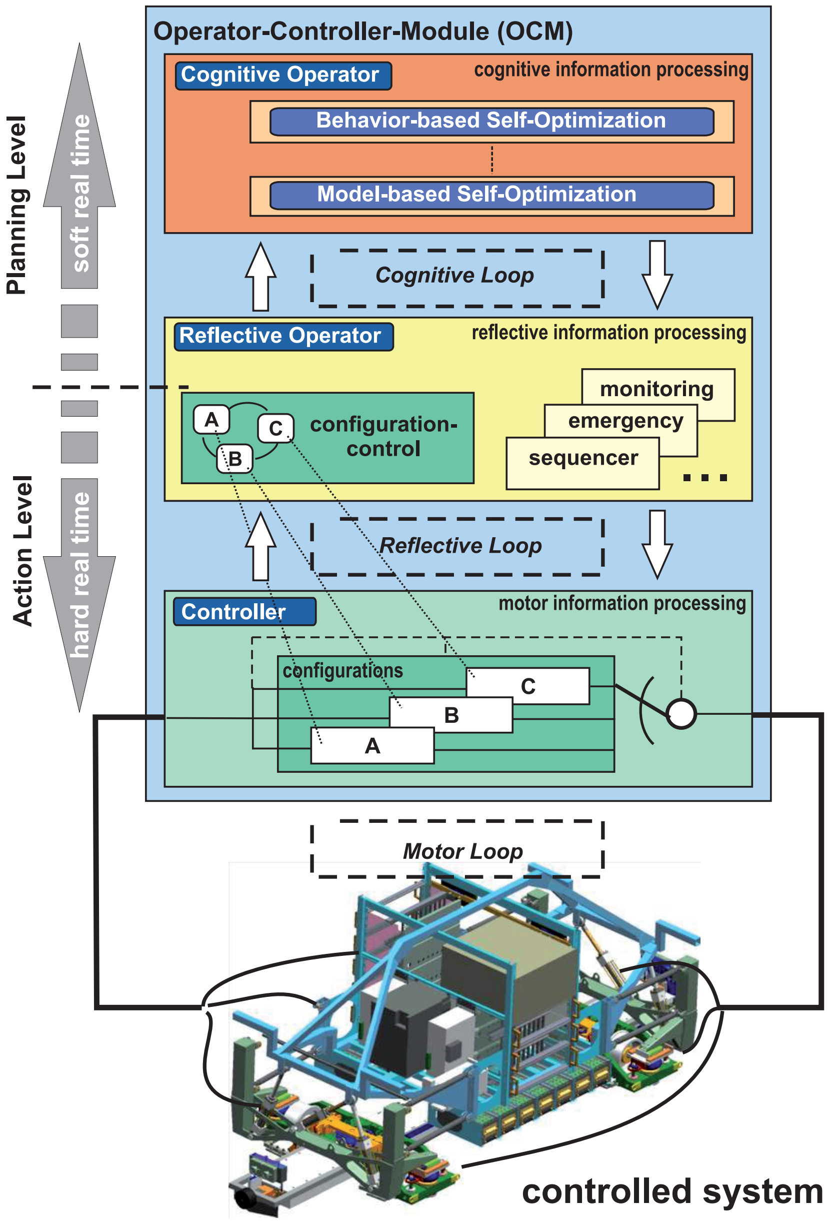


Figure 21: Operator Controller Module

optimization process, a configuration is proposed to the reflective operator. When the monitored system is in a state allowing for a transition into this proposed configuration, the transition is triggered by the reflective operator. Nevertheless, the reflective operator is always able to force a switch into a different configuration when needed. This information is then forwarded to the cognitive operator triggering again the self-optimizing process. The OCM is very complex and developed by different research teams providing software components for the controller module, the reflective operator and the cognitive operator.

One of the main problems of building such complex systems is the integration of software components into a big integrated system as described by Broy in [Bro06]. He uses the automotive domain as example for a mechatronical systems. When considering a top line car, there are about 70 ECUs[1] installed with different tasks, from hard real-time tasks for actuating elements to soft real-time tasks for multimedia devices to non real-time elements like the ECU for the window lifter. For the integration of such a system Broy says:

"Traditionally quite unrelated and independent functions (such as braking, steering, or controlling the engine) that were freely controlled by the driver get related and start to interact. The car turns from an assembled device into an integrated system. Phenomena like unintentional feature interaction become issues." [Bro06]

This unintentional feature interaction becomes an extremely important issue to be handled safely as the verification of a complex integrated system is an extremely hard task also identified by Broy:

"Since today, by their design, architecture and the interaction between the sub-systems are not precisely specified, and since the suppliers realize the sub-systems in a distributed process, it is not surprising that integration is a major challenge. First of all a virtual integration and architecture verification is not possible today, due to the lack of precise specifications. Second, in turn the sub-systems delivered by the suppliers do not fit together properly and thus the integration fails. Third when trying to carry out the error correction due to the missing guidelines of

1 Embedded Control Units

architecture, there is no guiding blue print to make the design consistent." [Broo6]

In a distributed system where every task is placed on a single ECU the unintentional interaction or a hardware failure are fairly the only issues endangering a component's functionality. But this kind of feature distribution is currently something up for discussion as the trend is to have less dedicated ECUs in favor of more centralized multi-functional hardware, which is also stated by Broy:

"The car of the future will certainly have much less ECUs in favor of more centralized multi-functional multipurpose hardware, less communication lines and less dedicated sensors and actuators. Arriving today at more than 70 ECUs in a car, the further development will rather go back to a small number of ECUs by keeping only a few dedicated ECUs for highly critical functions and combining other functions into a small number of ECUs, which then would be rather not special purpose ECUs, but very close to general-purpose processors. Such a radically changed hardware would allow for quite different techniques and methodologies in software engineering." [Broo6]

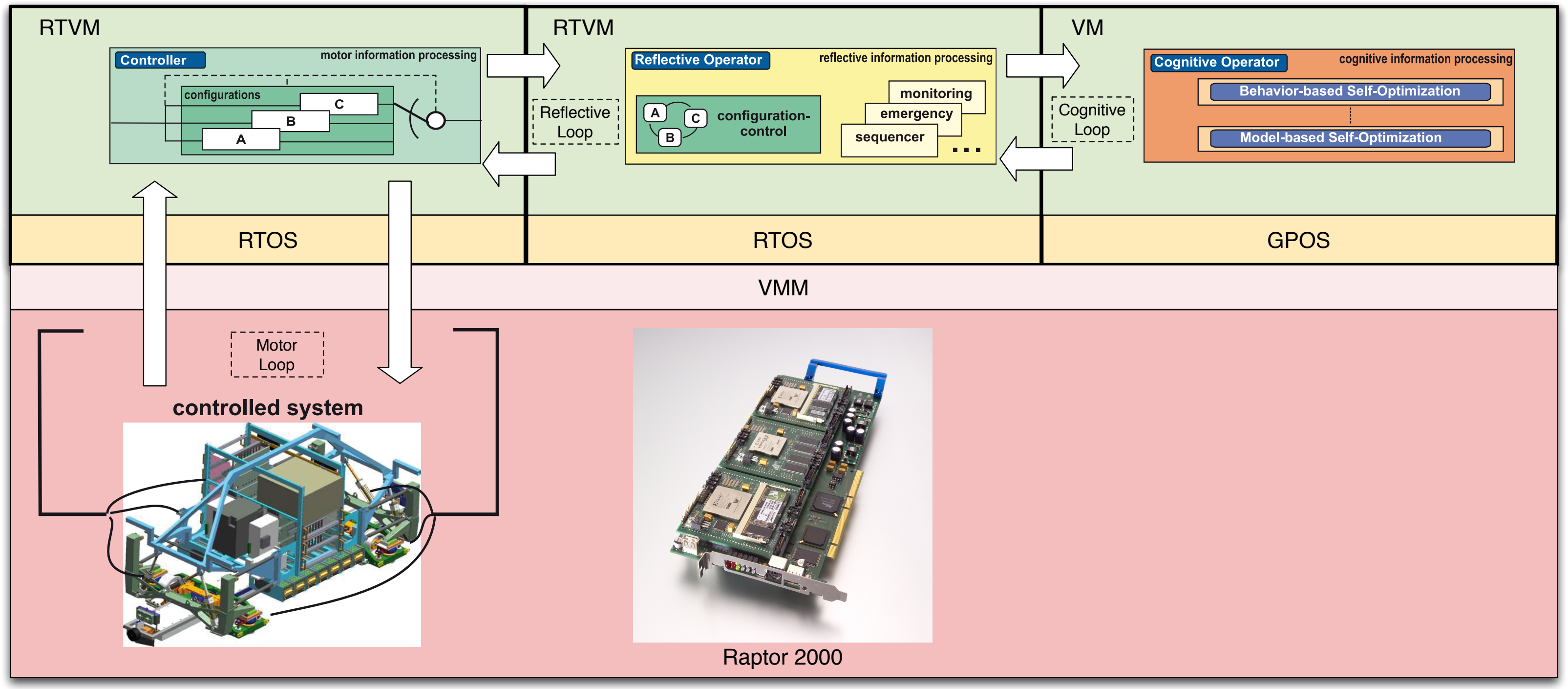


Figure 22: OCM virtualized

The trend towards more centralized multi-functional hardware boosts the problem of unintentional interaction of software components as they share the processor, memory and I/O devices in this case. Thus, the task of the system software is to prevent unintentional interactions, which are not based on the communication between the components like the domination of hardware resources (for example the processor or memory), faulty implementations allowing buffer overflows, heap overflows, stack overflows, race conditions and so on, as these unintentional interactions endangers all components running on the same hardware. This is a typical task for system software and is normally covered by using virtual memory isolating the tasks from each other, but there is a new demand for high-level functionality in such complex embedded systems. Reconsidering the OCM, one can easily see that the cognitive operator requires a lot of high-level APIs to perform the optimization tasks using machine learning approaches, model-based optimization or knowledge-based systems. This is typically not a task of embedded RTOSs. This is the point where virtualization can show its strength as virtualization in the example of the OCM allows to isolate the cognitive operator into a virtual machine together with a GPOS providing all of the high-level functionality needed by the cognitive, while isolating the reflective operator and the controller module in RTVMs running an RTOS (see figure 22). Thus, virtualization helps to simplify the integration process of components with different requirements to their operating systems as it allows to run multiple operating system while spatially isolating them and preventing unintentional interactions like resource domination or attacks resulting from faulty implementations. A big advantage of virtualization against the use of a single operating system providing all functionality is that VMMs are by design very small and are better to verify than a big operating system full of high-level functionality.

3.1 PROBLEM STATEMENT

The overall goal of this thesis is the integration of software components into a big integrated system as depicted in figure 23. Thus, there are given real-time systems which may have been executed on different to the integrated host system. To make

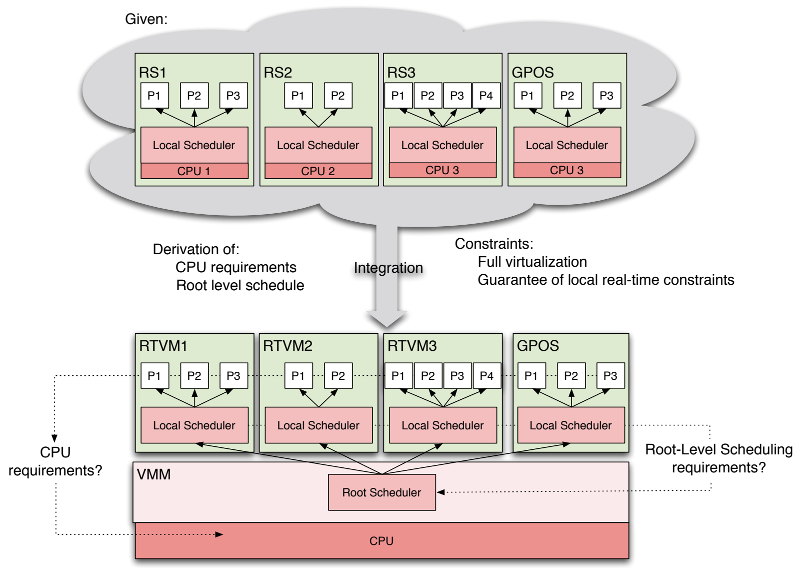


Figure 23: Problem of integrating given real-time systems into an virtual system hosting these real-time systems as RTVMs.

these real-time systems executable on the integrated system, every real-time system is encapsulated as RTVM[2]. These RTVMs are then executed on an especially designed VMM, which ensures temporal and spatial isolation to prevent the described unintentional interactions. There already exist a few commercial virtualization platforms or VMMs for a range of embedded processors. nearly all of them being proprietary systems. All of the available products only use paravirtualization trying to provide reasonable performance and support realtime applications only by the use of dedicated resources. Naturally, this limits the applicability of virtualization using these products to a subset of all possible scenarios, as in general the paravirtualization interfaces are not standardized. Especially whenever there are applications that cannot be paravirtualized since the source code is not available, these applications cannot be virtualized using the currently available virtualization products, since this would require a binary analysis of the whole application, which most often is not completely possible. Thus, within this thesis, a VMM is designed to overcome this problem by introducing a configurable hybrid VMM architecture designed and implemented for the PowerPC405 processor which allows for the virtualization of unmodified applications as well as paravirtualized applications

2 Real-Time Virtual Machine

or even a combination of both. To describe this more in a more formal manner, the ABI is kept to be configurable. The paravirtualization effort is thus decreased, as only the required hypercalls need to be implemented in the guest OS. The support for paravirtualization is motivated by the integration of open source GPOSs for High-Level API support like Linux which already provide paravirtualization interfaces. Support for realtime applications was the next major goal of the design, which allows the integration of any kind of scheduling mechanism for VMs while being completely deterministic. Furthermore, the high configurability allows the system to be optimized explicitly for the intended field of use. This affirmed the research in this direction with hybrid virtualization being a relevant topic in industrial embedded systems. Finally, it is desirable to know in advance how the WCET of the executed guests are affected, as these WCETs are necessary to determine the CPU requirements of the virtual real-time system.

When supporting multiple VMs on a virtualization platform, the VMM, needs to implement a scheduling algorithm according to which it switches between the VMs. From the point of the VMM the executed VMs are just processes, but as they execute an OS, they schedule a set of tasks by themselves (see figure 23). In the case of full virtualization, these VMs appear to be blackboxes while in the case of paravirtualization, the VMs can communicate with the VMM scheduler. The goal of a VMM is to give to each VM the illusion of having the resources of a complete system at its disposal, while these resources are only subsets of a physical machine. Thus, the VMM needs to implement a partitioning policy which ensures the correct timing behavior of all executed VMs. For RTVMs this timing behavior has hard deadlines the VMM has to cope with. Existing approaches are restricted to the application of paravirtualization when supporting hard real-time, which is not in general possible due to the already mentioned licensing restrictions, or they are not able to derive the root level schedule automatically from a given set of real-time systems. Thus, the goal of this thesis is to automatically derive an integrated virtual system hosting the given real-time system while guaranteeing the real-time constraints and preventing the application of paravirtualization, as this would require the availability of the guest OS source code.

To sum up, the goals of thesis are described in a hierarchical manner:

1. Integration of given real-time and general purpose systems into an integrated virtual real-time system
    a) Configurable hybrid VMM
        i. Configurable ABI (Full and Paravirtualization)
        ii. Extensible scheduler interface
        iii. WCET Determination of virtualized guests
    b) Hierarchical Scheduling of RTVMs
        i. Derivation of CPU requirements
        ii. Derivation of root-level schedule

All these goals share the constraints of full virtualization support, local real-time constraints and low jitter.

3.2 RELATED WORK

This section describes the related work which is relevant for the goal of providing a configurable hybrid VMM (1a of the problem statement in section 3.1). The goal of hierarchical scheduling RTVMs is addressed in the next chapter, as this is separately discussed due to being thematically totally different from the design of a VMM.

3.2.1 Academia

The related work from academia addresses mainly technical details like memory and I/O virtualization, where a lot of papers can be found. Unfortunately, there exist some few papers on paravirtualization extensions which do not address the defined goal of providing a configurable ABI, but they are worth mentioning, as especially the previrtualization approach is interesting for minimized paravirtualization effort when having unlinked assembly code as origin. Finally, an approach is introduced to implement virtual memory suitable for mixed criticality systems like the proposed configurable hybrid VMM, which is able to host GPOSs beside RTVMs.

GandalfVMM

The GandalfVMM was developed at the University of Tsukuba by Oikawa et. al [OIN06] and focuses on the simplification of paravirtualization overhead. The authors claim that their approach called Mesovirtualization [IO07] reduces the effort of paravirtualizing a guest OS by two orders of magnitude. One significant characteristic of mesovirtualization is how a VMM handles sensitive instructions used in guest OSs. While they are emulated by a VMM very much like in full virtualization, only the essentials are emulated in GandalfVMM. There are some cases that sensitive instructions which are not emulated by a VMM produce unexpected results for a guest OS. Rather than having every sensitive instruction changed to trap to a VMM and handled it with hard work, mesovirtualization slightly modifies the guest OS source code preventing an interrupt to the VMM. For some parts of the host machine that are considered safe to be dedicated or shared, the VMM does not virtualize these parts and allows guest OSs to touch them directly. Such characteristics lead to a lightweight VMM, because there is no need to virtualize the full ability of the host machine. It also leads to the reduction of VMM's use of processor time, which makes it possible to provide higher performance to guest OSs.

Previrtualization

Due to the intrusive nature of paravirtualization, guest OSs are mostly restricted to a special VMM when they are paravirtualized. Responsible for this cutback are the instruction level and structural modifications that build a very specialized interface to the VMM. The idea of pre-virtualization is to modularize and decouple this specialized interface from the guest OS to be flexible and degradable to allow for the execution on raw hardware and VMMs that lack the support of pre-virtualization.Therefore, sensitive instructions will be padded with a sequence of no-op instructions and annotated to determine their location at runtime by the VMM. The VMM is able to replace these no-op instructions with higher performance alternatives that delegate the instructions to a so-called "in-place VMM". The in-place VMM provides a virtual model of the CPU and the devices and is able to defer and batch hypercalls depending on the state of the virtual cpu and device models. It is necessary to be able to provide the scratch space by padding the unlinked assembly code of the guest OS as at this step the padding is simply possible due to the fact that addresses are available as symbols [LUC+06].

Memory Virtualization

The design of the memory virtualization approach depends on the hardware support of the target processor architecture as already presented in section 2.2.3. When using a processor architecture with architected page tables, the VMM manages the real map tables of every VM and adjusts the PTR[3] to the real map table of the currently active VM. The VMM has no control over the replacement strategy applied on the TLB and thus the VMM cannot organize the TLB to manipulate the TLB hit or miss rate for real-time or non real-time VMs. There are already several approaches to increase the TLB hit rate and to reduce the upper bound for memory access time. To some extent, the results are remarkable, but they do not solve the main problem. Virtual memory and real-time constraints cannot be reconciled through a simple increase of the memory management performance. A non-deterministic, high-performance TLB miss handling is first of all still non-deterministic. In this thesis, an extension to the multiple page table design introduced by Bennett and Audsley [BA01] is proposed. This extension covers real-time and non real-time requirements in combination with a dynamic partitioning of the TLB to improve the performance of soft and non real-time tasks at the cost of hard real-time tasks as they do not have any benefit from being faster than required by their WCET. The main goal is to reduce the negative effect of context switches on the TLB miss rate of specific soft and non real-time tasks, without losing the ability to handle hard real-time tasks within their WCET.

Arithmetic-Based Address Translation [ZP05] bypasses the TLB through the replacement of most of the virtual address translations with fast arithmetic add operations. It considers that the physical allocation of virtual pages conforms to certain rules. Sequences of virtual page numbers are identified and mapped to sequences of consecutive physical page frames. The result is a virtual-to-physical address translation in constant time for these virtual addresses, but a default TLB is still used for the rest.

3 Page Table Register

After comparing different approaches to improve TLB performance, Peng et al. [PLW+06] proposed a superpage design which aims to reduce TLB miss rates by improving TLB coverage. Talluri et al. [TKHP92, TH94] as well investigated the reduction of TLB miss overhead through a page enlargement. Jacob et. al. reinforced the dominant impact of the miss rate on the TLB performance in [JM98b, JM98a]. Two-level TLB is a technique of Peng et al. [PLW+06] which aims to reconcile high coverage with low latency. Multi-level TLBs constitute a way to decrease the average translation time, but do not guarantee deterministic response times. Intermediate-level Skip Multi-Size Paging by Suzuki and Shin [SS97] is based on Multi-level Paging. These approaches do not take into account that real-time systems may be composed of hard, soft and even non real-time tasks with differing timing characteristics. Bennett and Audsley [BA01] discovered this property and proposed a modular page table design (figure 24). They suggest the implementation of custom page miss handlers, depending on the individual real-time constraints. An implementation – which is possible for software-managed TLBs – provides full virtual memory support for less critical tasks and a predictable address translation at the expense of only a restricted virtual memory support for hard real-time tasks. This work is based on their findings.

Another interesting approach to increase the TLB performance is the partitioning of the TLB. Channon and Koch [CK97] proposed a partitioning scheme of the TLB based on reference and ownership characteristics. They divided the TLB into sections for instruction fetching, data fetching and a section for the operating system kernel. The partition boundaries are adjusted at runtime to balance the TLB misses within the partitions.

3.2.2 Industry

At the beginning of this section, an introduction to engineering software quality standards of the automotive and avionic industry is presented to show up their requirements. The following presented VMMs from Greenhills, Lynuxworks, Windriver and SYSGO are based on theses engineering standards. Furthermore, an overview of XEN is given, as XEN is very popular in the server consolidation domain. All of those presented approaches

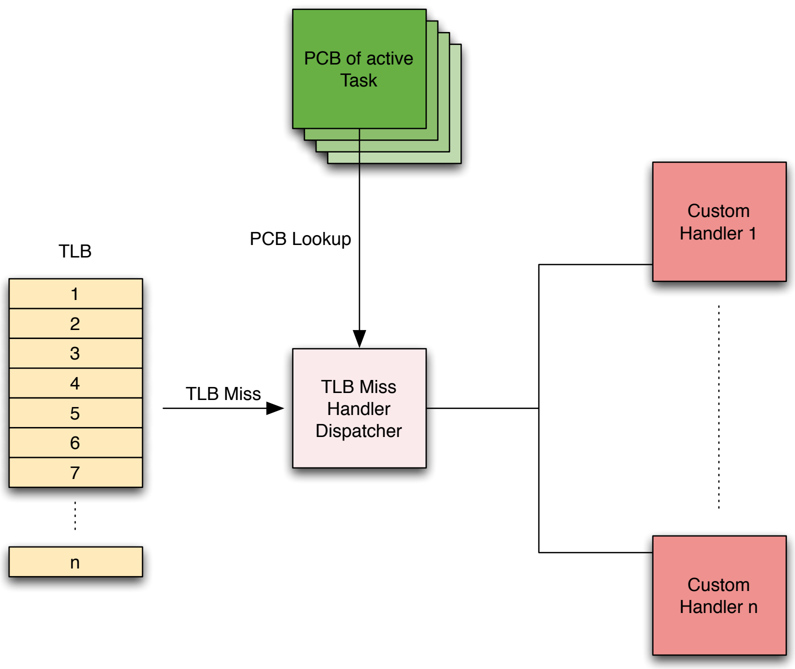


Figure 24: Modular Page Tables

require paravirtualization to support hard real-time and do not provide an extensible scheduler interface to address the problem of hierarchical scheduling by configuring or exchanging the VMM scheduler.

Engineering standards

In the market of embedded systems, it is common that companies, which are associated via the chain of economic value added of one or multiple products, agree on a combined standard of their solution. In the area of real-time operating systems, different standards exist. Some standards specify interfaces rather the functionality between operating system and applications. Other standards concentrate on the development process of dependable embedded systems to certify a maximum of software quality. These standards can also be applied to the development process of operating systems to certify the dependability of them. For example, the OSGi[4] has created a standard for networked applications. In the automotive sector, many developers use the

4 Open Service Gateways initiative

operating system specification for car control system OSEK [5]. This standard has been partly published as an international standard [ISO05]. DO-178B[6] is a norm for the development of software from the avionic area. The standard has been developed from the RTCA[7] and EUROCAE[8]. The American authority FAA [9] employs the norm for certification of software and software development processes in the area of avionic. The FAA certificate is the highest security certificate issued by the FAA. Integrity Secure Virtualization is built upon technology taken from Integrity RTOS-178B, which has been certified besides FAA DO-178B Level A with EAL 6+ [10] by the NSA[11]. The Evaluation Assurance Levels 1-7 are defined in the Common Criteria for Information Technology Security Evaluation. Common Criteria is a framework in which computer system users can specify their security functional and assurance requirements, vendors can then implement and/or make claims about the security attributes of their products, and testing laboratories can evaluate the products to determine if they actually meet the claims. In other words, Common Criteria provides assurance that the process of specification, implementation and evaluation of a computer security product has been conducted in a rigorous and standard manner.

Different automotive manufacturers and suppliers have allied to an international consortium and try to establish an open standard for the electric and IT architecture in the automotive field with AUTOSAR[12]. The core of the architecture of AUTOSAR is the AUTOSAR RTE[13] that abstracts from a real topology of control devices. The programming language ADA is often used to program dependable embedded real-time systems. For this purpose the language is adequate because it supports special attributes to increase the dependability: e.g. type safety, run-time tests for memory overflow or simplified program verification.

5 German: Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen; English: Open Systems and their interfaces for the Electronics in Motor Vehicles
6 Software Considerations in Airborne Systems and Equipment Certification
7 Radio Technical Commission for Aeronautics
8 European Organisation for Civil Aviation Equipment
9 Federal Aviation Administration
10 Evaluation Assurance Level 6+ : semi-formally verified design and tested
11 National Security Agency of the United States of America
12 AUTomotive Open System ARchitecture
13 Run-Time Rnvironment

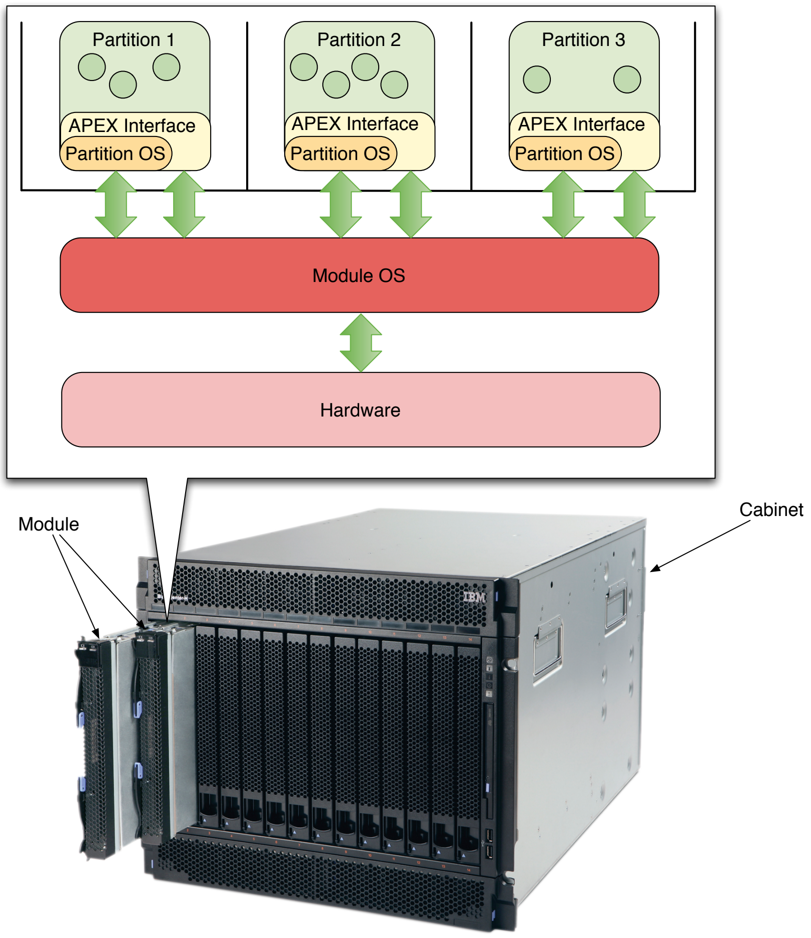


Figure 25: Arinc653 architecture diagram.

Since 1983 a ISO/ANSI standard for this programming language is existing.

ARINC 653

The ARINC 653 specification defines the functionality that an operating system must guarantee robust spatial and temporal partitioning together with an avionics application programming interface. The standard application interface is called *APEX*[14] and defines a set of software services an ARINC 653 compliant OS needs to provide to avionics application developers. The ARINC 653 standard only specifies this interface while it leaves the implementation details to the OS vendors.

14 Application Executive

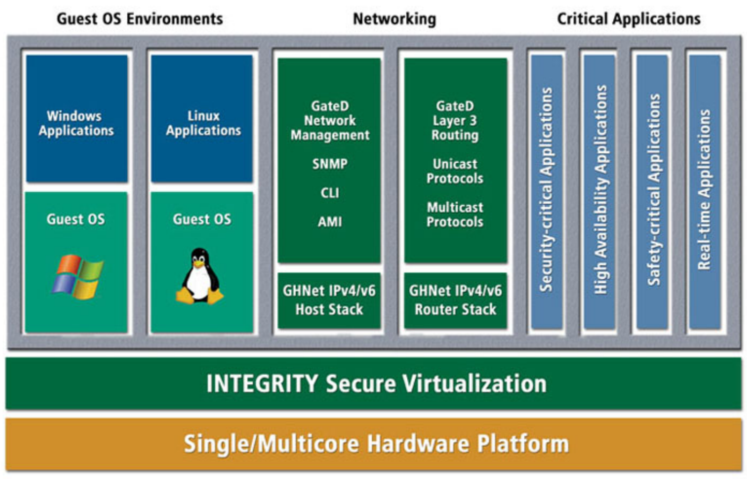


Figure 26: Green Hills Secure Virtualization architecture diagram.

The architectural design of the ARINC 653 specification is in general identical to the MILS design 2.2.4. The separation kernel is called *Module OS* and is deployed on a single processing unit called *module* and is able to host one or more avionics applications and to execute them independently. Due to the MILS based architecture, the *Module OS* provides the needed spatial isolation for fault containment. A partition in the sense of ARINC 653 encapsulates an avionic *program* distributed over a number of *processes* being executed by a *Partition OS*.

Greenhills Integrity Secure Virtualization

Integrity Secure Virtualization is a separation kernel based on the MILS architecture introduced in section 2.2.4 and is able to host arbitrary guest OSs alongside a comprehensive suite of real-time applications and middleware. The architecture of Integrity Secure Virtualization is shown in figure 26. The support for real-time applications is achieved by the usage of real-time technology taken from Integrity RTOS-178B into the separation kernel. Thus, real-time applications are executed directly by the separation kernel and do not require an RTOS within a partition to host them. This extends the separations kernel task of Integrity Secure Virtualization to real-time management within real-time partitions. Somehow, this is against the general design of separation kernels, as this adds additional code to the separation kernel making the formal verification more difficult. Nevertheless,

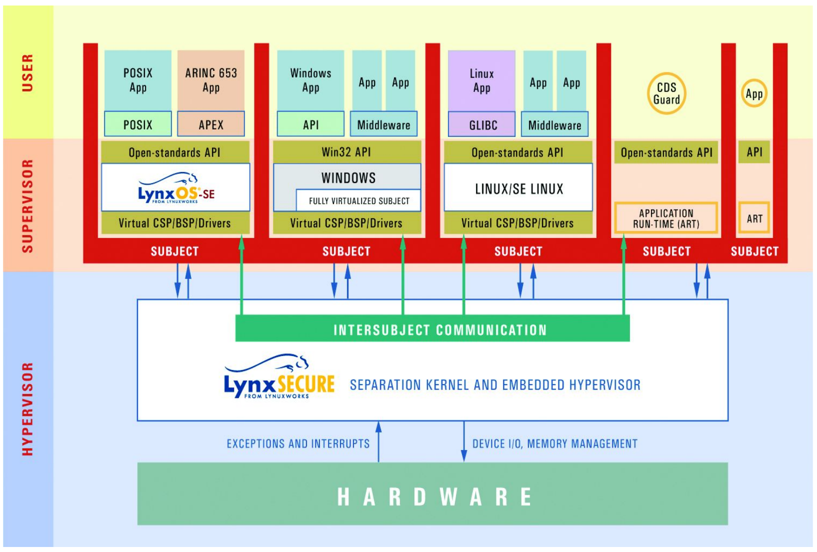


Figure 27: Lynuxworks Lynxsecure architecture diagram.

Integrity Secure Virtualization has been certified FAA DO-178B Level A for avionic systems controlling passenger and military jets [Gre10].

Lynuxworks Lynxsecure

The Lynxsecure separation kernel, depicted in figure 27 is also obviously based on the MILS system architecture with strict adherence to data isolation, damage limitation, information flow policies and periods processing identified in this architecture. Thus, the different subjects within the figure represent resource partitions encapsulating an OS or another runtime environment. The resources assigned to each resource partition are dedicated exclusively. Lynxsecure thus guarantees resource availability at any time. The resources are accessible from within the partitions through the virtual chip support package (CSP) and virtual board support packages (BSP). The virtualization of these support packages ensure that the guest OS is virtually executed on supported hardware.

Lynxsecure offers the possibility to use full and paravirtualization side by side on the same hardware. The full virtualization feature for example can be used to host Windows as a guest OS. However, LynxOS-SE, developed by Lynuxworks, is the only supported RTOS and is virtualized using paravirtualization. The

Lynxsecure separation kernel has been designed to be certifiable to Common Criteria EAL-7[15] and FAA DO-178B level A. Up to now Lynxsecure has not obtained EAL 7 [Lyn09].

Windriver VMM

The Windriver VMM has entered the market after Greenhills Integrity Secure Virtualization and Lynuxworks Lynxsecure. It is also based on the MILS architecture. Windriver VMM uses paravirtualization to host guest OSs. The resource partitions, which encapsulate the guest OSs, are called Virtual Boards. Compared to Integrity Secure Virtualization and Lynxsecure, the Windriver VMM does not offer anything special [Win10].

SYSGO PikeOS

In contrast to Greenhills Integrity Secure Virtualization, Lynuxworks Lynxsecure and the Windriver VMM, Sysgo follows with PikeOS the architectural design principles of a microkernel. Figure 28 shows the architectural design of PikeOS. The separation of the microkernel and the PSSW[16] helps to keep the microkernel smaller and thus less fault-prone. The PSSW can be understood as an abstracted management layer of the providing microkernel while the Pike OS microkernel provides the main functionality typically provided by an RTOS microkernel:

- Task & Thread Management
- IPC
- Memory Management
- Interrupt Handling
- Scheduling

The required temporal and spatial isolation is realized by introducing two types of partitions:

- Resource Partitions
- Time Partitions

A resource partition is a set of PikeOS tasks sharing a bounded set of kernel resources assigned to the resource partition and

15 Formally Verified Design and Tested
16 PikeOS System Software

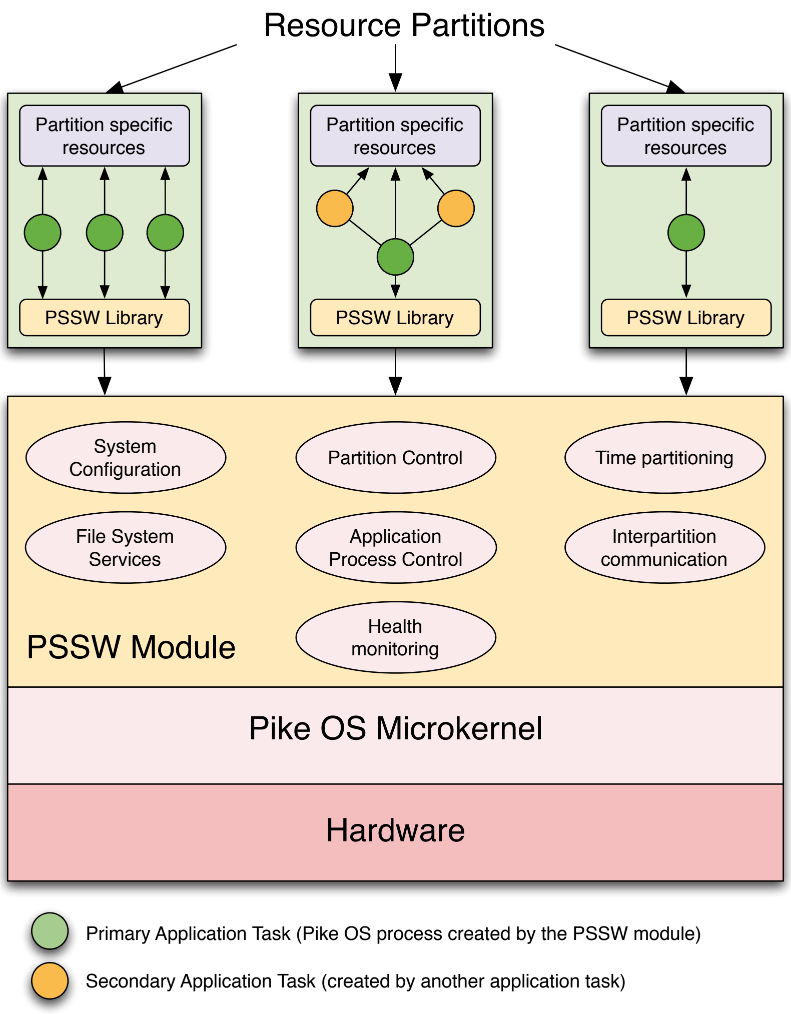

Resource Partitions
Partition specific resources
Partition specific resources
Partition specific resources
PSSW Library
PSSW Library
PSSW Library
System Configuration
Partition Control
Time partitioning
File System Services
Application Process Control
Interpartition communication
Health monitoring
PSSW Module
Pike OS Microkernel
Hardware
Primary Application Task (Pike OS process created by the PSSW module)
Secondary Application Task (created by another application task)

Figure 28: PikeOS architecture diagram.

exception monitoring handlers. PikeOS itself handles hardware interrupts, trace and breakpoint exceptions and system calls by itself and does not forward them to the resource partition exception monitoring handlers. Nevertheless, the kernel is not always capable of handling all faults and passes the fault information to user mode handlers. The exception monitoring handlers assigned to the resource partitions are the entry and exit points for user mode exception handling and are not exchangeable by the resource partition itself. In between the exception monitoring handlers PikeOS offers a short and full exception handler being configurable by the user. The whole process of exception handling is implemented using the IPC mechanisms implemented by the PikeOS microkernel. Thus, upon an exception the kernel generates on behalf of the faulting thread an IPC message, which is sent to the exception monitoring handler of the assigned resource partition and waits for a response of the user mode handler being implemented in either the short or full exception handler. Within each resource partition, a PSSW library is linked in to provide the PikeOS PSSW API to the software encapsulated in the resource partition.

With resource partitions implementing spatial isolation PikeOS supports the remaining requirement of temporal isolation for virtualizing real-time system by providing time partitions. Each PikeOS thread is assigned to a specific time partition. The concept of time partitioning of PikeOS is explained in section 4.2.2.

PikeOS is restricted to paravirtualization of guest OSs. Thus, the used guest OSs need to be adapted to the PSSW library and the interrupt handling mechanisms provided by the PikeOS microkernel. This requires as already mentioned the source code of the guest OS to be adapted, what can be hard to realized depending on the licensing restrictions of the guest OS vendor. [Sys10]

XEN

XEN is an extremely popular open-source VMM for server consolidation and has been developed at the University of Cambridge. The approach of XEN is straightforward in the sense of system virtualization by multiplexing physical resources at the granularity of an entire operating system. One of the top design goals of XEN is to provide high performance virtualization and strong isolation especially for machine architectures not fulfilling the Popek and Goldberg criteria (see 2.2.2) for full virtualization, such as the x86 instruction set. Some examples for the non virtualizability of the x86 ISA are the instruction *popf*, which has different kernel and system mode behavior, the instruction *smsw*, which stores the machine status but does not trap in user mode, and the instructions *sgdt* and *sldt* to manage descriptor table, that also do not trap in user mode. The goal of virtualizing an uncooperative ISA like the x86 ISA (see 2.2.2) is achieved in XEN by the use of a high performance paravirtualization architecture depicted in figure 29. XEN even goes further by proposing high performance paravirtualization for machine architectures fulfilling the Popek and Goldberg criteria, because "completely hiding the

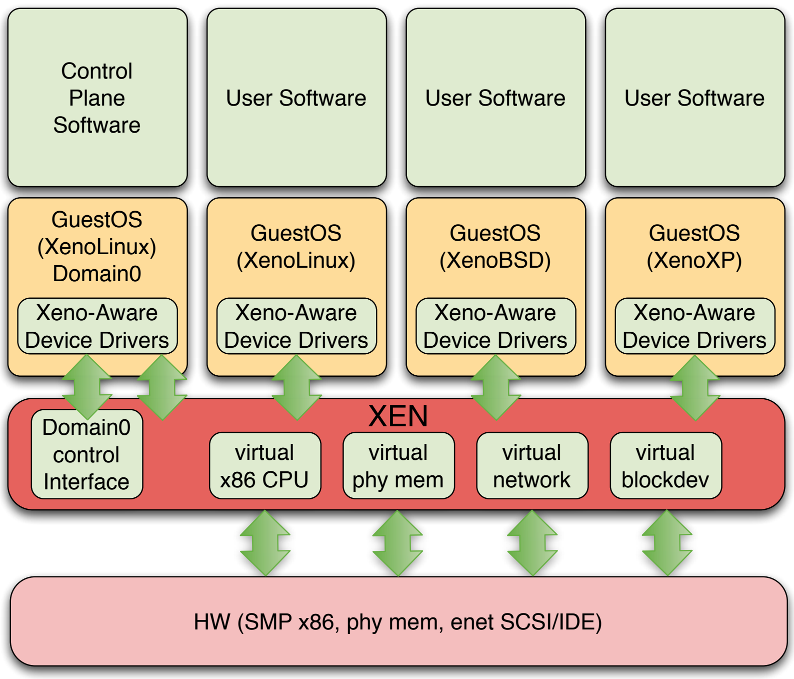

Control Plane Software
User Software
User Software
User Software
GuestOS (XenoLinux) Domain0
GuestOS (XenoLinux)
GuestOS (XenoBSD)
GuestOS (XenoXP)
Xeno-Aware Device Drivers
Xeno-Aware Device Drivers
Xeno-Aware Device Drivers
Xeno-Aware Device Drivers
XEN
Domain0 control Interface
virtual x86 CPU
virtual phy mem
virtual network
virtual blockdev
HW (SMP x86, phy mem, enet SCSI/IDE)

Figure 29: XEN Architecture

effects of resource virtualization risks both correctness and performance [BDF+03]". While this may be true for performance it is not true for correctness as Popek and Goldberg have shown in [PG74], because when the set of sensitive instructions is a subset of the privileged instructions a virtual machine monitor holding the efficiency, resource control and equivalency properties can be constructed (see also section 2.2.2). Nevertheless, XEN is very popular as it provides very good performance on x86 architectures and very inspiring approaches especially for virtualization of devices other than the CPU [BDF+03].

As XEN is based on paravirtualization two mechanisms are introduced to realize control interactions between XEN and a XEN Domain:

- Hypercalls
- Events

The hypercall interface of XEN is responsible for performing a control transfer from a XEN domain to the XEN VMM by using a synchronous software trap to perform privileged operations analogous to the use of system calls in conventional operating systems.

The event interface provides asynchronous communication from XEN to a XEN Domain. The event interface replaces the usual delivery mechanisms for device interrupts and allows the lightweight notification of important events. As a result, pending events are stored in a per-domain bitmask which is updated by XEN before invoking an event-callback handler specified by the guest OS. The callback handler is responsible for resetting the set of pending events, and responding to the notifications in an appropriate manner [BDF+03].

3.2.3 Summary

The related work section showed that there only exist VMMs supporting hard-real time constraints in combination with paravirtualization of the RTOS, which is against the goal of this thesis (see 3.1). Especially the related work for virtualization architectures of embedded systems lacks of existing academic VMM approaches. Only GandalfVMM represents a VMM approach targeted at embedded systems and was developed at the University of Tsukuba by Oikawa et. al. Nevertheless, GandalfVMM does not address any architectural decisions, but focuses more on their mesovirtualization approach, which restricts the paravirtualization overhead to minimal set of instruction necessary to be able to execute the guest OS in virtualized environments. Nevertheless, the approach is restricted to paravirtualization and thus requires the source code of the guest OS to be applicable. Besides this fact, the approach lacks of a discussion on how to decide whether a sensitive instruction may still be executed in the context of the VM without causing an interrupt instead of executing it in the context of the VMM emulating it.

There already exist a few commercial virtualization platforms or VMMs for a range of embedded processors nearly all of them being proprietary systems. All of the available commercial products only use paravirtualization trying to provide reasonable performance and support realtime applications only by the use of dedicated resources. Naturally, this limits the applicability of virtualization using these products to a subset of all possible scenarios. Especially whenever there are applications that cannot be paravirtualized since the source code is not available, these applications cannot be virtualized using the currently available virtualization products since this would require a binary analysis of the whole application which most often is not completely possible.

To sum up, there is no solution given either from academia or even industry which fulfills the goal of having a VMM offering full virtualization to support RTVMs. Furthermore, there does not exist a VMM allowing an arbitrary combination of full and configurable paravirtualization. The idea of having an extensible scheduler interface is also not addressed in the presented related work. Thus, the following part of this chapter addresses exactly those problems to be solved by the introduced VMM design.

3.3 DESIGN

The development of a configurable hybrid VMM for embedded real-time systems is the first topic addressed in this thesis. The task to realize the idea of a configurable hybrid VMM was assigned to Daniel Baldin. His diploma thesis [Bal09] covered the prototypical development. The thesis was supervised by me and the results have been published in [BK09]. Thus some of the parts presented in the following sections and not appearing in [BK09] have first been described in [Bal09]. Sections containing such parts are marked by a reference to [Bal09].

A broad overview of existing VMMs and sophisticated virtualization techniques has so far been given, showing that the actual existing solutions lack the support for hard real-time when using full virtualization. Furthermore, there does not exist a a solution providing a configurable paravirtualization interface, reducing the paravirtualization effort to the really needed paravirtualization hypercalls. The need for an extensible scheduler interface is also addressed by the proposed design.

The approach presented in this thesis overcomes this problem by introducing a configurable hybrid VMM architecture designed and implemented for the PowerPC405 processor which allows for the virtualization of unmodified applications as well as paravirtualized applications or even a combination of both. Support for realtime applications was a major goal of the design, which allows for the integration of any kind of scheduling mechanism for VMs while being completely deterministic. Furthermore, the high configurability allows the system to be optimized explicitly for the intended field of use. This affirmed the research in this direction with hybrid virtualization being a relevant topic in industrial embedded systems.

Besides these advantages, virtualization suffers from its inherent overhead. This overhead is heavily dependent on the ISA and on the hardware support for the emulation process, context saving, memory virtualization and privilege management. The hardware support for virtualization is currently in the focus of several chip designers as Intel with their Intel VT extension and AMD with their pacifica extension. They are currently providing Desktop and Server CPUs with their hardware extensions. With the ATOM processors Intel started the introduction of their virtualization technologies to the embedded market for netbooks. Nevertheless, there is a lack for hardware support of virtualization in the embedded systems domain. Thus, it is very interesting to identify the bottlenecks of virtualization in embedded systems and derive possible cost-efficient hardware extensions for embedded microcontrollers.

In section 3.3.1, the configurability of the hybrid VMM ABI is introduced while section 3.3.2 describes the architecture and the support of full and paravirtualization of the hybrid VMM design. In section 3.3.3, the virtualization concepts of the processor are introduced including especially interrupt handling and register access. To support multiple VMs simultaneously, the VMM needs to implement a scheduling policy. The scheduler interface is described in section 3.3.5. The design does not rely on a single scheduling approach. Instead, it leaves the implementation of the scheduling policy up to the developer by providing an extensible scheduler interface. The support for RTVMs is discussed in chapter 4, as this is a very complex problem on its own. Another very important problem to be addressed in section 3.3.6 is the virtual memory management, because the virtual memory management is essential to guarantee spatial isolation in memory. The presented memory virtualization approach was published in [GK08]. Finally a short overview of the possibilities to realize I/O virtualization is given in section 3.3.7.

3.3.1 Configurability

Achieving minimal overhead as needed for the intended field of use is one of the major design goals of this system. This goal is met by the high flexibility offered through the possibility to configure a wide range of system components. The idea is based on the concept of RTOS configurability introduced by Ditze in [Dit98a, Dit98b].

The configuration of the VMM is realized in a very fine granular structure by the extensive use of preprocessor statements within all VMM components. Thus, the system designer is able to enable or disable features of the virtual machine monitor or specify which parts of the hypercall interface shall be supported. The principle workflow of the configuration is depicted in figure 30. Based on the configuration files, the preprocessor eliminates, adds, or changes code segments inside the implementation files to create source code which does not suffer from unneeded code parts any more. This is a very valuable feature if the platform is about to be deployed inside a very memory limited environment.

The virtualization platform allows the system designer to explicitly define whether there is the need for full-virtualization or paravirtualization or even both. It is even possible to configure the support for special parts of the host ISA as for example support for virtual memory. The hypercall interface may explicitly be configured by defining whether paravirtualized drivers are supported, inter partition communication is needed and which scheduler needs to be used. Each of the components can then be configured as well to allow the platform to match the needs of the target system to a maximum amount. An example for a hybrid configuration supporting full and paravirtualization is depicted in figure 31. [BK09]

3.3.2 Architecture

In order to meet the requirements of high performance and configurability the virtualization platform provides a hybrid VMM interface as already depicted in figure 31. Therefore, the VMM is implemented as a hybrid VMM which uses full-virtualization as the basic virtualization technique with additional support for

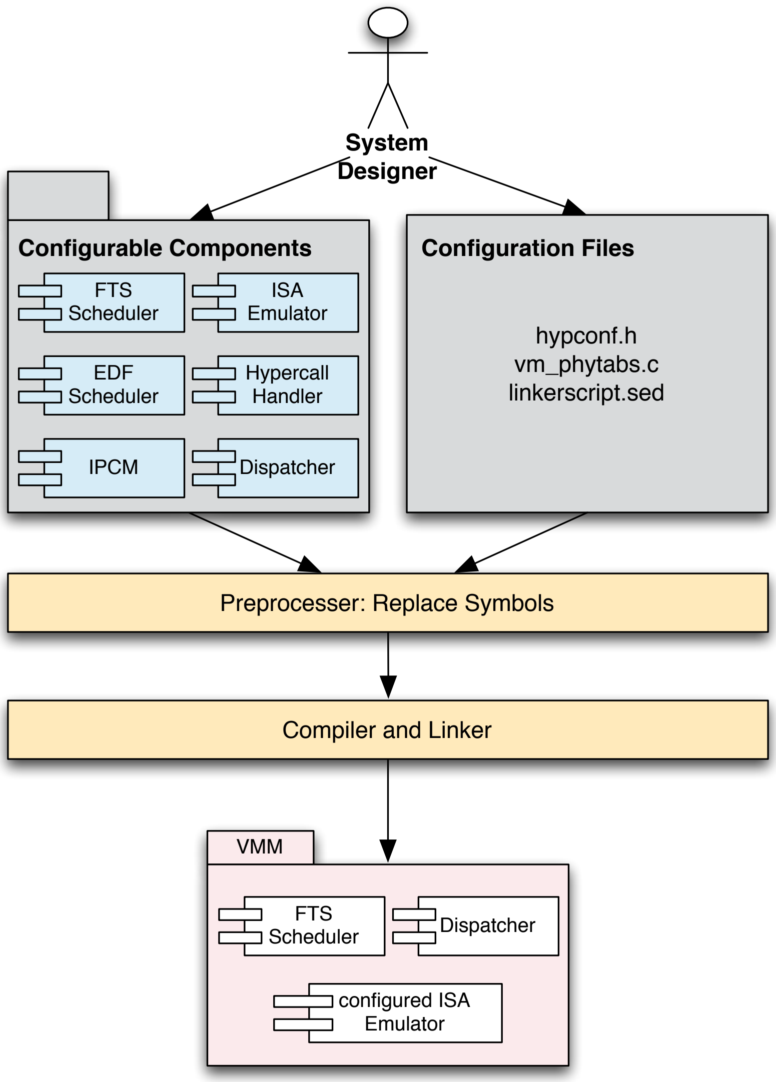


Figure 30: Configuration flow of the virtualization platform.

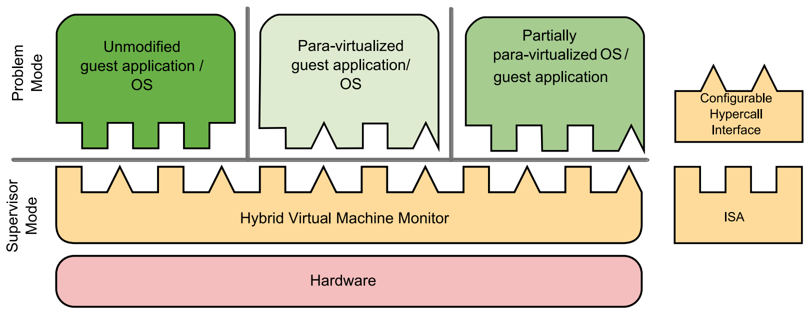

Problem Mode
Supervisor Mode
Unmodified guest application / OS
Para-virtualized guest application/ OS
Partially para-virtualized OS / guest application
Configurable Hypercall Interface
Hybrid Virtual Machine Monitor
ISA
Hardware

Figure 31: The virtual machine monitor allows for the virtualized execution of any kind of guest application. Left: an unmodified application. Middle: a completely paravirtualized application. Right: a partially modified application.

paravirtualization. Partially paravirtualized applications or self-modifying applications are also supported by the use of a fall-back feature which ensures the consistency while executing in and switching between both kind of states. The configurability of the VMM and the paravirtualization interface allows the system designer to create a system which is tailored for the particular field of application and saves as much memory as possible.

In general, the design is based on the multiple independent levels of security (MILS) architecture [Obj08]. As illustrated in figure 32, only the minimal set of components which are needed to implement the secure partitioning, scheduling and communication between VMs are part of the VMM running in privileged mode. All other components called "Untrusted VMP Modules" are placed inside a separate partition and are executed in user mode as this is one of the fundamental concepts of the MILS architecture. The communication between partitions is controlled by the Inter Partition Communication Manager (IPCM) who will on request create shared memory tunnels between two partitions which can be used for VMs to ease the communication between each other. Especially this feature allows formerly physically spread systems that had to use a real-time capable bus system (e.g. a CAN-bus) for communication to enhance the security, robustness and performance of the information flow between each other if they are virtualized and placed on top of the same VMM.

The fundamental components of the whole system are formed by the interrupt handlers as seen in figure 32. Since the VMM executes the guest application in user mode, any interrupt oc-

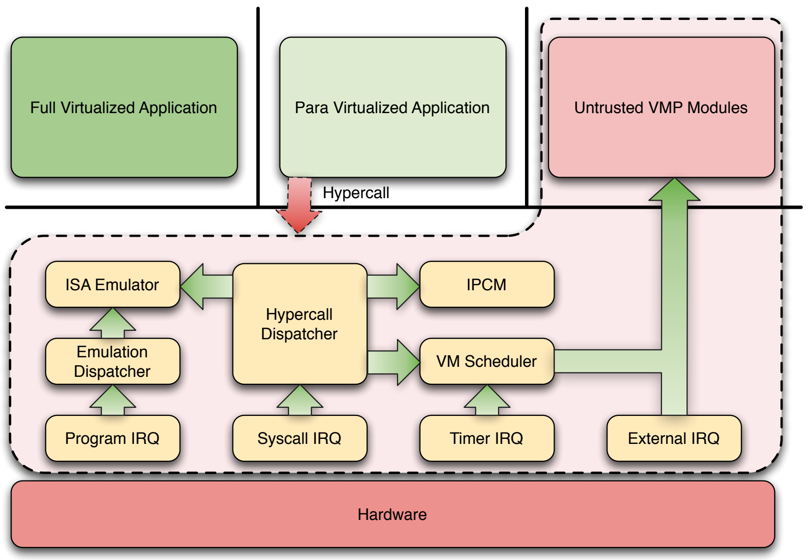


Figure 32: Information and control flow of the components used inside the virtualization platform.

curring is delegated to the VMM first which then has to analyze the interrupt and forward it to the appropriate component or the virtual machine itself. Program interrupts raised amongst others by privileged instructions inside the guest application are forwarded to the emulation routine dispatcher which will determine the corresponding ISA emulation routine. Whenever the virtualization platform has been configured to support paravirtualization, applications running inside a virtual machine can use hypercalls to emulate privileged instructions, call the IPCM, use paravirtualized I/O drivers or call scheduler related functions. Especially the last component offers methods to set scheduler related parameters at runtime or the possibility to yield the cpu directly in order to allow system designers to incorporate special scheduling mechanisms. [BK09]

Full Virtualization

The full virtualization components are formed by the already described program IRQ Handler, the Emulation Dispatcher and the ISA Emulator (see figure 32). A flow chart is depicted in figure 33 to visualize the following description of the involved components. The emulation dispatcher is responsible for the analysis of the program IRQ and the dispatching to the associated emu-

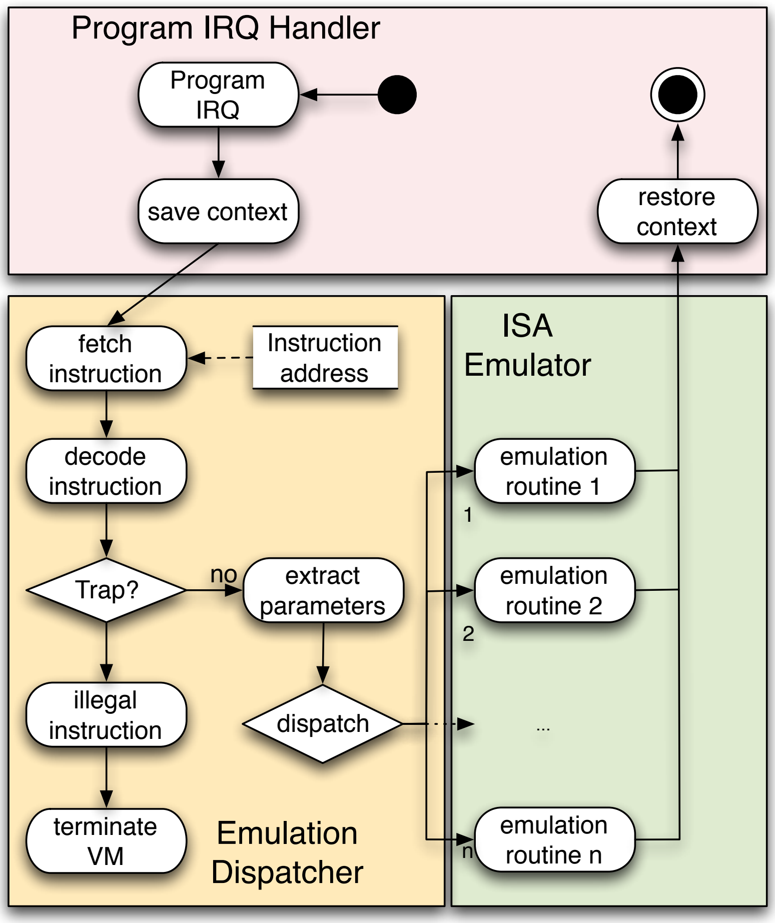


Figure 33: Flow Chart of the Full Virtualization components.

lation routine of the ISA emulator. The set of emulation routines is configurable by the configuration process introduced in 3.3.1. Thus, not needed emulation routines can be removed from the final VMM binary. The emulation process is based on the basic interpretation approach presented in 2.3.1, as techniques like dynamic binary translation are not suited for the use in real-time systems due to their runtime overhead. The developer always has to keep in mind that the emulation of a sensitive instructions needs to be bounded in its execution time. The first step of the emulation process is to fetch the instruction to be emulated. This step depends on how the hardware provides the source instruction that caused the trap. In case of the Power ISA, the address of the instruction is stored in a reserved register. Thus, the address needs also to be fetched from memory before it can be decoded. The decoding step identifies the instruction and its parameters and passes them to the dispatching routine which calls the associated emulation routine for the identified instruction with its parameters. The decoding step could also be performed offline when the hardware provides the address of the trapping instruction. This is known as pre-decoding [SN05b] and is closely related to binary translation (see 2.3.2). A pre-decoded copy of the binary is saved in memory having at least the same size as the original binary. The pre-decoded version of the binary contains the already extracted instruction and their parameters in easily accessible fields what reduces the decoding step to load instructions only. Nevertheless, the offline pre-decoding approach suffers from the code-discovery problem like binary translation, making it only usable in special cases. To overcome the code-discovery problem, pre-decoding could also be applied online, which is known as incremental pre-decoding. In this case, the pre-decoding is performed with the knowledge of the current program flow from the actual program location up to the point where an indirect branch or jump is performed, as then, the target location within the binary code is not known in advance.

Paravirtualization

The syscall IRQ handler and the hypercall dispatcher build the paravirtualization part of the hybrid VMM design. The first part of the paravirtualization covers the emulation of sensitive in-

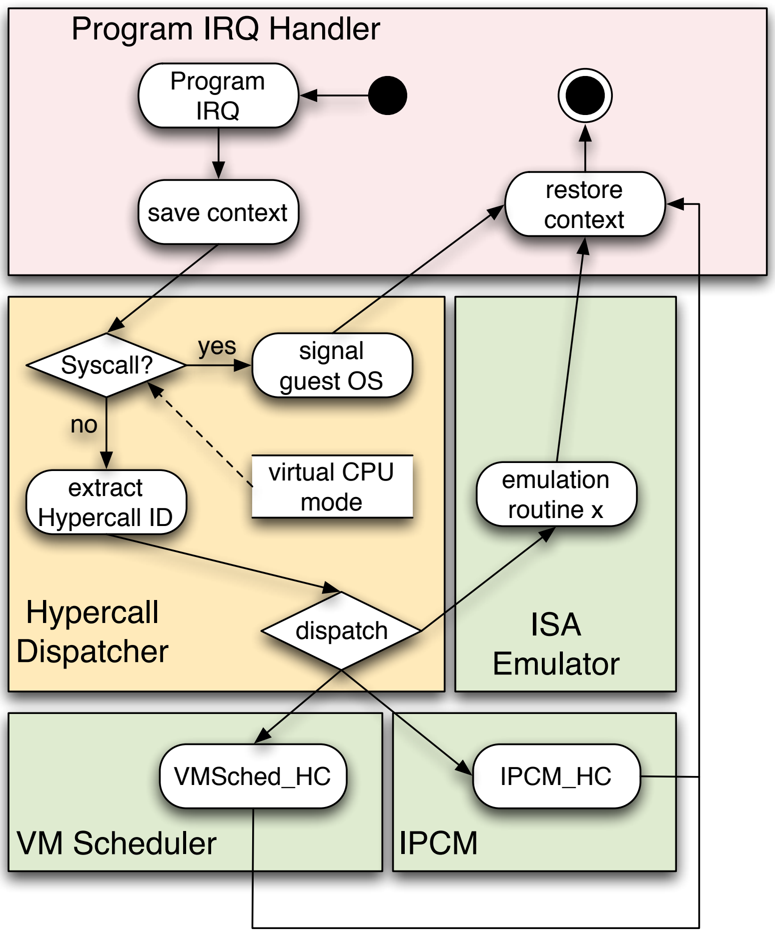
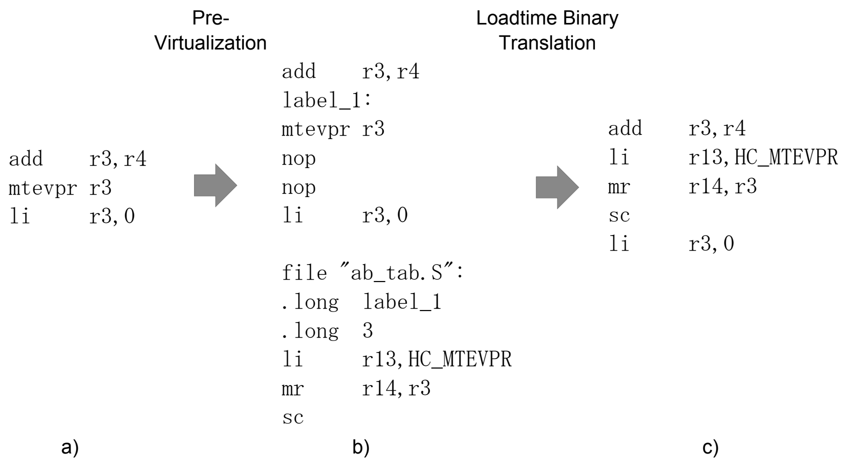


Figure 34: Flow Chart of the Paravirtualization components.

structions to speed up the trap and emulate approach used in full virtualization mode. Therefore, the hypercall provides all information needed in easy accessible fields to minimize the decoding step in the hypercall handler. Thus, implementing this kind of hypercalls in a guest OS can be considered as manually annotated pre-decoding. The second part of the paravirtualization adds additional functionality to the system ABI by providing high-level hypercalls to control the behavior of the VMM Components such as the scheduler, the IPCM and other components that may be added using the configurability of the hybrid VMM approach. A very important requirement of the hypercalls is their bounded execution time, what has to be kept in mind upon implementation always.The flow chart of this two paravirtualization parts is depicted in figure 34. When the syscall IRQ handler detects a hypercall, the hypercall dispatcher is invoked, while the syscall is signaled to the guest OS and the context is restored when the syscall IRQ handler detects a syscall from a user program to its guest OS. For the decision whether a syscall IRQ is a syscall or a hypercall, the virtual cpu mode of the executed VM is needed (see 2.2.1). The hypercall dispatcher can easily decode the hypercall as it is passed using easy accessible fields. The representation of these fields depends on the hardware architecture as they may be passed in special registers, scratchpad memory and so on. After the hypercall has been decoded, the hypercall dispatcher can call the associated hypercall handler. As already described this can be an emulation routine or a high-level component like the IPCM, the scheduler or even an untrusted I/O driver encapsulated in userspace .

A big problem of paravirtualization is that up to now, no method exists to perform paravirtualization automatically using special tools on a given binary. An approach in that direction is called pre-virtualization and has been shortly presented in 3.2.1. Pre-virtualization parses unlinked binaries and replaces sensitive instructions by hypercalls. The technique is based on the idea to append "no operation" (nop) instructions to every privileged instruction inside the guests applications source code in order to create space for a later replacement of these instructions by hypercall instructions to the virtual machine monitor. The overall process is illustrated exemplarily in figure 35. To support pre-virtualization, a python script, which first parses the source code of an application for privileged instructions, has been im-



```
add     r3,r4
mtevpr  r3
li      r3,0
```

```
add     r3,r4
label_1:
mtevpr  r3
nop
nop
li      r3,0

file "ab_tab.S":
.long   label_1
.long   3
li      r13,HC_MTEVPR
mr      r14,r3
sc
```

```
add     r3,r4
li      r13,HC_MTEVPR
mr      r14,r3
sc
li      r3,0
```

Figure 35: Pre-Virtualization illustrated. a) Base source code. b) Pre-Virtualized source code and Pre-Virtualization table. c) The final paravirtualized code.

plemented for the hybrid VMM. Whenever a privileged instruction is found, its address is stored inside a pre-virtualization table together with the binary code of the hypercall instructions. The code itself is only modified by appending nop instructions to the privileged instructions found. Therefore, the intermediate pre-virtualized code can still be executed natively. At load-time, the virtual machine monitor finally paravirtualizes the pre-virtualized applications by the use of the pre-virtualization table which is stored within the applications data area. As the hypercalls typically comprise more than one instruction, the addresses within the binary are shifted. This is the reason why pre-virtualization can only be applied to unlinked binaries, as it is then possible to adjust all addresses automatically. Nevertheless, it is worth adding this feature to the hybrid VMM design as it does not increase the complexity of the VMM, since it is based on automated preprocessing steps and helps to automate paravirtualization, because the manual paravirtualization can be quite hard, because the guest OS has to be adapted completely to match the hypercall ABI. Considering a Linux version to be executed with XEN requires the modification of about 3000 lines of code [BDF+03] what is close to a complete VMM like Lynxsecure with about 8000 lines of code.

3.3.3 Processor Virtualization

This section describes the core components handling the processor virtualization. The processor provides different interrupts being the entry points to the VMM, thus IRQ Handling and dispatching is addressed in section 3.3.3. Besides this the access to the processor registers needs to be controlled by the VMM. In the case of full virtualization, this is handled within the VMM upon an interrupt signaling the access to a register by a sensitive instruction. This causes a lot of overhead due to the interrupt handling mechanisms. When it is possible to apply paravirtualization, a more sophisticated approach can be applied. This approach is presented in section 3.3.3 .

IRQ Handling and Dispatching

The IRQ Handlers are the heart of the VMM design. They ensure the proper activation of the VMM in case of an interrupt and form the only entry point to the VMM. The design is based on four types of interrupts.

- Program IRQ: A Program IRQ is raised when the execution of an illegal instruction or an attempted execution of a privileged instruction from user-mode traps. This IRQ triggers the full virtualization components of the hybrid VMM design.
- Syscall IRQ: A Syscall IRQ is raised when a syscall instruction is executed to activate to signal a request to the system software. This IRQ triggers the paravirtualization components of the hybrid VMM design.
- Timer IRQ: A Timer IRQ is raised when the timer device of the hardware has reached its requested value.
- External IRQ: External IRQs are raised by external devices like I/O devices.

The task of the interrupt handlers is to save the context, analyze the interrupt, dispatch the interrupt to the appropriate component of the VMM, handle the interrupt and return to the preempted program. Figure 36 shows this sequence after a privileged instruction trapped. The time t_s represents the period of time needed by the context saving routine, t_a represents the time to identify the source of the interrupt, t_d represents the time

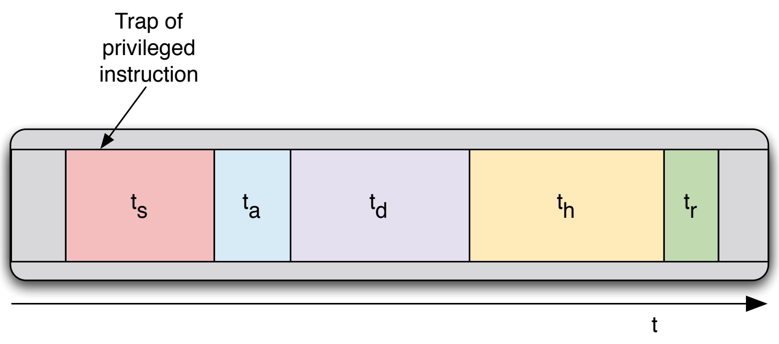
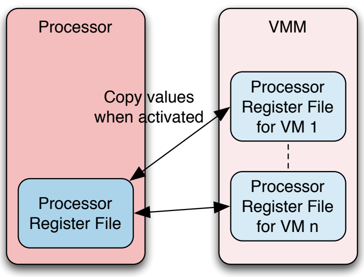


Figure 36: Steps performed upon handling an IRQ [Bal09].

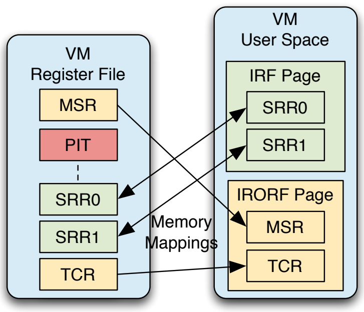
needed by the dispatching routine, t_h represents the required time to perform the handling of the interrupt and t_r represents the time needed to restore the context of the preempted program. The time t_s heavily depends on the hardware architecture and usually takes most of the time of an interrupt handler. Thus, the efficient implementation of this part of the interrupt handler is crucial for the performance of the whole VMM. The next part of the interrupt handler analyzes the interrupt for the reason of the interrupt. The analysis heavily depends on how the information on the interrupt cause is represented in hardware. The timer interrupt is an example for this case as it is only raised by a single possible event. When there is a dedicated interrupt line for a special event, no time is needed to identify the source of the interrupt reducing the time needed by the dispatching routine to zero, as the associated handler can be directly called, which is the case for the timer IRQ. In the case of a shared interrupt line, the analysis routine can become very complex, which is the case for the Syscall and the Program IRQ.

The program IRQ is the central entry point for the full virtualization part of the hybrid VMM design. Besides the case of the execution of an illegal instruction, it is raised when a privileged instruction traps in user mode. As already mentioned before, the dispatching process to the associated emulation routines can become very complex, because the source instruction that triggered the program IRQ has to be identified. The complexity heavily depends on the information provided by the hardware upon a triggered program IRQ.

The syscall IRQ is the central entry point of the paravirtualization part of the hybrid VMM design. Whenever a paravirtualized


Processor
VMM
Copy values when activated
Processor Register File for VM 1
Processor Register File
Processor Register File for VM n

(a) Processor Virtualization by using Register Files.



(b) Innocuous Register File Mapping. Temporarily innocuous register are mapped into pages inside the VM memory space.

VM wants the VM to service a hypercall, it triggers a syscall to activate the VMM. As the syscall IRQ is typically used by user-mode programs to activate the operating system the VMM must distinguish between hypercalls to the VMM and syscalls to the guest OS. This distinction is quite easy, as a hypercall performed by the guest OS is executed as a syscall in virtual supervisor mode, while a regular syscall of a user program is executed as a syscall in virtual user mode (see 2.2.1) [Bal09].

Innocuos register file mapping

Virtualizing the processor is one of the main tasks of the virtual machine monitor. Every virtual machine gets its own set of virtual register stored inside the Register File as depicted in figure 37a. Whenever the virtual machine is activated, the registers are copied into the registers of the physical processor. During the execution, the virtual machine has unrestricted access to the user space registers of the processor. Registers that need to be accessed by the use of privileged instructions will trap to the virtual machine monitor which then emulates the behavior of that instruction on the virtual register inside the register file. However, there exist registers which can be accessed without having immediate influence on the state or behavior of the virtual machine. An example is given by the Save Restore Registers (SRR) of the PowerPC Processor. The value of these registers will only be used whenever the processor executes the "recover from interrupt" (rfi) instruction. In order to speed up the access to these

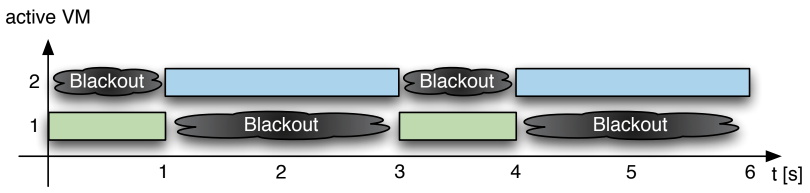


Figure 37: When sharing the timer device, the VMs are suffering from blackouts during their phase of inactivity [Kai08a].

kind of registers, Proteus uses a technique called Innocuous Register File Mapping (IRFM) which allows virtual registers to be mapped to memory pages inside the VMs accessible memory area as illustrated in figure 37b. Virtual registers which may be read directly without the need of any emulation are mapped into a read only memory page. Registers that can be either read or written without immediate influence are made accessible inside a readable and writable memory page whose address can be configured to conform to the systems memory map. Therefore, access to these registers is possible by using load and store commands instead of trapping to the virtual machine monitor which speeds up the virtualized execution of a program dramatically. A similar approach has been used inside the ia64 port of the Xen VMM [BDF+03] for desktop systems to speed up the information flow between the VMM and the VMs. However, the approach has not been applied completely to all possible registers of the ISA like this virtualization platform does.

Since the IRFM feature can only be used by paravirtualized applications partially modified applications may still use privileged instructions to access the virtual registers. In order to ensure the consistency of the mapped registers and the virtual registers inside the register file, the virtual machine monitor updates the mapped registers whenever a value is written to the register file. This feature is called the Innocuous Register File Mapping Fallback Feature and may be enabled if the system designer is unsure whether all privileged instructions are paravirtualized or not. Especially for systems using third party libraries without access to the source code, this is a very valuable feature [BK09].

3.3.4 Timer Virtualization

In general, the hardware systems today offer a timer register and a so-called programmable interval timer (PIT). The time is represented as system ticks being periodically executed with a specific frequency f. Thus, the time T represented by a single system tick is $T = \frac{1}{f}$. When an interrupt is needed after a certain time has expired, the PIT can be used. Upon request, the PIT is filled with a given value and is decremented by one upon every system tick. When the value reaches zero, a timer interrupt is triggered. The virtualization of a timer device is a quite complex problem, as the timer device represents the notion of time of a system and has to be shared among different VMs. During this sharing, the inactive VMs suffer a blackout (see figure 37), as the timer device continues incrementing at the rate f. In the case of paravirtualization, the virtual machine is aware of experiencing blackouts while in full virtualization, the VM is not aware of these blackouts. This may lead lead to undeterministic behavior, but is not necessarily a consequence. To ensure the correct timing behavior of full virtualized VMs, the timer virtualization must hide the blackouts.

As a scheduling mechanism for full virtualized VMs/RTVMs, a fixed time slice scheduler (FTS scheduler) is assumed, as it allows an easy implementation of proportional share distribution among different VMs and is the base scheduler used in chapter 4. In figure 38, such a schedule is shown. The x-axis denotes the time passed in the hardware timer register given in host cycles. The y-axis denotes the number of cycles supplied to the VM in virtual cycles. The solid line represents the real execution, while the dashed line represents the idealized execution assuming the supply could be assigned linear by using infinite timeslicing. When considering the example of VM_1, depicted in figure 38, being scheduled with VM_2, depicted in figure 39, one can see that the VM_1 is active periodically from time $S_1 = 0ms$ to time $E_1 = 1ms$ relative to the beginning of the time period $P = 3ms$, and VM_2 is active from time $S_2 = 1ms$ to time $E_2 = 3ms$ relative to the beginning of the time period $P = 3ms$. Both VM receive a share of $\frac{E_i - S_i}{P}$ of the processor. In case of VM_1 this share is equal to $\frac{1}{3}$ of the processors capacity, while in case of VM_2, this share is equal to $\frac{2}{3}$.

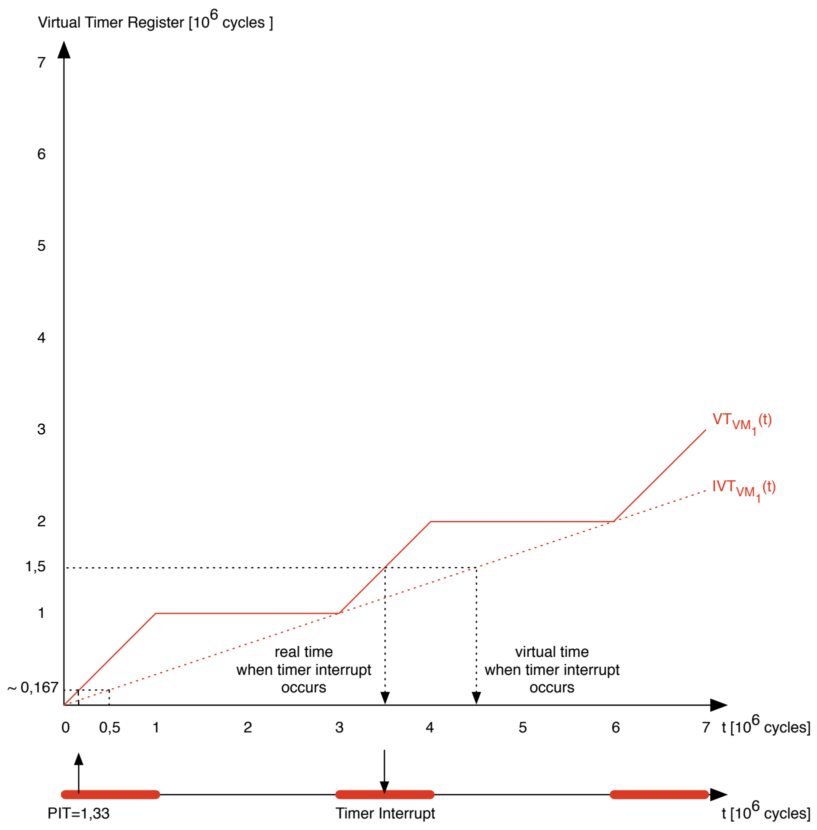


Figure 38: Timer scaling in case of full virtualization

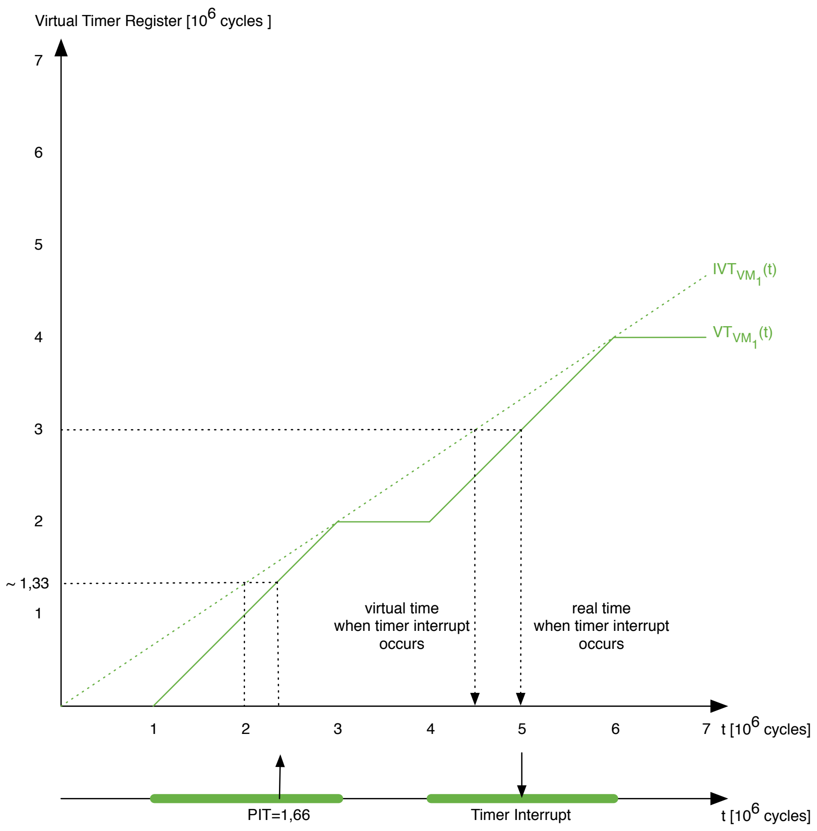


Figure 39: Timer scaling in case of full virtualization

A very simple approach to virtualize the timer without blackouts would be to substract the cycles passed during the blackout from the virtual timer register of the virtual machine. The virtual timer register may then be multiplied with the inverse of the share being $\frac{P}{E_1 - S_1}$, being 3 for VM_1 and being 1.5 for VM_2, to receive the real cycles passed in the hardware timer register to achieve this supply. This implies the virtual cycle supply being ideally given as linear function called $ICS_{VM}(t)$ (Ideal Cycle Supply), where t is given in host cycles. As already mentioned before, this function is depicted by the dashed line in figures 38 and 39.

$$ICS_{VM_i}(t) = \frac{E_i - S_i}{P} \cdot t, \tag{3.1}$$

One could think of also multiplying the virtual timer value by $\frac{P}{E_i - S_i}$ to be passed to the hardware PIT register upon a PIT request, but this would result in an incorrect value for the PIT, as the result of multiplying the virtual timer value by $\frac{P}{E_i - S_i}$ is the virtual time expressed in host cycles and is called t_{ICS}. The time t_{ICS} can be seen as the virtual time running contiguously in the VM. The real cycle supply to the VM is given as a piecewise function called $CS_{VM_i}(t)$, as the VMs are executed in fixed time slices with the given processor frequency f and thus the time t represents the real time passed when the virtual timer register has reached the supply of $ICS_{VM_i}(t_{ICS})$ at the virtual time t_{ICS}. Thus, the real time t has to be passed to the hardware timer register. The value of t can be determined by the intersection of $CS_{VM_i}(t)$ with the constant function being equal to the current virtual timer value.

Thus, the time t of $CS_{VM_i}(t)$ is mapped to the time t_{ICS} of $ICS_{VM_i}(t_{ICS})$ where both supply functions are equal:

$$t_{ICS} \to t, ICS_{VM_i}(t_{ICS}) = CS_{VM_i}(t) \tag{3.2}$$

This preserves the correct timing behavior based on the cycle supply given by a FTS schedule. The mapping can be realized by first determining the cycles Δt passed since the VM has been active. This is the time a since the VM has got last activated substracted from the current real time t.

$$\Delta t = t - a \tag{3.3}$$

Then, the time PIT_{VM} of the requested PIT interrupt relative to the last activation of the VM is determined by adding the cycles supplied by $CS_{VM}(\Delta t)$, as these numbers of cycles have already been supplied since the last activation of the VM:

$$PIT_{VM_i} = PIT + \Delta t. \tag{3.4}$$

Now PIT_{VM_i} contains the cycles to be supplied to the VM. As the PIT register of the system needs to be set to the real time value when the supply of PIT_{VM_i} is finished by $CS_{VM_i}(t)$ the value of t has to be determined by the VMM. This is realized by first determining the number of full periods x with supply $E_i - S_i$ contained in PIT_{VM_i}.

$$x = \lfloor \frac{PIT_{VM_i}}{E_i - S_i} \rfloor. \tag{3.5}$$

Then the remaining supply R being supplied in period $x+1$ is calculated by:

$$R = x - \lfloor x \rfloor. \tag{3.6}$$

Now it is possible to calculate t_{CS} by:

$$t_{CS} = x \cdot P + S_i + R \cdot (E_i - S_i). \tag{3.7}$$

The value t_{CS} can now be put directly into the PIT hardware register. As can be seen in figures 38 and 39, the real time when the interrupt occurs is different to the virtual time of the VM. This is based on the mapping introduced in equation 3.2. This has to be considered by the developer when defining the task sets for the given VMs under a given FTS schedule.

An example for this is given in figures 38 and 39. The first example of VM_1 shows a task requesting a PIT interrupt in $1.33 \cdot 10^6$ virtual cycles relative to the current time. The current real time when the request is issued is $t = 0.167 \cdot 10^6$ host cycles being equivalent to the virtual time $t_{ICS} = 0.5 \cdot 10^6$ host cycles. The calculation using equations 3.3 to 3.7 results in the real time $t = 3.5 \cdot 10^6$ host cycles.The virtual time t_{ICS} can be calculated using equation 3.1 and results in $t_{ICS} = 4.5 \cdot 10^6$ host cycles as $ICS_{VM_1}(t_{ICS}) = 1.5 \cdot 10^6$ virtual cycles. The time difference of the virtual time when interrupt occurs to the virtual time where the interrupt was requested is equal to $4 \cdot 10^6$ host cycles. This is the expected time passed in host cycles, because VM_1 has a share of $\frac{1}{3}$ what is equal to $1.33 \cdot 10^6$ virtual cycles when the virtual clock would tick with $\frac{1}{3}f$.

The second example of VM_2 shows a task requesting a PIT interrupt in $1.66 \cdot 10^6$ virtual cycles relative to the current time. The current real time when the request is issued is $t = 2.33 \cdot 10^6$ host cycles being equivalent to the virtual time $t_{ICS} = 2 \cdot 10^6$ virtual cycles. The calculation using equations 3.3 to 3.7 results in the real time $t = 5 \cdot 10^6$ host cycles.The virtual time t_{ICS} can be calculated using equation 3.1 and results in $t_{ICS} = 4.5 \cdot 10^6$ host cycles as $ICS_{VM_2}(t_{ICS}) = 3 \cdot 10^6$ virtual cycles. The time difference of the virtual time where the interrupt occurs to the virtual time where the interrupt was requested is equal to $2.5 \cdot 10^6$ host cycles. This is the expected time passed in host cycles, because VM_2 has a share of $\frac{2}{3}$ what is equal to $1.66 \cdot 10^6$ virtual cycles when the virtual clock would tick with $\frac{2}{3}$ f.

With this knowledge, it is possible to design the interrupt handling of the VMM. This design is depicted as flow chart in figure 40. Directly after the timer interrupt was triggered, the interrupt is acknowledged by the VMM. Then, the smallest PIT value is determined to assign the triggered PIT interrupt to an available PIT request from the VMM or an arbitrary VM. If the interrupt was triggered for the VMM, all virtual timer registers are decremented by the value of the expired PIT interrupt, the context of the active VM is saved and the scheduler is called. In the other case, the triggered PIT interrupt is assigned to one of the available VMs. Depending on the auto reload flag, the virtual timer register for the according VM is reloaded. Then, the PIT register is set to the minimum value of all available virtual PIT registers. Afterwards the PIT interrupt is queued for the target VM, because the VMM first has to determine whether the VM has interrupts enabled or not. If the target VM has interrupts disabled, the active VM is resumed directly. If not, the interrupt is flagged in the corresponding virtual register of the target VM. Then, it is checked whether the target VM of the interrupt is equal to the active VM, because otherwise, a preemption may be necessary if this feature is enabled.

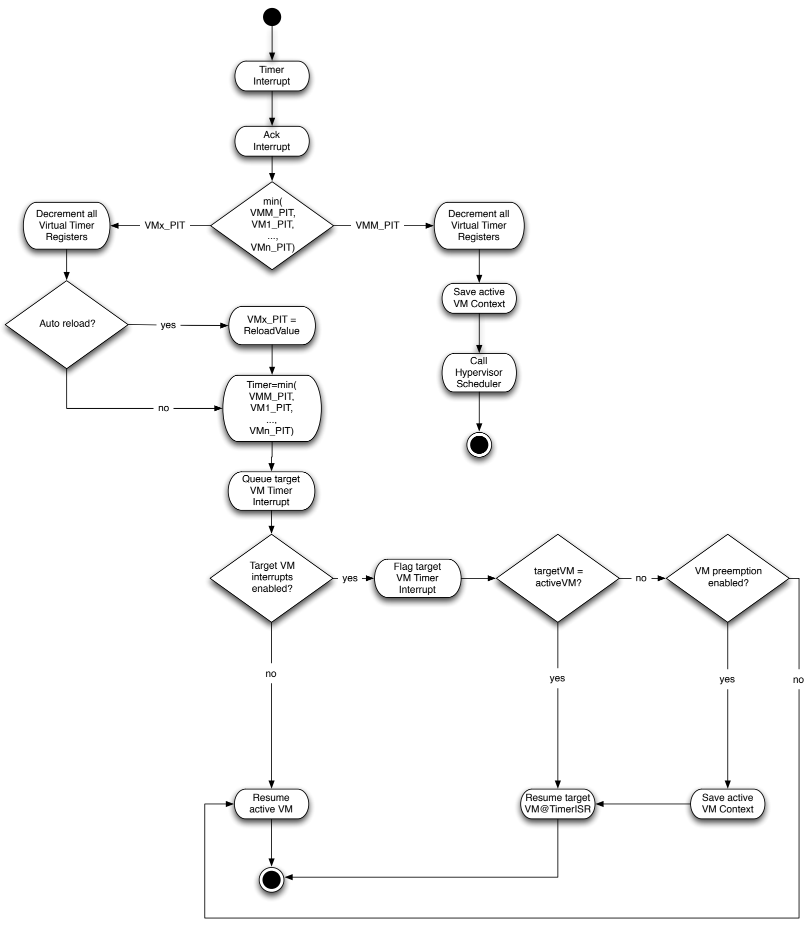

Timer Interrupt
Ack Interrupt
min(VMM_PIT, VM1_PIT, ..., VMn_PIT)
VMx_PIT
VMM_PIT
Decrement all Virtual Timer Registers
Decrement all Virtual Timer Registers
Save active VM Context
Call Hypervisor Scheduler
Auto reload?
yes
VMx_PIT = ReloadValue
no
Timer=min(VMM_PIT, VM1_PIT, ..., VMn_PIT)
Queue target VM Timer Interrupt
Target VM interrupts enabled?
yes
Flag target VM Timer Interrupt
targetVM = activeVM?
no
VM preemption enabled?
no
yes
yes
no
Resume active VM
Resume target VM@TimerISR
Save active VM Context

Figure 40: Timer Virtualization Flow Chart

3.3.5 Scheduler

The scheduler is the main component of the VMM as it multiplexes the CPU among the different VMs. Due to the design decision of a configurable hybrid VMM, the scheduler component implements a uniform interface, which is depicted in figure 41. This interface has to be implemented by every scheduler of the VMM. The method *sched_init* is responsible for the initialization of the scheduler component. This can be for example the initialization of the necessary data structures. The next VM to be executed is determined using the method *sched_getNextVMIndex* which implements the scheduling policy. The activation duration of the VM in processor cycles necessary to fulfill the scheduling task is returned by the method *sched_getNextTimerEvent*. To guarantee that the VMM will regain control of the CPU, the VMM uses this value to program the timer to generate a timer interrupt after this duration. Furthermore, the interface defines two methods useful for paravirtualization. The first method is called *sched_yield* and allows a VM to give back the control of the CPU to the VMM, which is then able to assign the CPU to another VM. This is not possible in the case of paravirtualization, as the VMM is in general not able to notice whether a VM idles or not. The second method is called *sched_setParam* and allows a paravirtualized guestOS to pass different important scheduling parameters, e.g. the next deadline, to the VMM. Figure 41 shows the three basic scheduling approaches currently implemented in the VMM. An in-depth discussion of scheduling VMs will be given in 4 as this is a complex problem especially when applying full virtualization, which is the main focus of this thesis. Nevertheless, a comparison to the paravirtualized scheduling mechanisms will be given to demonstrate the advantages and disadvantages of both approaches [Bal09].

3.3.6 Memory Virtualization

As stated in section 3.2.1, Bennett and Audsley addressed the problem of coexisting virtual memory access time restrictions for real-time and non real-time tasks with their modular page table approach depicted in figure 24. Upon a TLB miss, the TLB Miss Handler Dispatcher performs a look-up on the PCB of the

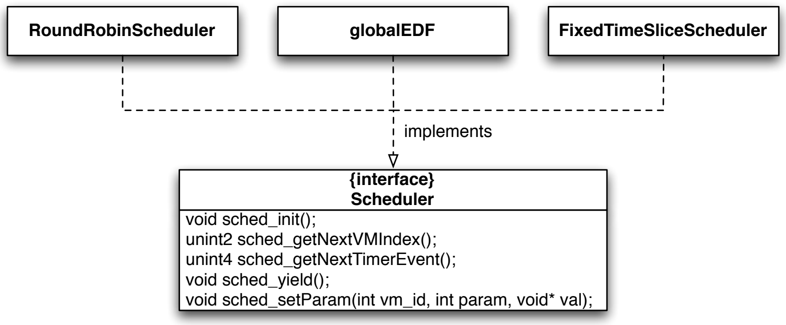


Figure 41: Interface of the VMM Scheduler component [Bal09].

active task to determine the assigned miss handler for this task. The obtained miss handler is called to find the appropriate TLB translation entry within the page table of the process. Due to the fact that different miss handlers can be used, it is possible to use distinct page table architectures for different tasks.

To enable deterministic memory access for hard real-time tasks, Bennett and Audsley propose to use a fixed length array indexed by the virtual page address to find the appropriate translation entry in time $O(1)$. This approach is only appropriate for small logical address spaces as the size of the real map table is proportional to the logical address space. As an example, a 32 bit microprocessor with a minimum page size of 1024 Byte is assumed. In this case, the fixed length array would need about 11.5MB of memory at least. Thus, a solution consuming less memory would be desirable. There are a lot of approaches existing for general purpose OSs like Bit Maps [Tan07], Segmentation [Tan07], Multi-level Paging [Tan07], Short-Circuit Segment Trees [SH02] and Paged Segmentation [Tan07]. A very interesting extension to short circuit trees called intermediate-level skip multi size paging has been proposed by Suzuki et. al in [SS97]. The memory consumption of this approach is superior to the other approaches, and the processing overhead is the same class being also $O(1)$. Nevertheless, the usage of a fixed array would be more performant since the lookup could be realized in a single step while the lookup when using ISMSP requires n steps in the worst case with n being equal to the depth of the tree. Please note that n is user defined and bounded.

The next important step after the translation entry has been found is the replacement of an existing TLB entry with the required entry. There exist a lot of approaches to handle this problem, such as FIFO [Tan07], LFU [Tan07], LRU [Tan07], or the clock algorithm [Tan07]. When performing a WCET Analysis of hard real-time tasks, an upper bound has to be assumed for every memory access to guarantee the task will finish before its deadline. In the general case the number of entries of a mapping table, such as the real map table, exceed the number of available TLB entries. Thus, a TLB miss has to be assumed for every single look-up as the TLB does not cover 100% of the entries. As hard real-time tasks do not benefit from a completion of a memory access before this upper bound, hard real-time tasks do not benefit from having more than one TLB entry. Additional TLB entries would increase the memory coverage and decrease the probability of a TLB miss, but could not increase the hit rate to 100%. In most instances, the memory requirements of a task are too high to make all addresses available in the TLB, assuming a fine granular memory management.

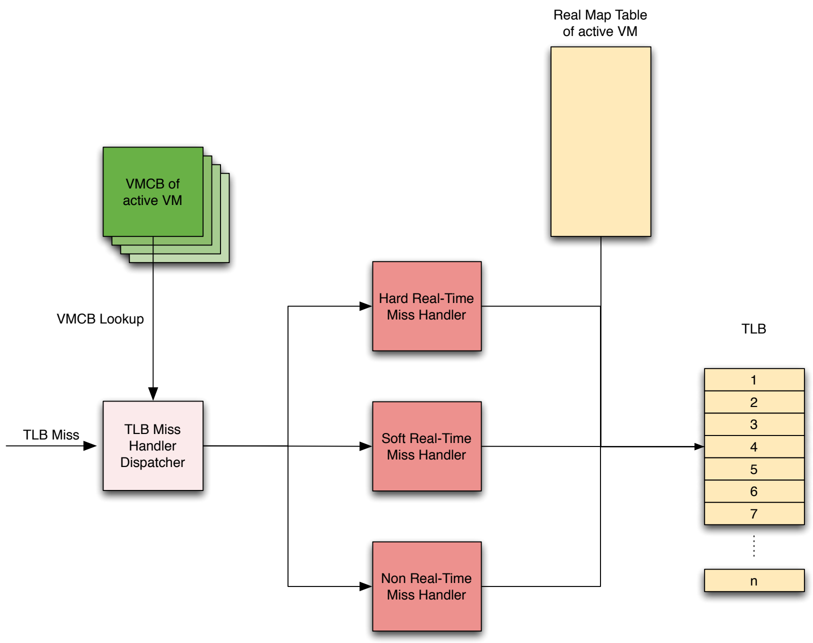

Real Map Table of active VM
VMCB of active VM
VMCB Lookup
TLB Miss
TLB Miss Handler Dispatcher
Hard Real-Time Miss Handler
Soft Real-Time Miss Handler
Non Real-Time Miss Handler
TLB
1
2
3
4
5
6
7
n

Figure 42: Mixed Priority Paging with Dynamic TLB Partitioning

The fact that hard real-time tasks do not take any advantage of more than one TLB entry, a software managed TLB design is used in this thesis to modify the Multiple Page Approach to the approach called Mixed Priority Paging, which is depicted in figure 42. The modification is realized by a static assignment of the first entry of the TLB to all available hard real-time tasks. The remaining space within the TLB is shared between soft and non real-time tasks. Soft real-time tasks and non real-time tasks benefit from having faster memory access times as this results in a shorter response time which may be directly visible for the user. The main problem addressed by this partitioning is the immediate TLB miss caused by the first memory access after a context switch, because the entries within the TLB do not belong to the active task anymore. This effect is known as *cold cache* phenomenon. The goal is to prevent TLB misses after a context switch without slowing down the context switch itself by invalidating the whole TLB and a prefetch of TLB entries to prevent long latencies within context switches. Instead, an exclusively assignment of TLB entries to specific soft and non real-time tasks (*locked entries*) is introduced. This shall prevent TLB misses right after a context switch and keeps the context switch fast by using ASIDs as described in section 2.2.3. The number of locked TLB entries for a specific soft or non real-time task should be changeable dynamically, as the memory access behavior may be highly dynamic during the runtime of the task.

To realize the Mixed Priority Paging with dynamic TLB partitioning for hard, soft and non real-time tasks, a software managed TLB design as introduced in section 2.2.3 can be used. For every miss handler a replacement policy has to be implemented. This is very simple for hard real-time tasks: there exists only one possible entry for replacement which makes the replacement possible in $O(1)$.

In the case of soft real-time tasks, a solution for the following aspects has to be provided:

- TLB locked entry replacement policy
- Overall TLB replacement policy
- Partitioning policy

The replacement policy for the locked TLB entries is the most demanding problem. To prevent TLB misses after a context switch with the help of ASIDs, the page table entries with the highest access probability have to be kept in the TLB as locked entries. For this it is inevitable to know the number of accesses over a certain period of time – or at least a good approximation – to get an idea which pages have a high probability to be used again. The execution time before the scheduler preempts the active process is a good choice for the period of time in which the number of accesses is counted. Just before dispatching the next process, the top n entries are placed in the locked area of size n, if they haven not already been located there. The placement is performed at the end of the execution phase, because the locked entries take effect not before the process gets dispatched again. To prevent a blocking of the locked area by entries that have been accessed very frequently and are not accessed anymore, the access counters of every TLB entry should be reset to 0 before the next process gets dispatched. This ensures that it is possible to adapt to changed memory access behaviors.

The global TLB replacement policy is responsible for the entries that are not locked. These entries can be used by all tasks, even the tasks that have exclusively assigned TLB entries. For the area of locked entries, the most frequently used entries are chosen. If the entries most frequent used are known, it is possible to find out the least frequently used entry. Therefore, it is possible to apply least-frequently-used (LFU) to the globally available entries of the TLB. This has the nice side-effect that an entry that has been used quite frequently and may have experienced a small period in which it has not been accessed, will not be replaced.

The next step is to find out how to manage the number of locked entries for a specific soft or non real-time task. The best metric to achieve this is the TLB miss ratio TLB_{MR}, which represents the percentage of all memory accesses that lead to a TLB miss. Based on the TLB miss ratio, two thresholds γ_{inc} and γ_{dec} for each soft and non real-time task are introduced. They are checked right after the process is preempted. The number of locked entries is incremented by 1 if $\mathrm{TLB}_{MR} > \gamma_{inc}$ and decremented if $\mathrm{TLB}_{MR} < \gamma_{dec}$. The incrementation by 1 prevents a task from grabbing all available TLB entries for its locked area without giving other tasks the possibility to increase their number of locked entries. The check is performed at the end of the execution phase. Another important question as regards to the partitioning policy is which tasks should be able to lock TLB entries. A partitioning scheme in which only small areas (i.e. one or two entries) are locked for a high number of tasks should be prevented. This would lead to a fully partitioned TLB with only a very small number of available slots for each task. Therefore, it is proposed to allow only tasks the locking of TLB entries which heavily depend on memory accesses, as they would benefit most from having locked entries. Furthermore, it is important to limit the total number of locked entries, preventing a heavy performance loss for tasks that are not allowed to lock TLB entries. There always has to be at least one globally available slot – otherwise, address translation would not be possible for tasks without locked entries [GK08].

3.3.7 I/O Virtualization

The IPCM is a very sensitive part of the hybrid VMM architecture, as it allows to communicate between different spatially isolated VMs. Therefore, the IPCM provides a hypercall called *ipcm_create_tunnel* that creates a shared memory region accessible by both VMs only using the memory virtualization provided by the underlying architecture. As already stated in section 2.2.4 the information flow control policy is very important as the policy defines which VMs may communicate directly using the shared memory provided by the IPCM. When using an unprotected shared memory region for the communication of two VMs the advantage is that the VMs have to implement the communication protocol being used reducing the complexity of the VMM. Nevertheless, this is problematic from the point of view of the spatial isolation provided by the VMM as the VMM is not in control of the performed interactions. Thus, the VMM cannot prevent bad behavior leading to unintentional interactions on this communication channel. To secure this channel against bad behavior the VMM can introduce a protocol to be used. An interesting approach is to use the I/O ring concept introduced by XEN [BDF+03]. The big advantage of the I/O ring concept is the ability to en- and dequeue multiple items at once to reduce the hypercall overhead.

Besides the capability of direct Inter-VM I/O the VMM needs to support device virtualization. To realize this two concepts are applied:

- Dedicated I/O using memory mapped I/O (MMIO)
- Shared I/O using Hypercalls

The concept of dedicated I/O using MMIO is suitable for full virtualization. The memory area of the device is mapped into the address space of the associated VM. Thus, the VM has full access to the device itself. The problem of this approach is that the sharing of devices between multiple VMs is not possible at all. Thus, hosting multiple VMs using I/O devices requires a dedicated I/O device for each VM. The advantage of this approach is that the VMM does not need to handle any trapped instruction or hypercall reducing the virtualization overhead to zero.

Sharing an I/O device between multiple VMs is much more complex. Considering the case of MMIO devices being shared between multiple full virtualized VM this can be very tricky to handle, because the virtualization needs to be handled at the I/O operation level itself. The VMM needs to provide a memory mapping for the MMIO device registers to each VM using the device. This comes at the cost of extra TLB slot usage and the problem of monitoring access to this memory mappings. When providing the memory area as writeable the VMM has no chance to monitor the state of the device. This can easily result in inconsistent states of the device when multiple VMs access the memory area without synchronization. The synchronization of the MMIO area is not trivial since the write access has to be granted to a single VM and revoked after the VM has finished its access to device and the device is ready for a new access. The problem for the VMM is to monitor when the VM has finished its access, because once the VMM has granted write access to the MMIO area the VMM is not able to monitor the memory accessing instructions of the MMIO area violating the required resource control property presented in 2.2.2. Thus, the VMM is not able to notice when it can revoke the access to the device from the VM. To enable the VMM of monitoring the MMIO area it needs to be write protected. In this case every memory access to the MMIO area traps to the VMM. The VMM is then able to emulate these memory accesses using a virtual device data structure for each machine. When the VMM detects that a request is completely performed on the virtual device it can enqueue this request to the real device. This requires the integration of virtual devices into the VMM and makes the VMM much more complex. Besides the complexity of the VMM the virtualization overhead at the I/O operation level increases dramatically. Actually research focuses more in the direction of I/O virtualization at the level of hardware support [BYMK...06, UNRS05] and the device driver level [EPF06] or even combinations of both [RS07]. [Bal09]

3.3.8 Summary

The presented design was the first design of a hybrid VMM and was published in [BK09]. It supports the full virtualization of unmodified guests, the execution of paravirtualized guests and a combination of both where the full virtualization is realized as a fall back mechanism when no hypercall is available for the required action. Thus, the goals of a configurable ABI and an extensible scheduler interface, defined in section 3.1, have been achieved while fulfilling the constraints of providing full virtualization and real-time support.

The configurability is provided by the intensive usage of preprocessor statements which allow for a fine granular configuration of all VMM components. The configurability also allows the exchange of scheduling policies by providing a special scheduler interface. The need for spatial isolation, which is essential for preventing unintentional interactions, is guaranteed by the MILS-based architecture. The core functionality of the VMM is intentionally kept very small to enable the verification of the VMM. Additional untrusted functionality can be shifted into user space.

The need for high level APIs and high performance is addressed by providing a configurable hypercall interface for the underlying ISA which also includes the IRFM feature, allowing access to most of the ISA registers by simple paravirtualized memory accesses. To enable semi-automatical paravirtualization, previrtualization is implemented as a preprocessing python script allowing for a complete automatic paravirtualization if the unlinked object files of the guest is available.

To fulfill real-time requirements, the design always considered determinism by bounded execution times. The virtual memory management of architected TLBs has been specially optimized in this case. The mixed priority paging partitions the TLB to favor soft and non real-time tasks instead of hard real-time tasks, as it is in general not possible to guarantee a 100% coverage of TLB entries to page table entries.

The introduced extensible scheduling interface is defined at an quite abstract level to keep it generic. For an information flow from the VM to the VMM, the method *sched_setParam* has been introduced, which allows for the passing of arbitrary data by memory pointers to the VMM. Thus, the scheduler of the VMM and the scheduler of the VM need to be adopted for this communication. Unfortunately, an implementation of this method is not required when focusing on full virtualization. A general discussion about the schedulability of RTVMs is given in chapter 4.

To sum it up the main goals of providing a configurable ABI and an extensible scheduler interface with regard to real-time and full virtualization support have been achieved. The only goal left to be addressed is the determination of the WCET of the virtualized tasks. This will be addressed in the following evaluation.

3.4 EVALUATION

The implementation and evaluation of the design proposed in section 3.3 was the main part of the diploma thesis of Daniel Baldin [Bal09]. The implementation was realized on a PowerPC 405 FX microprocessor. Therefore, at the beginning of this section the virtualizability of the Power ISA is shown in section 3.4.1. The following section 3.4.2 covers the determination of the WCETs of the VMM. With the help of the WCETs of the VMM, it is possible to determine the WCETs of the VMs.

3.4.1 PPC405 ISA Analysis

The basic requirement for supporting efficient full virtualization is the compliance of the Popek and Goldberg theorem by the PowerPC 405 ISA. As classical RISC processor, the PowerPC offers a set of registers, which are only accessible in supervisor mode through dedicated read and write instructions for these registers. In figure 43, the register set of the PowerPC 405 FX is depicted and shows the registers accessible in user mode and supervisor mode. Table 3 shows a short description of a subset of the register available only in supervisor mode by the dedicated read and write instructions *mtspr* and *mfspr* (move to/from special purpose register). When taking an in-depth look at the register set, one can see that within the user model, only registers are readable and writeable by innocuous instructions which do

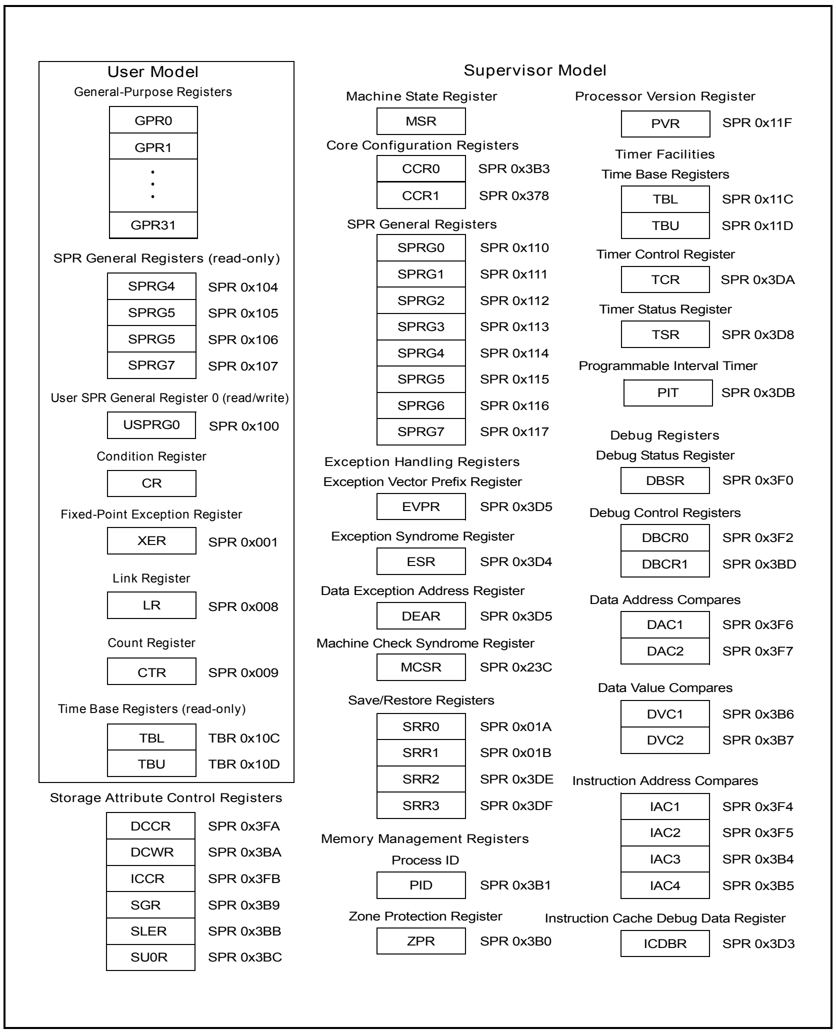


Figure 43: PowerPC 405 register set [Xil09].

not violate the resource control property of am VMM defined by Popek and Goldberg, as all registers managing resources are sensitive instructions available only in supervisor mode. Thus, it is possible to simplify theorem 2.6 for the PowerPC 405 FX ISA. This is done by checking whether all sensitive instructions accessing the registers of the supervisor model are trapping in user mode. This can easily be shown, since all sensitive instructions accessing registers of the supervisor model are privileged and cause a trap when executed in user mode according to the PowerPC 405 FX users manual.

With this knowledge, it was possible to implement emulation routines for all sensitive instructions using the approaches introduced in section 3.3 while preserving the Popek and Goldberg criterions introduced in section 2.2.2 [BK09].

3.4.2 Worst Case Execution Times

As defined in section 3.1, the knowledge on how the WCET is affected by the virtualization is essential for RTVMs to determine the upper bound of virtualization overhead to guarantee the needed determinism for real-time applications and to determine the CPU requirements of the final virtual real-time system. The design presented in 3.3 considered the required determinism by its offline configurability preventing unnecessary runtime overhead and the proposed virtualization techniques to be applied. The determination of the WCET of a VM is divided into two steps. First, the virtualization complexity at instruction level is determined. In general, an instruction is taken from the ISA, which can be a privileged or non-privileged instruction. The set of privileged instruction is called I_p. The WCET of instructions of the set of privileged instructions needs to be determined, because the virtualization induces overhead by the emulation of the VMM, while the WCET of the non-privileged instructions can be determined without considering this overhead. Additonally interrupt events have to be considered as well and are considered as atomic events with a given WCET, because the VMM does not allow for nested interrupts.

$$WCET : I_v \rightarrow \mathbb{N} | I_v \in \{i | i \in I_p \vee i \in IRQ\} \tag{3.8}$$

Register Short Name	Register Long Name	Resource
MSR	Machine State Register	Manages the processor execution modes.
CCR0-1	Cache Control Register	Manages the cache behavior.
EVPR	Exception Vector Prefix Register	Contains the physical address of the offset for the interrupt handlers
SRR0-4	Save and Restore Register	Contains the MSR, PC and other important register values when an interrupt occurs. When returning from an interrupt, the values of the register can be restored by setting the associated SRR.
PID	Process Identification	Contains the Process ID used for selecting the active TLB entries of the architected TLB.
ZPR	Zone Protection Register	Contains policies for memory protection
TBU/TBL	Time Base Lower/Upper	Contains the number of processor cycles since the register was last resetted.
TCR	Timer control register	Controls the timer behavior of the process.
TSR	Timer Status Register	Contains information about the status of the fixed and programable interval timer (FIT/PIT).
PIT	Programable Intervall Timer	Counts down the processor cycles to the next PIT interrupt.

Table 3: Subset of PPC405FX registers available in supervisor mode

The determination of the WCET of a whole VM requires a control flow graph analysis of the VM to determine the path with maximum execution time within this graph. One of the biggest problems of determining this path is given by loops depending on input values, which lead to a state explosion problem when testing all possible cases. To solve this problem, sophisticated methods [PB00, FH04, WEE+08] and tools like aiT from AbsInt have been developed to derive this path efficiently and as accurately as possible while minimizing the overestimation of these methods. Once this path called P_{WCET} is known, it can be used to determine the WCET of the VM including the overhead induced by the VMM. The path P_{WCET} with length n is then given as a sequence of instructions of the underlying ISA and occuring interrupts during the execution of P_{WCET}.

$$P_{WCET} \in P = (ISA \cup IRQ)^n \tag{3.9}$$

To clarify this modelling, an example for a path p_{WCET} with length $n = 4$ is given as

$$P_{WCET} = ('\text{li r0,1}', '\text{pit}', '\text{ba 0x2400}', '\text{mfsrr0 r3}') \tag{3.10}$$

The path p_{WCET} can consist of privileged, non-privileged instructions and interrupt events. For the determination of the WCET of the path P_{WCET}, the functions $WCET_f : P \to \mathbb{N}$ and $WCET_p : P \to \mathbb{N}$ are defined. $WCET_f$ represents the virtualized execution using full virtualization, while $WCET_p$ represents the virtualized execution using paravirtualization.

$$WCET_f(p) = \sum_{i \in p} WCET(i) + \sum_{i \in p | i \in I_v} (WCET_f(i) - WCET(i)) \tag{3.11}$$

and

$$WCET_p(p) = \sum_{i \in p} WCET(i) + \sum_{i \in p | i \in I_v} (WCET_p(i) - WCET(i)) \tag{3.12}$$

The function $WCET_f(p)$ or $WCET_p(p)$ first calculates the WCET of P_{WCET} by assuming it was executed, and not virtualized, and then adds for every instruction of interrupt the induced overhead of the full or paravirtualization by adding the WCET of the virtualization step and then subtracting the native execution time of the virtualized instruction or IRQ as it is not executed natively anymore. [Bal09]

Measurements

To determine the WCETs of the emulation of privileged instructions and interrupts as accurately as possible, their execution has been simulated cycle accurately using ModelSim [Mod09] together with the VHDL design of a Virtex II Pro FPGA, encapsulating a PowerPC405FX as hard wired core. The advantage of this approach is the cycle accuracy which is not given when using measurements based on the timer register or I/O Pins due to their inherent delays. Using ModelSim allows for the monitoring of the write-back pipeline stage of the five stage pipeline of the PowerPC405FX at any time. Thus, it is possible to determine the exact number of cycles passed until an instruction passed the pipeline completely.

In general, the execution time for the virtualization depends on the parameters of the virtualized instruction. Thus, to determine the WCET, the parameters creating the longest execution paths have been chosen. This was possible due to the internal knowledge of the virtualization process and due to the short execution paths generated by the slim design of the VMM. Otherwise, a pipeline analysis would have been necessary to determine the WCET for all possible parameters. The determination of the WCET of the native execution was realized by disabling the cache and placing the instruction at the beginning of the cache line boundary to force a line fill, which is performed in the PPC405FX to fetch the following instructions even with caching disabled. In general, with caching enabled and a hot cache, the PPC405FX is able to execute instructions in one up to five cycles [IBM05]. Table 4 and 5 show the results for the measured WCETs, with caching disabled, of the privileged instructions (see 3.4.1) of the PowerPC405FX ISA.

The cost of a line fill can be determined by substracting 5 cycles from the measurement of the WCETs of the native execution, because the execution of a single instruction in an empty 5 stage pipeline takes 5 cycles and thus, the cost of a line fill is 52 cycles. These 52 cycles are already included in the values of $WCET_p$ in table 4 and 5.

	rfi	**rfci**	**wrteei 1**	**wrteei 0**
$WCET_{native}$ **[cycles]**	57	57	57	57
$WCET_f$ **[cycles]**	2656	2645	2455	2044
$WCET_p$**[cycles]**	2463	2452	2054	1687
$WCET_i$ **[cycles]**	-	-	-	-
Full/native	46,60	46,40	43,07	35,86
Para/native	43,21	43,01	36,03	29,60
IRFM/native	-	-	-	-

	tlbwehi	**tlbwelo**	**mtmsr**	**mftcr**
$WCET_{native}$ **[cycles]**	57	57	57	57
$WCET_f$ **[cycles]**	3394	2832	2531	57
$WCET_p$**[cycles]**	2555	1819	2075	1654
$WCET_i$ **[cycles]**	-	-	-	180
Full/native	59,54	49,68	44,40	38,75
Para/native	44,82	31,92	36,40	29,02
IRFM/native	-	-	-	3,16

	mfevpr	**mftsr**	**mtevpr**	**mttsr**
$WCET_{native}$ **[cycles]**	57	57	57	57
$WCET_f$ **[cycles]**	2218	2248	2215	2323
$WCET_p$ **[cycles]**	1568	1654	1447	1634
$WCET_i$ **[cycles]**	180	180	180	-
Full/native	38,91	39,44	38,86	40,76
Para/native	27,51	29,02	25,39	28,67
IRFM/native	3,16	3,16	3,16	-

	mttcr	**mttbu**	**mttbl**	**mtpit**
$WCET_{native}$ **[cycles]**	57	57	57	57
$WCET_f$ **[cycles]**	2266	2266	2419	2389
$WCET_p$ **[cycles]**	1558	1547	1711	1714
$WCET_i$ **[cycles]**	-	-	-	-
Full/native	39,75	39,75	42,43	42,07
Para/native	27,34	27,14	30,02	30,07
IRFM/native	-	-	-	-

Table 4: $WCET_f$ and $WCET_p$ measurement. $WCET_i$ represents the $WCET$ using IRFM (see section 3.3.3) for register access [Bal09].

	mtsrr	**mttbl**	**mtpit**	**mtsprg**
$WCET_{native}$ **[cycles]**	57	57	57	57
$WCET_f$ **[cycles]**	2218	2419	2389	2176
$WCET_p$ **[cycles]**	1468	1711	1714	1433
$WCET_i$ **[cycles]**	180	-	-	180
Full/native	38,91	42,43	42,07	38,17
Para/native	25,75	30,02	30,07	25,14
IRFM/native	3,16	-	-	3,16

	mtzpr	**mtpid**	**mfmsr**	**tlbrehi**
$WCET_{native}$ **[cycles]**	57	57	57	57
$WCET_f$ **[cycles]**	2221	2302	2056	2731
$WCET_p$ **[cycles]**	1519	1655	1612	1879
$WCET_i$ **[cycles]**	-	-	180	-
Full/native	38,96	40,39	36,07	47,91
Para/native	26,65	29,03	28,28	32,97
IRFM/native	-	-	3,16	-

	tlbrelo	**mfsrr**	**mfsprg**	**mfpid**
$WCET_{native}$ **[cycles]**	57	57	57	57
$WCET_f$ **[cycles]**	2686	2282	2191	2243
$WCET_p$ **[cycles]**	1733	1639	1568	1629
$WCET_i$ **[cycles]**	-	180	180	180
Full/native	47,12	40,03	38,44	39,19
Para/native	30,40	28,75	27,51	28,58
IRFM/native	-	3,16	3,16	3,16

Table 5: $WCET_f$ and $WCET_p$ measurement. $WCET_i$ represents the $WCET$ using IRFM (see section 3.3.3) for register access [Bal09].

	SC IRQ	**PIT IRQ**	**FIT IRQ**
$WCET_f$ **[cycles]**	1206	1978	1140
$WCET_p$ **[cycles]**	1506	2202	1386

Table 6: $WCET_f$ and $WCET_p$ measurement for interrupt latency overhead induced by the VMM. $WCET_i$ represents the $WCET$ using IRFM (see section 3.3.3) for register access [Bal09].

Compared to the native execution, the emulation of sensitive instructions induces an enormous overhead from factor 38 up to factor 60 in the case of full virtualization and from factor 25 up to factor 44 in the case of paravirtualization. The introduction of the IRFM (see section 3.3.3) increased the performance of the paravirtualization dramatically, resulting in a factor of only 3 compared to the native execution. The mean factor between full virtualized and native execution is about 42, while the mean factor between the paravirtualized and the native execution is about 31 with IRFM disabled and about 22 with IRFM enabled, assuming a uniform distribution of the instruction occurrences. Thus, the full virtualization needs on average about factor 1.35 more cycles than the paravirtualization when the IRFM is not used. When the IRFM is used, the full virtualization needs about factor 2 more cycles than the paravirtualization.

Besides the emulation of native instructions, the VMM is also responsible for virtualizing occuring interrupts as described in sections 3.3.3 and 3.3.4. The occurrence of such an interrupt at first activates the VMM, which decides how to proceed with the interrupt, because the interrupt may have the VMM or the VM as target. This creates additional latencies to the handling of interrupts.

The measurements of these latencies are listed in table 6. It is interesting to see that the paravirtualization using the IRFM feature increased the interrupt latency, while the full and paravirtualization without IRFM are equal in latency. This effect is caused by the mapping of certain registers to the memory of the guest VM. When an interrupt occurs, some of the registers are changed to represent the state before the occurence of the interrupt. This state has to be forwarded to the guest OS interrupt handlers and consequently, then after IRFM feature has been enabled, the state has to be written to the memory location of the IRFM area

	Cycles
$WCET_p(p)$	23563
Measurement	22492

	Cycles
$WCET_f(p)$	37794
Measurement	37394

Table 7: Application of functions $WCET_f(p)$ (3.11) and $WCET_p(p)$ (3.12) compared to the real measured execution times.

of the guest OS to synchronize the machine state with the state represented in the IRFM. This additional synchronization overhead has increased the interrupt latency, while the runtime access time to registers mapped in the IRFM has decreased dramatically as shown in table 4 and 5. Thus, the usage of the IRFM feature is useful when the number of occuring interrupts does not destroy the advantage of the IRFM register access speedup within the guests. [Bal09]

Case Study

In the preceding section, the induced worst case overhead of the single emulation routines and the IRQ handling of the VMM have been shown and mean values have been calculated to show roughly what the mean worst case overhead could look like for a virtualization of a guest OS on the PowerPC405FX. Nevertheless, the real worst case overhead depends on the execution trace being generated by the guest OS and its Tasks. To show the application of the function $WCET_f(p)$ (3.11) and $WCET_p(p)$ (3.12) to determine the WCET for full and paravirtualization, a case study of the RTOS ORCOS is performed. ORCOS has been developed at the University of Paderborn and is designed for the application in deep-embedded systems. The big advantage of using ORCOS for this case study was the availability of the source code and the knowledge of internal details being highly relevant for the application of the proposed VMM design.

The first part of the case study shows the determination of the WCET for the RTOS ORCOS when it has to perform process dispatching after the occurrence of a timer interrupt in the case of full and paravirtualization with IRFM. The execution trace of this dispatching process is finite and unique and could thus be formulated as:

$$
\begin{aligned}
p = (&\texttt{pit}, \ldots, '\texttt{mfsrr0 r27}', '\texttt{mfsrr1 r28}', \ldots, '\texttt{mfpid r31}', \ldots, \\
&'\texttt{mttsr r0}', \ldots, '\texttt{wrteei 0}', \ldots, '\texttt{mttsr r0}', '\texttt{mtpit r4}', \ldots, \\
&'\texttt{mfmsr r2}', \ldots, '\texttt{mtsrr0 r9}', '\texttt{mtsrr1 r2}', '\texttt{rfi}')
\end{aligned} \tag{3.13}
$$

The trace only lists the execution of sensitive instructions that triggered the VMM for emulation. First, the needed execution time of this trace has been measured for the case of full and paravirtualization using ModelSim. Finally the execution trace p has been used to calculate $WCET_f(p)$ and $WCET_p(p)$. Table 7 shows the results of the functions $WCET_f(p)$ and $WCET_p(p)$ and the result of the measurement by simulating the trace using ModelSim. In the case of paravirtualization, the approximation of the WCET by the function $WCET_p(p)$ is about 4.5% higher than the value of the real execution. This is due to the assumption of occurring line fill effects when using paravirtualization (see 3.4.2). The approximation of the fully virtualized WCET is about 1% higher than the value of the real execution. Thus, the approximation of the functions $WCET_f(p)$ (3.11) and $WCET_p(p)$ (3.12) are a good indication for the real WCET.

The question of how to determine an approximation of the virtualization WCET for a given instruction or program trace p has been addressed in this section, but there is still the question to answer what especially creates this overhead in virtualization. Therefore, the overhead induced by the different components of the VMM design will be determined in detail in the next section. [Bal09]

3.4.3 Virtualization overhead

Within this section, the induced overhead of the different VMM components is analyzed to identify the bottlenecks of the virtualization process. It has to be noted that this heavily depends on the hardware used. Thus, the result of this section does not only focus on the improvement of the implementation, but also on the improvement of the hardware support to boost virtualization even on embedded hardware.

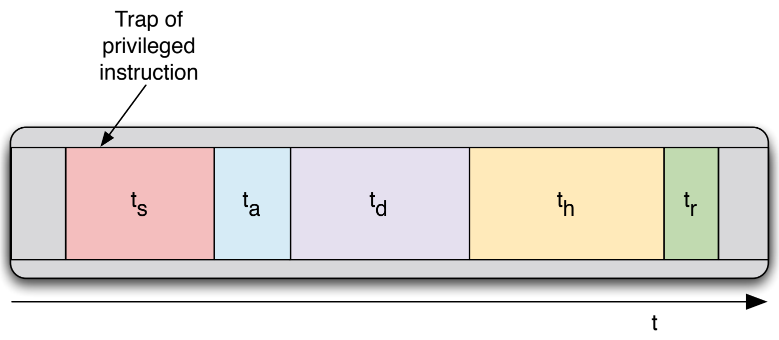


Figure 44: Virtualization overhead

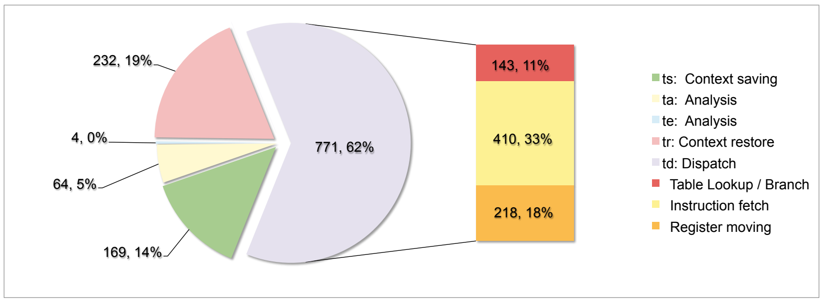


Figure 45: Virtualization overhead of *mtevpr* in detail [Bal09].

To determine the cost of the different components of the VMM design it is helpful to reconsider figure 44. First, the VMM saves the context upon an interrupt taking time t_s. Then, the VMM needs to identify the kind of interrupt and its source taking time t_a. Afterwards, the dispatcher is responsible for delegating the identified interrupt to the associated component, what takes the time t_d. Then, the handling is performed by this component taking the time t_h. Finally, after the handling is finished, the control is transferred from the VMM to the target VM by restoring the context of the target VM taking the time t_r. Figure 45 shows these values.

The overhead t_s, t_r is constant and depends on the hardware architecture, while the analysis overhead t_a depends on the source interrupt. For the instruction *mtevpr*, this overhead is caused due to the analysis of a Program IRQ being raised upon execution of the *mtevpr* in userspace. The analysis needs to decide whether the program IRQ was raised in user mode or in supervisor mode. Due to the program IRQ being ambiguous, the action to be performed needs to be identified. In case of the *mtevpr* instruction, this needs 5% of the whole emulation process. When an emulation is required, the instruction that caused the interrupt must be loaded from memory, because the Power Architecture only stores the address of the instruction in a save and restore register. This is denoted as instruction fetch of the dispatching overhead $t_d$and requires a memory access of the VMM to the address where the instruction is stored contributing 11% to the total overhead. Afterwards the instruction opcode and its parameters have to be identified within this instruction word being represented by the table lookup and branch overhead of t_d. Due to the instruction encoding of the Power Architecture, this task in the worst case requires a two level indirection analysis (in case of *mtspr* and *mfspr*) to identify the instruction. This task contributes 33% to the total emulation overhead. When the instruction has finally been identified, the parameter needs to be identified and moved from its register to the parameter register of the emulation method. This can simply be implemented as a switch statement.

```
source_reg = extract_source_register(inst)
switch(source_reg) {
case 0: regs[13] = regs[0]
case 1: regs[13] = regs[1]
case 2: regs[13] = regs[2]
...
}
```

Being in principle quite simple, this switch statement induces two indirect branches, which lead to a stall of the pipeline and thus contributes a large part of 18% to the whole emulation of the instruction. Thus, a total of 62% of the emulation overhead for the instruction *mtevpr* is induced by the dispatching to the associated emulation routine. The emulation routine of *mtevpr* is very simple, as it only needs to move the parameter value to the EVPR register of the VM and contributes only 4 cycles to the emulation overhead. Finally, the context saving and restoring routines contribute 14% and 19% to the emulation overhead.

The most costly part of the emulation is thus the dispatching process, as there are included several indirect branches which lead to a stall of the pipeline.

While the emulation of the instruction *mtevpr* does not require a large amount of computation time to emulate the instruction itself, this cannot be transfered to all instructions of the Power ISA. A register requiring a much larger emulation overhead is the machine state register (MSR) being accessed by *mtmsr* and *mfmsr*. When accessing this register, some checks need to be performed in the emulation routine. One example is the activation and deactivation of interrupts. When interrupts are deactivated for a specific VM, the interrupts are buffered by the VMM, as the interrupts are only virtually disabled, because the sharing of the Timer register requires a permanent activation of interrupts to guarantee the resource control property of the VMM (see 2.2.2). Upon reactivation of interrupts for a VM, the emulation routine of *mtmsr* needs to check whether there are buffered interrupts pending for the VM. When there are pending interrupts, the handling of these interrupts needs to be invoked. This interrupt handling mechanisms require about 50% of the whole emulation routine for the MSR register. [Bal09]

3.4.4 Footprint

In order to be deployed on small scale nodes, the memory overhead introduced by the virtual machine monitor needs to be as small as possible. Since the VMM is configurable completely, the amount of binary and memory footprint occupied by the virtual machine monitor strongly depends on the features used by the target system. Thus, it was necessary to build configurations that allowed for the extraction of the binary and memory footprint of the configurable features. The results are presented in table 8.

The values have been derived by compiling the different configurations using the GNU C Compiler 4.5 with optimization level 2. Remember that for additional VMs, the memory footprint of the IPCM, TLB and the IRFM features need to be multiplied by the number of available VMs. Furthermore, 300 bytes of the core memory footprint are allocated for the VM context. Thus, this number also needs to be multiplied by the number of available VMs to get the total memory footprint.

In total, the VMM needs approximately 8KB of memory for its code and approximately 4KB of memory for its data structures. The data structures are statically allocated and thus, there is no

Feature	**Binary Footprint [bytes]**	**Memory Footprint [bytes]**
IPCM	464	2
TLB	960	648
Previrtualization	192	0
IRFM Fallback	620	0
IRFM	552	92
Paravirtualization	648	160
Full virtualization	472	2172
Core	4072	760
Total	7980	3834

Table 8: Binary and memory footprint of the VMM with one VM configured.

dynamic memory allocation during runtime for the VMM. The core already includes the emulation routines for full virtualization, as full virtualization is always used as a fallback feature. When enabling the full virtualization feature, mainly the memory footprint increases due to the dispatching tables used to speedup the full virtualization process. Nevertheless, full virtualization already works when only using the core configuration. [BK09]

3.4.5 Performance

The performance overhead introduced by the virtualization software has been measured for a set of application scenarios. As real-time operating system ORCOS (University of Paderborn, 2009) running on a PowerPC405 processor has been used to measure the overhead for two application scenarios. The results are shown in 9.

The first scenario used a simple periodic real-time task for the only purpose of counting a variable to a specific value. The execution time was measured for an interval of two hundred repetitions to get a reasonable measurement. The overhead produced here was about 60% compared to the native execution time. The high overhead in this scenario can be easily explained by the fact that the amount of time spent inside the operating system compared to the amount of time spent inside the real-time task was really high. Thus, the relative amount of privileged instructions used by the operating system which needed to be emulated was high, contributing to this kind of overhead.

A more realistic scenario was given by another real-time task which had to calculate a Fast Fourier Transformation for a set of input values for a fixed amount of repetitions. Since the time spent for the calculation was much higher compared to the former example leading to a much smaller amount of emulation routines called the overhead reduced to less than one percent if the application is executed para-virtualized. It is also possible to see that running the unmodified application in full-virtualization mode is only slightly slower. The overhead increases to 0.56% in that case. Considering embedded control systems, this renders the usage of full-virtualization extremely interesting. Furthermore, this observation gives reason to believe that the execution time of applications that could not be paravirtualized completely, since for example a part of the application has been linked against static libraries, will not suffer much by running inside the VMM introduced in this thesis.

Applications using only the user ISA as given by the third example application SimpleFFT, which calculated a Fast Fourier Transformation without using the support of an operating system, can be executed with native performance, because no instruction needs to be emulated by the virtual machine monitor. [BK09]

3.5 SUMMARY

The overall goal of this thesis is the integration of software components into a big integrated system. Thus, there are given real-time systems which may have been executed on different hardware than the integrated systems run on. To make these real-time systems executable on the integrated system, every real-time system is encapsulated as RTVM. These RTVMs are then executed on the VMM introduced within this chapter, which shall ensure temporal and spatial isolation to prevent unintentional interactions. There already exist a few commercial virtual-

Execution Mode	**ORCOS + CTask**	**ORCOS + FFTTask**	**FFT**
Native [ms]	10.73	5094.97	294.56
Full virtualization [ms]	26.51	5123.38	294.56
Paravirtualization [ms]	17.16	5117.94	294.56
Full Virtualization/ Native [%]	246.95	100.56	100
Paravirtualization/ Native [%]	159.92	100.45	100
Full / Para [%]	148.65	100.11	100

Table 9: Performance of three different virtualization scenarios.

ization platforms or VMMs for a range of embedded processors nearly all of them being proprietary systems. All of the available products only use paravirtualization trying to provide reasonable performance and support realtime applications only by the use of dedicated resources. Naturally, this limits the applicability of virtualization to a subset of all possible scenarios, as in general, the paravirtualization interfaces are not standardized. Especially whenever there are applications that cannot be paravirtualized, because the source code is not available, these applications cannot be virtualized using the currently available virtualization products, since this would require a binary analysis of the whole application which most often is not completely possible.

Thus, a VMM providing a configurable hybrid VMM architecture was designed and implemented for the PowerPC405 processor which allows for the virtualization of unmodified applications as well as paravirtualized applications or even a combination of both. To describe this in more formal manner, the ABI is kept to be configurable. The paravirtualization effort is thus decreased, as only the required hypercalls need to be implemented in the guest OS. The support for paravirtualization was motivated by the integration of open source GPOSs for High-Level API support like Linux which already provide paravirtualization interfaces.

Support for realtime applications was the next major goal of the design, which allows for the integration of any kind of scheduling by the introduction of an extensible scheduler interface and while being completely deterministic. Furthermore, the configurability allows the system to be optimized explicitly for the intended field of use. This affirmed research in this direction with hybrid virtualization being a relevant topic in industrial embedded systems. Finally, it is desirable to know in advance how the WCETs of the executed guests are affected, as these WCETs are necessary to determine the CPU requirements of the virtual real-time system. Therefore, the WCETs of all available privileged instructions, IRQ Handlers and Hypercalls have been determined to include these values in the final WCET of a real-time task being executed in a RTVM.

The support for real-time applications requires the temporal isolation of the executed RTVMs, which has not been addressed up to now, but the interface to provide this has been introduced by the realized extensible scheduler interface. The next chapter thus covers the problem of temporal isolation in a virtualized environment with hierarchical scheduling.

4 SCHEDULING OF FULL VIRTUALIZED HARD REAL-TIME SYSTEMS

Contents

The goal of a VMM is to give to each VM the illusion of having the resources of a complete system at its disposal. Nevertheless, the resources assigned to VM are in reality only subsets of a physical machine. When supporting multiple VMs on a virtualization platform the VMM needs to implement a scheduling algorithm. According to this algorithm the VMM switches between the VMs at the root-level of the hierarchy. From the VMM's point of view, the executed VMs are just tasks. The VMs itself host different OSs scheduling their own set of tasks at the local level (see figure 46). In case of full virtualization, these VMs appear to be blackboxes while in case of paravirtualization, the VMs can communicate with the VMM scheduler. Thus, the scheduling question is in general a two level hierarchical scheduling problem to be solved under the given communication constraint of full or paravirtualization. Answering this questions means to find a suitable virtualization host and a valid root-level schedule that does not violate the real-time constraints of the given RTVMs.

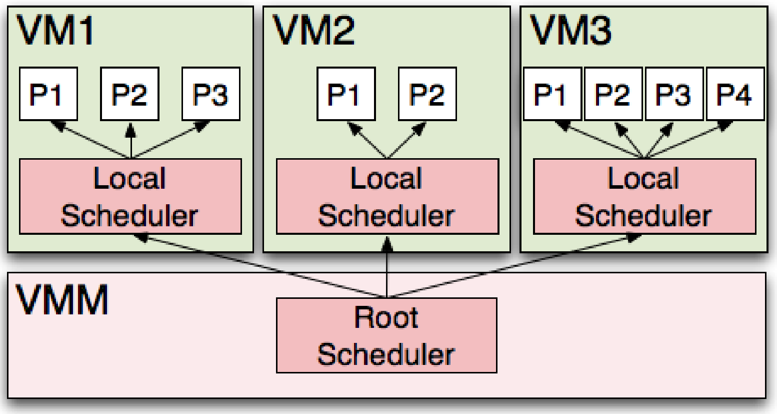


Figure 46: Hierarchy of involved schedulers

4.1 PROBLEM STATEMENT

The problem to be addressed in this thesis has already been described at the beginning of chapter 3. The overall goal is the integration of software components into a big integrated system as depicted in figure 47. The previous chapter has built the infrastructural fundament for this chapter by providing a highly deterministic VMM with an extensible scheduler interface and the possibility to determine the WCETs of the virtualized guests, which are required by the root-level scheduler.

When reconsidering figure 47 the open questions are the derivation of the CPU requirements and the derivation of the root-level schedule.

All existing methods presented in the following section 4.2, except for the open system environment lack the possibility of deriving the root-level schedule from a given set of real-time systems. At a first glance, the open system environment could be the key to the stated problems, but it requires paravirtualization violating the constraint of full virtualization. The compositional real-time scheduling framework is quite close in this direction but still requires the definition of static resource partitions at guest level and valid schedules on these partitions to finally derive the root-level schedule. Thus, it is not possible to derive the root-level schedule from only the given real-time systems.

The approach presented in this chapter will solve this problems by deriving the necessary cpu requirements to prevent an over-

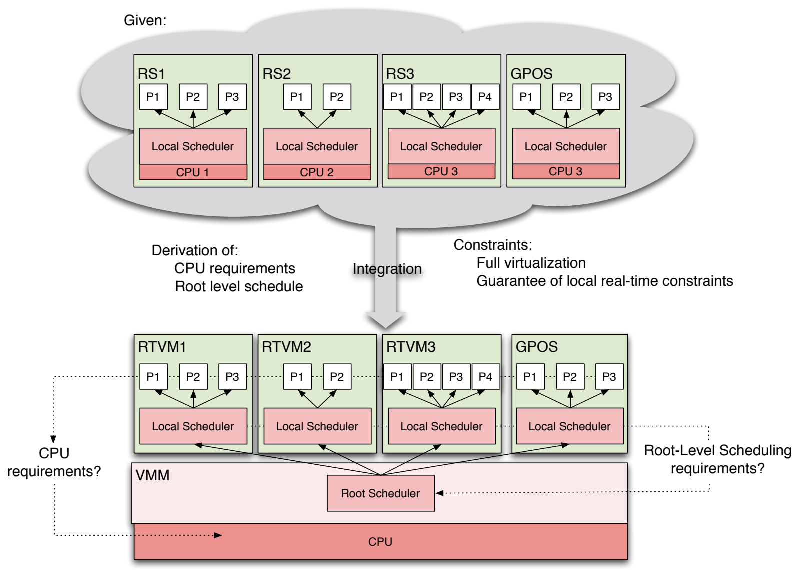


Figure 47: Problem of integrating given real-time systems into an virtual system hosting these real-time systems as RTVMs.

load situation on the host system. This is based on a normalization and scaling of the tasksets to a faster processor. This considers the specifics of the local scheduling policies like EDF or RM. When it is ensured that there is no overload situation, the root-level schedule is derived from only the given real-time systems while fulfilling the constraint of full virtualization at runtime. This is realized on specifically derived resource partitions which ensure the real-time execution of all real-time tasks in the virtual real-time system. But first an in depth look at the afore mentioned related work is performed, which covers the derivation of the root-level scheduler under the constraint of full virtualization.

4.2 RELATED WORK

The problem of hierarchical scheduling has already been addressed in academia and industry. Section 4.2.1 will begin with the related work originating from academia. Subsequent to this section, the related work developed in the industrial field is presented in section 4.2.2. Finally, a classification of the available related work is given in section 4.2.3.

4.2.1 Academia

The related academic work is presented in this section beginning with the general concept of partitioning resources at root-level. The introduced models for partitioning resources do not allow a derivation of a specific partitioning scheme which guarantees the schedulability of a real-time taskset. Instead, the developer needs to define such partitioning schemes on its own and needs to check these partitioning schemes for schedulability. Therefore, some of the approaches, provide a schedulability test for a given partition. Thus, when deriving a root-level schedule based on these partitioning approaches, a search has to be applied. That can be very complex as will be shown later in this chapter. Finally a paravirtualization-based approach called open system environment is presented, which is able to derive the root-level schedule at runtime for the RTVMs. Violating the full virtualization constraint, it is nevertheless not a solution for the problem stated in sections 3.1 and 4.1.

Proportional Share Scheduling

The general idea of proportional share scheduling [Tij80] is to provide a predefined amount of the processor capacity to each available guest. Let $A(t)$ be the set of all active clients at time t and let w_i be the computational weight assigned to client i. The share $f_i(t)$ of client i at time t is then defined as

$$f_i(t) = \frac{w_i}{\sum_{j \in A(t)} w_j} \tag{4.1}$$

Assuming a perfect fair system, the service time $S_i(t_0, t_1)$ assigned to a client i during the timer interval $[t_0, t_i]$ is given as

$$S_i(t_0, t_1) = \int_{t_0}^{t_1} f_i(t)dt \tag{4.2}$$

Equation 4.2 represents a system in which it is possible to assign infinitesimal small intervals of time to each client. The smallest assignable time interval is called time quantum q. Unfortunately, the assumption of q being infinitesimal small is not practical for

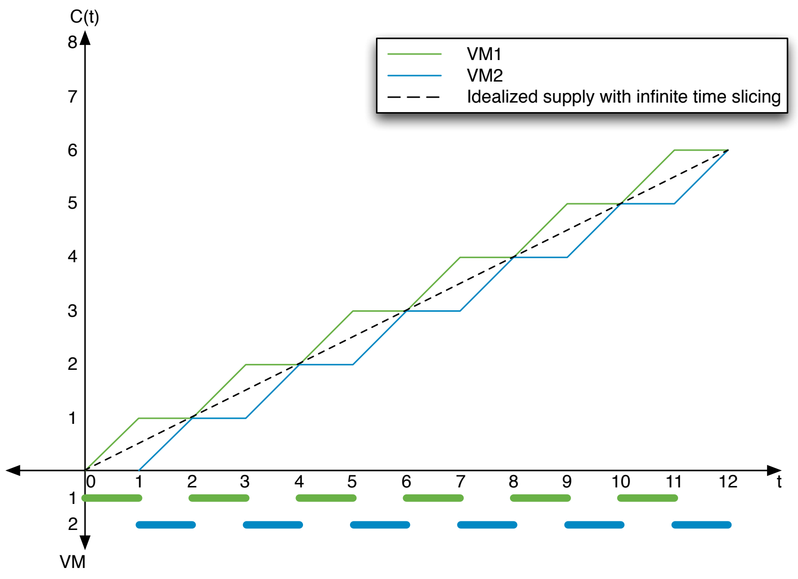


Figure 48: VM supply function C(t) vs real time elapsed.

real computing systems due to the overhead induced by context switching for very small values of q. Thus, one problem of proportional share scheduling is the decision on how to choose the granularity which is defined by the time quantum q, since q has a direct impact on the overhead induced by the scheduling. The other important problem arising is guaranteeing the service time $S_i(t_0, t_1)$ for specific intervals $[t_0, t_i]$. Due to the discretization using the time quantum q as smallest possible time interval, the real service time $s_i(t_0, t_1)$ may deviate from the idealized service time $S_i(t_0, t_1)$ based on infinitesimal time slicing.

The approximation for a time quantum $q = 1$ is depicted in figure 48. The dashed line represents the idealized supply function with infinitesimal time slicing, while the green and the blue lines show the approximation of the supply function for two virtual machines sharing the processor equally with a time slice length of one time unit ($q = 1$). The bars below the function indicate the activation of the virtual machines.

The deviation from the real assigned service time $S_i(t_0^i, t)$ to the ideally assigned service time $s_i(t_0^i, t)$ is called *lag*:

$$lag_i(t) = S_i(t_0^i, t) - s_i(t_0^i, t). \tag{4.3}$$

t_0^i denotes the point in time of which client i became active. The lag needs to be lower bounded for real-time systems to guarantee a certain minimal amount of processing capacity during a specific time interval $[t_0, t_i]$. Thus, the system designer can determine the minimal guaranteed service time for a given real-time task τ_i executed in an interval $[t_0, t_i]$.

The problems being mainly addressed in literature so far are the quality and the bound of the lag. Different proportional share algorithms, addressing these aspects have evolved in the 80s and 90s. *Weighted Fair Queuing (WFQ)* [DK89], *Packet Fair Queuing (PFQ)* [DK89], *Lottery Scheduling* [WW94], *Stride Scheduling* [Wal95] and *Earliest Eligible Deadline First* [SAWJ+96, JSMA98] are popular examples for such algorithms.

An open question all proportional share algorithms have in common is how to choose the time quantum q. A small time quantum q leads to a massive switching overhead while choosing a large time quantum q leads to a larger lag but less switching overhead.

Resource Partition Model

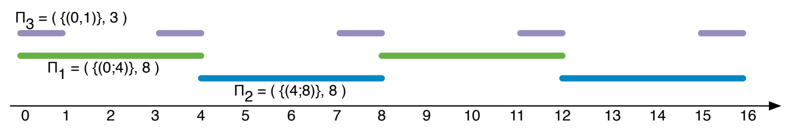
Mok, Feng and Chen introduced in [MFC01, FM02] the concept of *real-time virtual resources* that are shared among real-time task groups. They introduced a formalism for the schedulability analysis of a task group executed on a *single time slot* or *multiple time slot periodic partition*.

First they introduce the *static resource partition model*. In a nutshell, a (temporal) static partition is simply a collection of time intervals during which the physical resource is made available for a set of tasks which is scheduled on this partition. Two types of periodic resource partitions are introduced:

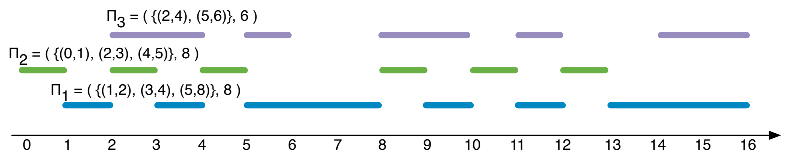
- *Single Time Slot Periodic Partitions (STSPP). See figure 49a.*
- *Multiple Time Slot Periodic Partitions (MTSPP). See figure 49b.*

In case of a STSPP, only one time interval exists ($N = 1$), while in case of a MTSPP, more than one time interval is specified ($N > 1$).

Definition 4.1. *A resource partiton Π is a tuple (Γ, P), where Γ is an array of N time pairs $((S_1, E_1), ..., (S_N, E_N))$ that satisfies $0 \leqslant S_1 <$*



(a) STSPP examples.


Π3 = ({(2,4), (5,6)}, 6)
Π2 = ({(0,1), (2,3), (4,5)}, 8)
Π1 = ({(1,2), (3,4), (5,8)}, 8)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

(b) MTSPP examples.

Figure 49: Resource Partition Model

$E_1 < ... < S_N < E_N$ *for some* $N \geqslant 1$*, and* P *is the partition period. The physical resource is available to the set of tasks executed on this partition only during intervals* $(S_i + j \cdot P, E_i + j \cdot P), 1 \leqslant i \leqslant N, j \geqslant 0$ *[MFC01].*

The static resource partition model is in general a more formal definition of the timeline scheduling approach presented in 2.1.2. The schedulability analysis of a task set executed on such a resource partition is no longer possible by comparing the utilization of the task set to traditional utilization bounds introduced by Liu and Layland in [LL73]. To solve this problem, Mok and Feng introduced the concept of critical instances for resource partitions and proposed schedulability tests for fixed and dynamic priority scheduling of the task sets executed on a given static resource partition. The static resource partition model approach of Mok and Feng provides a generalized formalization and schedulability analysis of timeline scheduling introduced in section 2.1.2

Mok and Feng extended this static resource partition model to the *bounded-delay resource partition model* to provide more flexibility. They argued that static resource partitions are an important technique for designing high criticality applications since the static model is very amenable to timing correctness certification, but in other cases, it is desirable to give more flexibility in granularity to the root scheduler. The partition schedule in case of the static resource partition model is defined directly by the time pairs γ and the partition period P.

Definition 4.2. *A bounded delay resource partition Π is a tuple (α, Δ), where α is the availability factor of the partition and Δ is the partition delay [MFC01].*

This definition actually defines a set of partitions, because there are many different partitions in static partition model that satisfy this requirement. The bounded-delay resource partition model allows the creation of a root-level schedule based on the definition of an availability factor α_k and the definition of partition delay Δ_k for each partition being scheduled by the root-level scheduler. The parameter α_k ensures that $\alpha_k \cdot L$ time units of the resource in any interval of length $L + \Delta_k$ are assigned to the partition with the delay bound Δ_k. Starting from any point in time, Δ_k specifies the maximum period of time the task group may have to wait before receiving its fraction of the processor. Nevertheless, the approach requires a valid schedule of the task set on a given static resource partition to create a transformation to a feasible schedule for the bounded-delay resource partition model. A very important thing to notice is that the resulting feasible schedule may not preserve the original scheduling policy. When this transformation is feasible for all partitions, a root-level schedule can be derived based on given resource requirements (α_k, Δ_k) with $\Delta_k \leqslant \Delta$ for each partition, where Δ is the maximal possible delay for the associated partition k.

The work of Mok and Feng provides a solid formal fundament for modeling and testing static resource partitions for schedulability of a given task set. The modeling of static resource partition requires the definition of the parameter γ, being an array of N time pairs in which the partition is active, and the period P of the resource partition by the system designer which can then be tested for schedulability using fixed or dynamic priority scheduling algorithms. Mok and Feng extended their work by the bounded resource partition model, which provides more flexibility in creating a root-level schedule due to the specification of the resource availability factor α_k and the partition delay Δ_k. The static resource partition model is applicable to virtualized environments based on full virtualization, as it is possible to combine resource partitions in a way that two partitions do not allocate a resource at the same time while creating feasible real-time schedules on the partitions. The bounded-delay resource partition model is only applicable to paravirtualized environments, because the derived schedule on a specific resource partition may not preserve the original scheduling policy. Thus, the local scheduler must be modified to implement the derived schedule.

Compositional Real-Time Scheduling Framework

Shin and Lee extended the *resource partition model* in [SL03, SL04, SL08] by schedulability tests for RM and EDF being applied on a given periodic resource $\Gamma(\Pi,\Theta)$. Π is the period of resource and Θ is the capacity of the resource supplied over the period Π. Additionally they introduced a method to determine the maximal utilization for a periodic task set being schedulable on a given periodic resource $\Gamma(\Pi,\Theta)$ using RM or EDF. Based on this result, they developed a compositional real-time scheduling framework where global (system-level) timing properties are established by composing independently (specified and) analyzed local (component- level) timing properties. This framework derives the timing requirements of the hypervisor scheduler from the timing requirements of the guest schedulers in a compositional manner, that is to say such that the timing requirement of the root-level scheduler is satisfied, if, and only if, the timing requirements of the guest schedulers are satisfied. The scheduling model on hypervisor level is schedulable, if and only if, each scheduling model on guest level is schedulable.

The approach of Shin and Lee is very interesting when defining RTVMs as a periodic resource and the designer wants to know what maximal utilization is schedulable on such a periodic resource or whether a given taskset is schedulable on that resource. Once the designer has specified the local periodic resource with their tasksets and tested them for schedulability, it is possible to derive a global schedule of that local periodic resources. Thus, a composition of individually specified periodic resources is achieved using a global scheduling mechanism. It is not possible to completely derive a valid root-level scheduler, since the partitioning schemes for the different RTVMs have to be specified manually by the designer. This thesis also covers the derivation of the partitioning schemes for a valid root-level schedule.

Open System Environment

The open environment for real-time applications developed by Deng and Liu in [DLS97, DL97] and the BSS-I [LB00] and PShED frameworks [LCB00] developed by Lipari et al. are hierarchical scheduling systems designed around the idea of providing a USP[1] to each real-time application. The USP abstraction guarantees that any task set (i.e. the set of real-time tasks comprising a real-time application) that can be scheduled without missing any deadlines (by a particular scheduler) on a processor of speed s can also be scheduled by that scheduler if it is given a USP with rate $\frac{s}{f}$ on a processor of speed f. The USP itself is realized by implementing a CUS[2]. The CUS replenishes its budget to its capacity B at the end of the servers period in comparison to an immediate replenishment when the budget is exhausted as in case of the total bandwidth server(TBS) [But04]. This is due to the reason that periodic tasks do not have any benefit from finishing much earlier than their deadline. The USP guarantee is characterized by a share of the processor and does not make any assumptions about the granularity at which this share will be granted. The characterization by the share is realized in the CUS replenishment policy as:

$$d_{server} = t + \frac{B}{U} \tag{4.4}$$

The higher the utilization of a server, the higher its replenishment rate. Compared to proportional share schedulers, this is the main difference, as proportional share schedulers are based on the granularity given by the time quantum q. Thus, the granularity information must be specified dynamically by communicating the next deadlines of all local schedulers at runtime to the root scheduler. This requires the usage of a paravirtualization interface between the guest OS hosting the local scheduler and the hypervisor hosting the root scheduler. With the knowledge of the local schedulers, the root scheduler is able to ensure that the USP assigned to each guest never uses more of its share of the processor over any time interval that matters to any USP. The requirement that all deadlines are specified dynamically adds run-time overhead and is not a good match for some kinds of schedulers. For example, a rate monotonic scheduler retains no

1 uniformly slower processor
2 Constant Utilization Server

information about task deadlines at run time, and consequently cannot make use of a USP guarantee based on a root-level EDF scheduler [Reg01].

Fixed-Priority-Driven Open Environment Scheduling

The original proposal of the open environment uses an EDF scheduler at root-level and constant bandwidth servers to encapsulate the guests. Besides the fact of EDF minimizing the maximum lateness of its scheduled tasks and the possibility to utilize the processor up to 100%, EDF requires much more implementation and runtime overhead than fixed-priority schedulers like RM. Thus, Kuo et. al proposed an approach using RM as root-level scheduler for the open system environment in [KL99]. The approach of Kuo et. al uses sporadic servers to encapsulate the guests. To guarantee the real-time execution of a guest, the period of the associated sporadic servers P_s has to fulfill the following inequality

$$P_s < \frac{d_i}{2 + \lceil \frac{c_i}{c} \rceil} \tag{4.5}$$

for every τ_i of the task set of the guest with d_i being the relative deadline of τ_i, c_i being the required computation time of τ_i, and c being the capacity of the sporadic server.

Like the original open system environment which uses EDF as root-level scheduler, the extension by Kuo et. al requires a paravirtualization of the guest OS, because the replenishment of the sporadic servers is based on the knowledge of their remaining capacity. Due to the general possibility of having idle times in a RTVM, the root-level scheduler cannot derive the remaining capacity of the scheduled sporadic servers without modifying the guest scheduler to communicate the resource consumption to the root-level scheduler.

Open System Environment Server Parameters

To analyze general research questions which are valid for all server-based hierarchical scheduling approaches like the Open System Environment (see 4.2.1), Lipari and Bini [LB05] built an abstract server model. The servers of this model are characterized by their maximum computational budget Q and their server period P. Thus, a tuple (Q, P) is assigned to every virtual machine. The virtual machine then receives Q units of execution time every P units of time. This is equivalent to the general periodic task model introduced in section 2.1.1. The problem addressed by Bini and Lipari is the question how to choose the tuple (Q, P) to guarantee the schedulability of the applications executed within the servers. Two opposite needs have to be balanced. A large P is desired to avoid a waste of time in context switches. Context switches have to be performed by the global scheduler and are costly in terms of time. Since a large $\alpha = \frac{Q}{P}$ leads to a high utilization, a small required bandwidth is desired to avoid a waste of total processor capacity. They answered this question for virtual machines composed of periodic and sporadic tasks which are scheduled by a fixed priority local scheduler.

To find the class of tuples of server parameters (Q, P) that make the task set feasible, all possible tuples are derived. Based on this result, the system designer could choose the best trade-off between a large P and a small α. An analysis of the temporal behavior of a server has to precede the schedulability analysis of the task set. The authors define a function $Z_s(t)$ as the minimum amount of time provided by server S in any time interval of length $t \geqslant 0$. A task is feasible on server S if the time $Z_s(t)$ provided by S is greater than or equal to the worst case execution time requested by this task and all tasks of higher priority, since all tasks of higher priority delay the execution of the analyzed task. The schedulability conditions are expressed as a set of linear inequalities. By solving these, a class of tuples with guaranteed schedulability is obtained and the system designer can select the server parameters.

The work of Lipari and Bini allows to compute the best server parameters for a two-level hierarchical scheduling system based on the open system environment. Other hierarchical scheduling approaches leave this open and just assume that the best server parameters are known. Their approach is however complex to implement and introduces a high scheduling overhead. The underlying server abstraction generalizes their findings and can be used for future research in the field of real-time servers.

4.2.2 Industry

The related work in the industrial field is mainly based on timeline scheduling approaches (see section 2.1.2), as they guarantee a strict timeliness which is necessary for a lot of applications involving sensors and actors. A very interesting approach of a hierarchical scheduled system is introduced by the ARINC 653, a standard developed for avionic computing applications. Unfortunately, the ARINC653 standard is based on creating the timeline schedules manually, what is violating the goal of deriving the schedule from the given real-time systems, as described in section 4.1. Finally, the operating system PikeOS implementing the ARINC 653 standard is presented.

ARINC 653

The architecture of the ARINC653 standard has already been described in section 3.2.2. Now, the approach of scheduling the partitions is presented in this section. In general, an ARINC653 partition is equivalent to a program encapsulated in a single application environment. The partitioning separates applications in space and time. The time partitioning is realized as a static configuration and a set of execution windows is assigned to each partition (see figure 50). The processes within the scope of a partition are scheduled by the partitions scheduler. The basic approach of the ARINC653 scheduling scheme can easily be identified as timeline scheduling (see 2.1.2). Thus, it can be modeled by the static resource partition model introduced by Mok and Feng (see 4.2.1). Nevertheless ARINC653 does also not give a solution to the problem of deriving a feasible root-level schedule

The ARINC653 standard is implemented by most of the available commercial VMMs such as Greenhills Integrity Secure Virtualization (see3.2.2), Lynuxworks Lynxsecure (3.2.2), Windriver Hypervisor (3.2.2), and Sysgo PikeOS (3.2.2). Thus, all root-level schedules of these hypervisors can be modeled using the concept of resource partitions introduced by Mok and Feng (see 4.2.1).

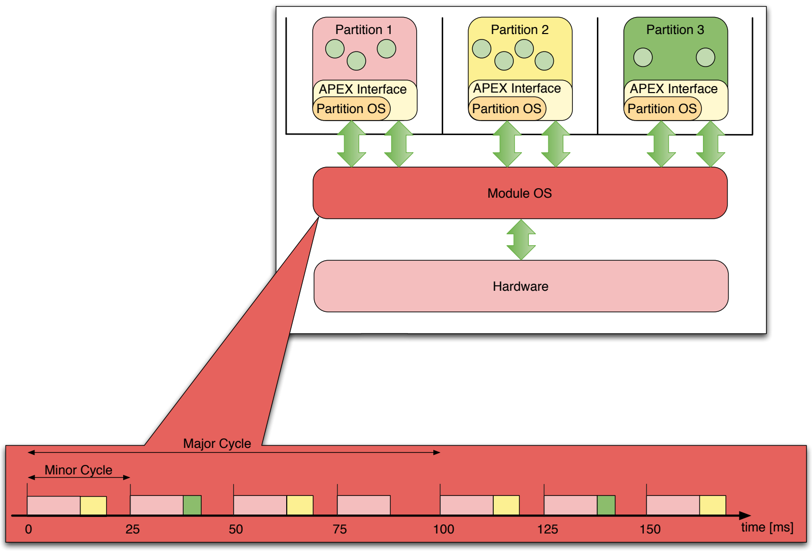


Figure 50: ARINC653 partition scheduling

Pike OS

As described in the previous section, PikeOS implements the ARINC653 standard and thus provides temporal isolation by timeline scheduling. Nevertheless, there are some small adaptations in wording and implementation worth to mention in PikeOS. The first adaptation affects the naming of partitions. PikeOS introduces two type of partitions:

- Resource Partitions
- Time Partitions

A resource partition is a set of PikeOS tasks sharing a bounded set of kernel resources assigned to the resource partition and exception monitoring handlers. Besides the assignment to a resource partition, each PikeOS thread is assigned to a specific time partition τ_i. In figure 51, σ_{VM} shows the activation of three different time partitions while σ_i shows the thread activations within the time partition τ_i. τ_0 is a special time domain in PikeOS which is alway active and is responsible for handling asynchronous events. Thus, the dispatcher needs to decide between the highest priority thread of the currently active time partition τ_i and the domain τ_0. This adaptation is the main difference to the standard ARINC 653 scheduling approach. Nevertheless, the scheduling in PikeOS can also be modeled by the

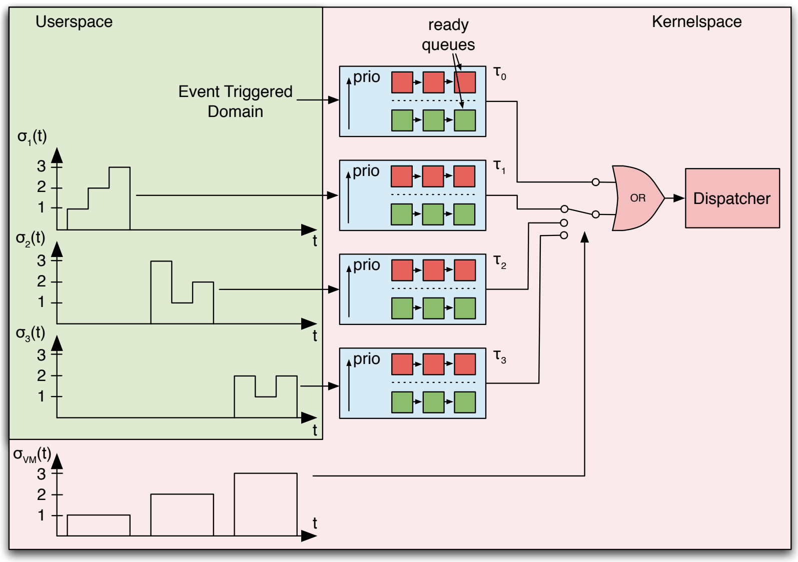


Figure 51: PikeOS scheduling mechanism. The OR operator realizes a preemptability of the time triggered partitions τ_i by the event triggered background partition τ_0

concept of resource partitions introduced by Mok and Feng (see 4.2.1) when this extension is not implemented.

4.2.3 Classification

In general, it is possible to classify the introduced scheduling algorithms based on their root-level scheduling behavior into the two basic classes:

- Static scheduling
- Dynamic scheduling

Static scheduling:
This class includes the classical approach of static proportional share scheduling and its direct derivates of the Fair Queueing class. The root-level scheduler statically partitions the shared resource and makes the partition available to the clients in multiples of the time quantum q at a fixed rate. The main problem when applying proportional share schedulers at the root-level of a real-time hierarchical scheduling is the choice of the time quantum q and the bound of the lag at arbitrary time. The time quantum q directly influences the overhead induced by the root-

level scheduler, because choosing a large time quantum q leads to less switching overhead, but a larger lag.

The timeline scheduling approaches as used in the ARINC 653 standard also fall into this category. The timeline is partitioned in minor and major cycles with the minor cycles being the time synchronization points, where the scheduler is activated by a timer interrupt. Thus, the minor cycle specifies the rate of the scheduling and influences the switching overhead comparable to the time quantum q of proportional share schedulers. In general, the static proportional share schedulers and the timeline schedulers are equivalent.

The static resource partition model is another approach of this class. Within the major cycles of a resource partition, which is called partition period P, the resource partitions activations are completely customizable by the system designer by defining the parameter Γ being an array of N time pairs. This is the main difference to static proportional share and timeline scheduling. Nevertheless, it is possible to model static proportional share schedules and timeline schedules as static resource partitions.

The compositional real-time scheduling framework is also included in the class of static scheduling, because it is based on the composition of static resource partitions based on given valid schedules on given resource partitions.

Dynamic scheduling:
Besides the static root-level schedulers, some dynamic root-level schedulers have been introduced. One is the Earliest Eligible Virtual Deadline First Algorithm, which is the only proportional share based algorithm in this class providing a bound of the lag of $O(1)$ at the time a deadline expires, what allows for the scheduling of firm real-time tasks using deadline based scheduling at the local scheduler level. Nevertheless, the algorithm also requires a communication of the local deadlines to the root-level scheduler what can only be realized by applying paravirtualization.

The original open system environment published by Deng et. al is based upon EDF scheduling of constant bandwidth servers. The decision of the granularity is based on the deadlines of the local schedulers and does not depend on the time quantum q like in proportional share schedulers. Therefore the root-level scheduler needs to communicate with the local schedulers re-

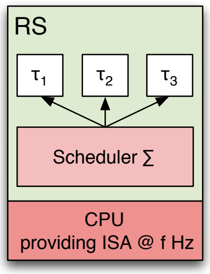


Figure 52: Model of a real-time system

sulting in a paravirtualization of the guest OS encapsulated in the server. The RM extension of the open system environment published by Kuo et. al introduces RM as root-level scheduler, but still requires paravirtualization.

4.2.4 Summary

The presented related work does not give a satisfying solution to the problem of deriving a root-level schedule from given real-time systems. Instead, there are interesting partitioning models, like the resource partition model, which provides a schedulability test for a given partitioning scheme. The resource partition model will be used in the following to model the root-level schedule, as it can be seen as an abstraction of existing static resource partitioning approaches. The class of dynamic root-level schedulers all rely on the paradigm of paravirtualization violating the full virtualization constraint of the problem statement (see section 4.1).

4.3 MODEL

The problem stated in section 4.1 first requires the modeling of the given real-time systems and the virtual system hosting these real-time systems. Thus, the definition of a given real-time system is given. The definition corresponds to figure 52.

Definition 4.3. *A real-time system* RS *is given as a 4-tuple* $RS_i(\Gamma, \Sigma, ISA, f)$. $\Gamma_i = \{\tau_i(T_i, C_i) | i = 1 ... n_i\}$ *represents the taskset being scheduled by the scheduling algorithm* Σ_i. *The taskset* Γ *is a periodic taskset*

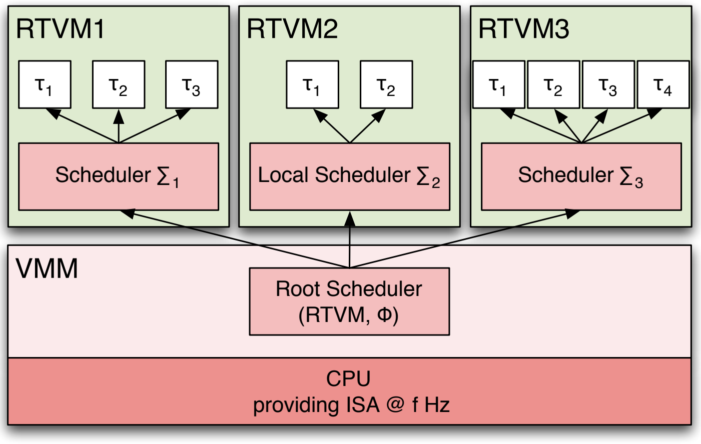


Figure 53: Model of a virtual real-time system

(see 2.1.1). The applied scheduling algorithm Σ is assumed to be either RM (see2.1.2) or EDF (see 2.1.2). The parameter ISA *represents the instruction set architecture of the hardware platform where the real-time system is compiled for. The parameter* f *represents the clock-rate to which the worst case execution time* C_k *of the task* $\tau_k \in \Gamma$ *refers to.*

Now the virtual real-time system (see figure 53) hosting multiple real-time systems is defined:

Definition 4.4. *A virtual real-time system* $\mathrm{VRS}((\mathrm{RTVM}_1, \mathrm{RTVM}_2, ..., \mathrm{RTVM}_n), \Phi)$ *is given by a 2-tuple of real-time systems being encapsulated as real-time virtual machines (RTVMs). These RTVMs are spatially and temporarily isolated by a VMM, which fulfills the VMM properties of Popek and Goldberg (see 2.2.2). The temporal isolation is ensured by the time partitioning policy* Φ.

The RTVMs $\mathrm{RTVM}_1, ..., \mathrm{RTVM}_n$ are either defined directly or can be derived from the given real-time systems $\mathrm{RS}_1, ..., \mathrm{RS}_n$. For the derivation of the RTVMs, it is further assumed that the given real-time systems share a common ISA and their clock rate is known. This allows to determine the clock frequency of the virtual host system. This simplification restricts the direct application to identical processor architectures used in the RTVMs in this thesis, which is not necessary in general since virtualization offers the emulation of different ISAs.

Other effects induced by the processor architecture like caching, pipeling and so on are not considered either, since this is covered by the research area for determining worst case execution times and Thus, is out of scope of this thesis.

The time partitioning policy Φ can be realized by applying one of the approaches presented in section 4.2. Since the goal of this thesis is to derive a fully virtualized virtual real-time system from given real-time systems RS or virtual real-time machine RTVM, a method based on single time slot periodic partitions (STSPPs) is presented to derive the partitioning policy Φ for the FVBTP. As local schedulers Σ_i either RM or EDF are assumed.

In the following, the problems of determining the CPU requirements and the derivation of a feasible root-level schedule for the integrated virtual system will be discussed. The sequential steps to find a solution for this problem are depicted in figure figure 54. First, a normalization of all given real-time systems is performed to determine a scaling factor in the next step. After the normalization and scaling steps have been performed, a valid root-level schedule is derived based on the concept of resource partitions introduced in section 4.2.1. This methodology will further be called *Full Virtualization by Temporal Partitioning* (FVBTP).

Finally, an evaluation based on hardware developed in CRC614 is presented to show the applicability and the performance of FVBTP. FVBTP has been published in [KBS09, KBG10].

4.4 TRANSFORMATION OF REAL-TIME SYSTEMS INTO REAL-TIME VIRTUAL MACHINES

When the set of RTVMs is not explicitly defined by the system designer, this set needs to be derived from the given set of real-time machines RS_i. This is the case when performing a consolidation of given real-time systems to an integrated virtual real-time system. As already stated in section 4.3, the given real-time systems share the same ISA and each real-time system RS_i provides the parameter f_i representing its clock rate. This information is necessary to derive the necessary clock rate of the virtual real-time system to be able to handle the load of its RTVMs. This clock-rate is considered to be a hint to the system designer

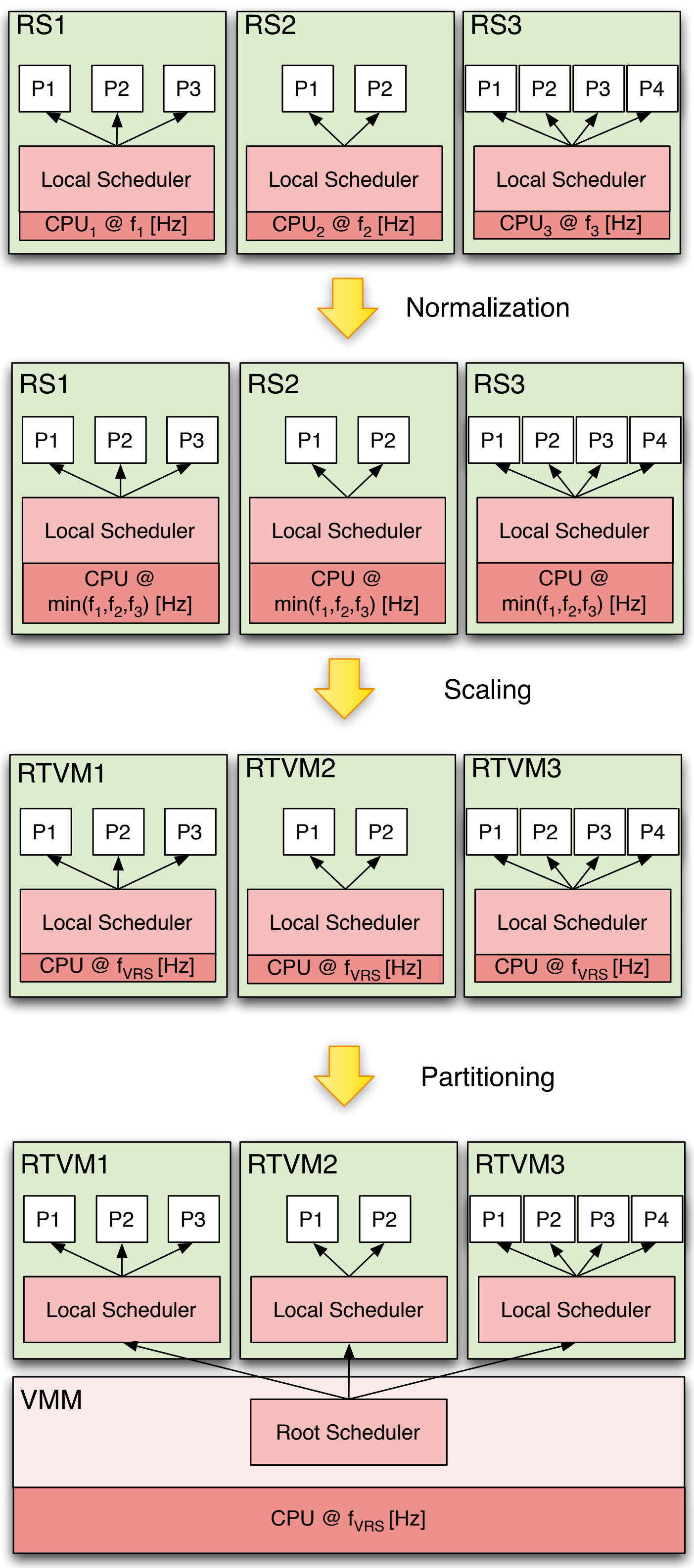


Figure 54: Methodology to realize full virtualization of RTVMs by temporal partitioning.

of the virtual real-time system, since the possible real-clock rates depend on the available hardware.

4.4.1 Normalization

To calculate the load of the virtual real-time system VRS, it is first necessary to calculate the load of the given real-time system RS_i. According to definition 2.5, the utilization of a given real-time system RS_i is calculated by

$$U(RS_i) = \sum_{RS_i(\Gamma)} \frac{C_k}{T_k} \tag{4.6}$$

First of all, it has to be noted that the WCETs C_k need to be the WCETs introduced in section 3.4.2, as the virtualization has an impact on the execution times. Since the given real-time systems RS_i may operate at different clock rates, the utilization of them cannot be summed up without normalizing them to a common clock rate. To achieve this, the clock rate f_i of each real-time system RS_i is used. The normalization is realized relative to the slowest real-time system RS_s

$$RS_s = RS_k | \forall i : f_k \leqslant f_i \tag{4.7}$$

Now it is possible to determine the normalization factor s_i to normalize each real-time system RS_i to the real-time system RS_s

$$s_i = \frac{f_i}{f_s} \tag{4.8}$$

Each normalization factor s_i shows how much faster RS_i is compared to RS_s.

Now, it is possible to normalize each real-time system under the assumption that each real-time system is executed on the slowest available hardware platform of RS_s. Thus, the execution times of the tasksets of each RS_i being executed on the hardware platform of RS_s have to be multiplied by its normalization factor s_i.

$$\begin{aligned} RS_{i_s} = (\{T_k, C_k \cdot s_i) | \tau_k(T_k, C_k) \in \\ RS_i(\Gamma)\}, RS_i(\Sigma), RS_i(ISA), f_s) \end{aligned} \tag{4.9}$$

Obviously, the utilization of each normalized real-time system RS_{i_s} is also multiplied by its normalization factor s_i

$$U(RS_{i_s}) = \sum_{RS_{i_s}(\Gamma_i)} \frac{C_k \cdot s_i}{T_k} = s_i \cdot U(RS_i) \tag{4.10}$$

4.4.2 Scaling the virtual real-time system

When executing all normalized real-time systems RS_{i_s} encapsulated as RTVMs on the hardware platform of R_s, it is very likely that this hardware platform is overloaded. Thus, it may be necessary to find a suitable hardware for the virtual real-time system to execute all real-time systems as RTVMs. This is achieved by determining the overall utilization of the normalized system and using this utilization as a speedup factor S being multiplied by the clock rate f_s of the normalized system. This results in the final clock rate f_{VRS} being the clock rate of the virtual real-time system hosting the RTVMs. Thus, the RTVMs can now be derived by applying S to the normalized real-time systems RS_{i_s}.

$$\begin{gathered} RTVM_i = (\{(T_k, \frac{C_k}{S}) | \tau_k(T_k, C_k) \in \\ RS_{i_s}(\Gamma)\}, RS_{i_s}(\Sigma), RS_{i_s}(ISA), f_{VRS}) \end{gathered} \tag{4.11}$$

The utilization of the single RTVMs and the virtual real-time system VRS can easily be calculated by:

$$U(RTVM_i) = \sum_{RTVM_i(\Gamma)} \frac{C_k}{T_k \cdot S} = \frac{1}{S} \cdot U(RS_{i_s}) = \frac{s_i}{S} \cdot U(RS_i) \tag{4.12}$$

$$U(VRS) = \sum_{i=0}^{n} U(RTVM_i). \tag{4.13}$$

The identification of the speedup factor S depends on the applied scheduling algorithm Σ_i of each real-time system, because the scheduling algorithms may differ in their utilization bound. In general, it is not possible to use the total utilization $U(VRS_s)$ of the normalized virtual real-time system when there are schedulers having a utilization bound less than one. Using the total utilization $U(VRS_s)$ as speedup may lead for example in the case of RM to the preemption of low priority tasks by high priority tasks before the low priority tasks are able to finish before their deadline. Thus, the speedup factor S needs to be relative to the utilization bound of the applied scheduling algorithm . For each scheduling algorithm σ the utilizations of the normalized real-time systems $U(RS_{i_s})$, relative to their utilization bounds, are summed up to build the speedup factor S_σ.

$$\alpha_i = \frac{1}{U_{lub}(RS_{i_s})} \tag{4.14}$$

$$S_\sigma = \sum_{i|\forall i:RS_{i_s}(\Sigma_i)=\sigma} \alpha_i \cdot U(RS_{i_s}) \tag{4.15}$$

It is assumed that either EDF or RM is used as local scheduler within the scope of this thesis. Thus, the final speedup factor S is given by:

$$S = S_{EDF} + S_{RM} \tag{4.16}$$

With this speedup factor S, it is possible to derive the required clock-rate for the VRS hosting the real-time systems RS_i as real-time virtual machines $RTVM_i$. This is simply done by multiplying the clock-rate of the slowest system RS_s, used for normalization, by the speedup factor S.

$$f_{VRS} = f_s \cdot S \tag{4.17}$$

This clock rate f_{VRS} denotes the minimum required clock-rate for the VRS. As there is hardly any hardware platform providing exactly this clock rate, the next faster one is to be chosen.

EDF

The speedup factor S_{EDF} for EDF-based real-time systems is the sum of their normalized utilization, as the utilization bound of EDF is 1. Thus, $\alpha_i = 1$

$$S_{EDF} = \sum_{i|\forall i:RS_{i_s}(\Sigma_i)=EDF} U(RS_{i_s}) \tag{4.18}$$

RM

Since the utilization bound of RM depends on the task set, the speedup factor S_{RM} needs to be defined relative to the least upper bound of the tasksets. Thus, $\alpha_i = \frac{1}{U_{lub}(RS_{i_s})}$
The speedup factor S_{RM} for RM-based real-time systems is:

$$S_{RM} = \sum_{i|\forall i:RS_{i_s}(\Sigma_i)=RM} \frac{1}{U_{lub}(RS_{i_s})} \cdot U(RS_{i_s}) \qquad (4.19)$$

4.4.3 Summary

This section described a methodology for deriving the clock-rate of the VRS to prevent the system from being overloaded by the executed RTVMs. Therefore, a normalization to a common base system (namely the slowest) is performed. Afterwards, the utilization of this system, considering the utilization bounds of EDF an RM, is determined. This utilization is chosen to be the speedup factor S applied to the clock-rate of the common base system to determine the final clock-rate of the VRS. This methodology only ensures the VRS from being overloaded, but it does not ensure the RTVMs to be schedulable by a root-level scheduler with an arbitrary partitioning policy. Thus, the next section covers the problem of ensuring the feasibility of the single RTVM schedule while having them scheduled by a root-level scheduler with a specific partitioning policy.

4.5 PARTITIONING POLICY

The first question to answer is which kind of partitioning policy is suited for the given requirements in section 4.1. The class of existing dynamic root-level schedulers is not suitable, because the implementation requires paravirtualization and thus violates the requirement of full virtualization. Thus, the remaining class to consider is the class of static root-level schedulers. Within this class, the static resource partition model is a generalization of the static proportional share and timeline scheduling algorithms. This allows the definition of arbitrary static partitions using either STSPP or MTSPP. STSPPs have the advantage of a simple modelling of a partition: required is only the definition of the partition period P and of one time interval during which the partition is active, which will further be called *activation slot*.

There are two problems being addressed when using static resource partitions. One is the determination of the activation slots γ_i of a static resource partition Π_i, and the other one is the determination of its period P_i. Both problems have to be solved under the real-time constraints given by the taskset $RTVM_i(\Gamma)$ being executed on Π_i. Thus, the choice of the proper activation slots and the proper period are essential to fulfill the given real-time constraints.

Another question is whether to choose STSPPs or MTSPPs as static resource partitions Π_i. Considering the switching overhead, MTSPPs with N activation slots introduce more switching overhead than a STSPP when their period P_{MTSPP} is chosen to be smaller than N times the period P_{STSPP} of a STSPP providing an equivalent resource allocation.

$$P_{MTSPP} < N \cdot P_{STSPP}. \tag{4.20}$$

An example to clarify this is depicted in figure 55. There are three static resource partitions given. The first has a period of 9), the second has a period of 5 and the third has a period of 11. All partitions provide a bandwidth of 40%, but differ in their switching overhead. Π_1 has a period of 9 being less than twice the period of the STSPP Π_2. According to equation 4.20 the expected switching overhead of Π_1 is higher than in the case of Π_2. Figure 55 shows this effect. Counting the switches up to time $t = 45$ results in 10 VM context switches caused by Π_1 while Π_2 causes 9 VM context switches and Π_3 causes 8 VM context switches. Thus, the choice of the period is essential for determining whether to use STSPPs or MTSPPs.

4.5.1 Activation slots

The derivation of the RTVMs presented in section 4.4.2 determines which bandwidth a resource partition needs to provide for a given $RTVM_i$. When assuming STSPPs as partitioning pol-

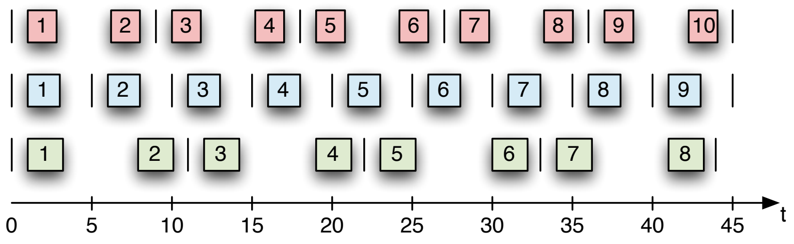


Figure 55: Comparison of switching overhead of STSPPs and MTSPPs

icy and P and S_1 are being chosen arbitrarily, the activation slot (S_1, E_1) of Π_i for VM_i can be calculated by:

$$E_1 = S_1 + \alpha_i \cdot U(RTVM_i) \cdot P \tag{4.21}$$

There are $P - (E_1 - S_1)$ possibilities to place this STSPP within the period P, because the slot must be placed in a way not to exceed the period P.

When assuming MTSPPs as partitioning policy, the activation slots $\{(S_1, E_1), ..., (S_N, E_N)\}$ of Π_i also need to fulfill the bandwidth requirement of $RTVM_i$:

$$\frac{\sum_{j=1}^{N} E_j - S_j}{P} = \alpha_i \cdot U(RTVM_i) \tag{4.22}$$

When defining the activation slots using either equation 4.21 or 4.22, it is ensured, that the fraction of the required resource bandwidth to the allocated resource bandwidth is equal to the utilization bound of the applied scheduling algorithm $RTVM_i(\Sigma)$:

Theorem 4.1. *Let the period* P *of a static resource partition* Π_i *be arbitrarily chosen. Then the ratio of the required utilization* $R = U(RTVM_i)$ *to the allocated utilization* $A = \frac{\sum_{j=1}^{N} E_j - S_j}{P}$ *is equal to the utilization bound* $\alpha_i = \frac{1}{U_{lub}(RTVM_i)}$ *of* $RTVM_i(\Sigma)$ *scheduling the associated taskset* $RTVM_i(\Gamma)$.

Proof. Figure 56 shows how the absolute times for the required computation time and the allocated computation time within the period P are calculated. By dividing the required computation

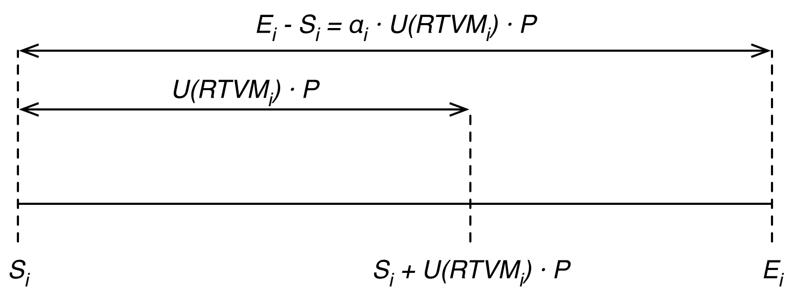

$E_i - S_i = \alpha_i \cdot U(RTVM_i) \cdot P$
$U(RTVM_i) \cdot P$
S_i
$S_i + U(RTVM_i) \cdot P$
E_i

Figure 56: Graphical illustration of the required computation time within the allocated interval $E_i - S_i$ of Π_i

time $R = U(RTVM_i) \cdot P$ by the allocated computation time $A = E_i - S_i = \alpha_i \cdot U(RTVM_i) \cdot P$, we obtain

$$\frac{R}{A} = \frac{U(RTVM_i) \cdot P}{\alpha_i \cdot U(RTVM_i) \cdot P} = \frac{1}{\alpha_i} = U_{lub}(RTVM_i(\Gamma))$$

For MTSPPs the theorem directly follows by rewriting equation 4.22.

□

By applying the methodology presented in section 4.4 to derive the RTVMs and the derivation of the activation slots presented in this section it is possible to guarantee that the ratio of the required computation time to the effectively allocated computation time over the period P is equal to the utilization bound of the applied local scheduler. The remaining problem to be solved is the choice of the period of the static resource partition. This will be addressed in the following section.

4.5.2 Period of the static resource partitions

When choosing the period for a static resource partition, the schedulability of the taskset executed on this partition needs to be ensured. This requirement does not permit to choose the period arbitrarily. An example for this is depicted in figure 57. The example shows the scheduling of two tasksets $RTVM_1(\Gamma) = RTVM_2(\Gamma = \{(8,2),(10,2.5)\}$ being executed on $\Pi_1 = \{(0,4),8\}$ and $\Pi_2 = \{(4,8),8\}$. The tasks of $RTVM_1(\Gamma)$ are mapped to

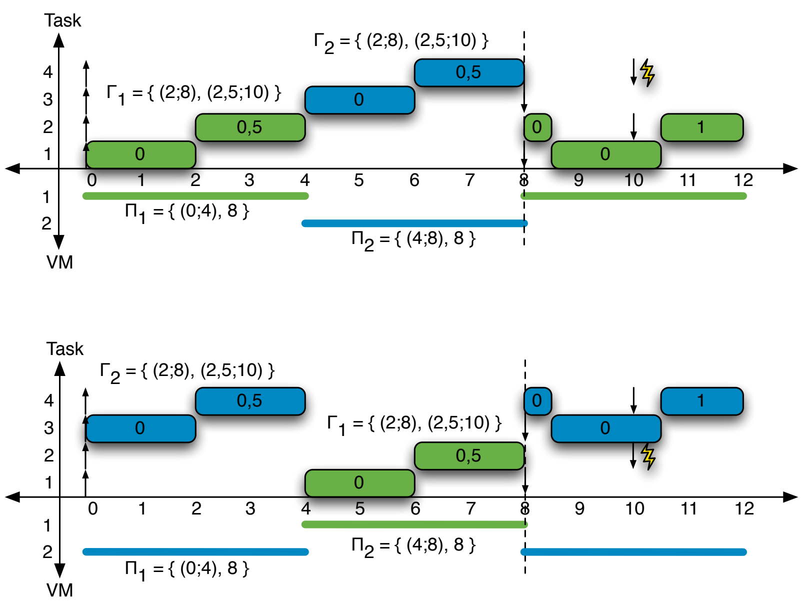

Task
Γ_2 = { (2;8), (2,5;10) }
Γ_1 = { (2;8), (2,5;10) }
Π_1 = { (0;4), 8 }
Π_2 = { (4;8), 8 }
VM
0,5
1

Figure 57: Example for the choice of an incompatible period length for static resource partition

numbers 1 and 2, while the tasks of RTVM$_2(\Gamma)$ are mapped to numbers 3 and 4.The first schedule shows the execution of RTVM$_1(\Gamma)$ on Π_1 and RTVM$_2(\Gamma)$ on Π_2, while the second schedule shows the execution of RTVM$_1(\Gamma)$ on Π_2 and RTVM$_2(\Gamma)$ on Π_1. Both tasksets are scheduled by EDF. The utilization of RTVM$_1$ is $U(RTVM_1) = 0.5$, and the utilization of RTVM$_2$ is $U(RTVM_2) = 0.5$. Both resource partitions provide a bandwidth of $\frac{4-0}{8} = \frac{8-4}{8} = 0.5$. The ratio of the required utilization to allocated utilization is 1 in all cases. Thus, the partitions are fully utilized and are not overloaded. The utilization of the partitions is equal to the utilization bound of EDF. Thus, a scheduling of the tasksets should be in general possible when considering only the utilization. When taking a look at the schedules of figure 57, one can see that there is a deadline miss at time $t = 10$ in both schedules. Now the choice of the period comes into play, as this is the reason why the deadlines are missed. Remembering the supply function (see figure 48 introduced in section 4.2.1) to show the impact on quantization, it is possible to find the problem. The supply function

$$C(\Pi, t) = \sum_{i=0}^{t} \text{active}(\Pi(\gamma, P), t) \tag{4.23}$$

$$\text{active}(\Pi(\gamma, P), t) = \begin{cases} 1 & \text{if} \quad (t \bmod P) \in \gamma \\ 0 & \text{else} \end{cases} \tag{4.24}$$

represents the accumulated activation time up to time t of Π_i, while the assigned utilization function

$$U_A(\Pi, t) = \frac{C(\Pi, t)}{t} \tag{4.25}$$

denotes the percentual part of the processor assigned to Π_i up to time t.

For each partition of the first schedule, $C(\Pi, t)$ and $U_A(\Pi, t)$ are depicted in figure 58. Due to the quantization introduced by the partitions Π_1 and Π_2, the real supply deviates from the idealized supply when assuming infinite time slicing ($P \to 0$). Note that in general, there are individual supply functions for each partition, but the example contains two partitions with identical supply functions. As one can see, the supply function for both VMs deviates from the idealized supply at time $t = 10$. The one with the supply function value below the idealized supply misses its deadline at time $t = 10$. Thus, an arbitrary choice of the period P is not possible when executing hard real-time tasksets on static resource partitions.

Lemma 4.2. *When* $\frac{\sum_{\gamma_i} E_j - S_j}{P_i} = \frac{1}{U_{lub}} \cdot U(RTVM_i)$ *for* γ_i *of the static resource partition* $\Pi_i(\gamma_i, P_i)$ *and* P_i *is chosen as* $P_i = \gcd(\{T_k | \tau_k(T_k, C_k) \in RTVM_i(\Gamma)\})$*, then no task* $\tau_k \in RTVM_i(\Gamma)$ *will miss its deadline, if* $RTVM_i$ *is transformed by* $RTVM_i = ((T_k, \frac{C_k}{S}) | \tau_k(T_k, C_k) \in RS_{i_s}(\Gamma)\}, RS_{i_s}(\Sigma), RS_{i_s}(ISA), f_{VRS})$ *with* $S = S_{EDF} + S_{RM}$.

Proof. To show that the assigned allocation is guaranteed at every single deadline of the taskset $RTVM_i(\Gamma)$, we assume that there exists a task $\tau_k \in RTVM_i(\Gamma)$ with deadline T_k where $U_A($

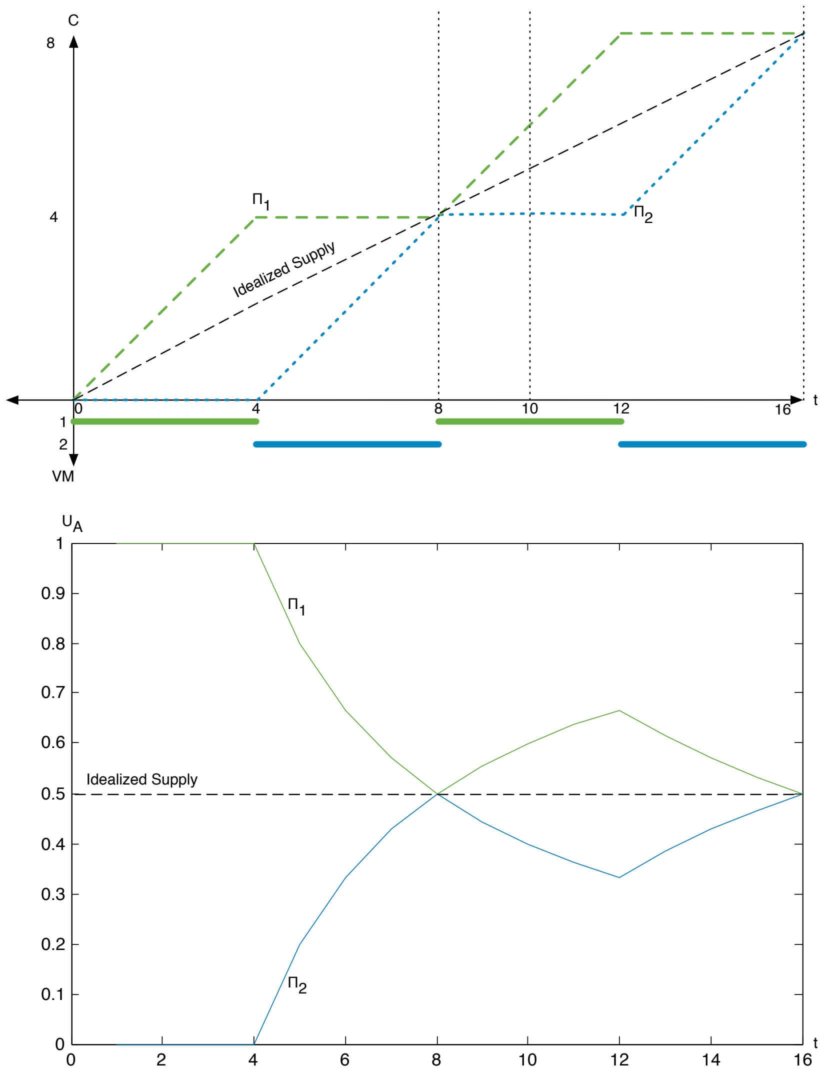


Figure 58: Assigned computation time and utilization over the real time elapsed of schedule 1 of figure 57

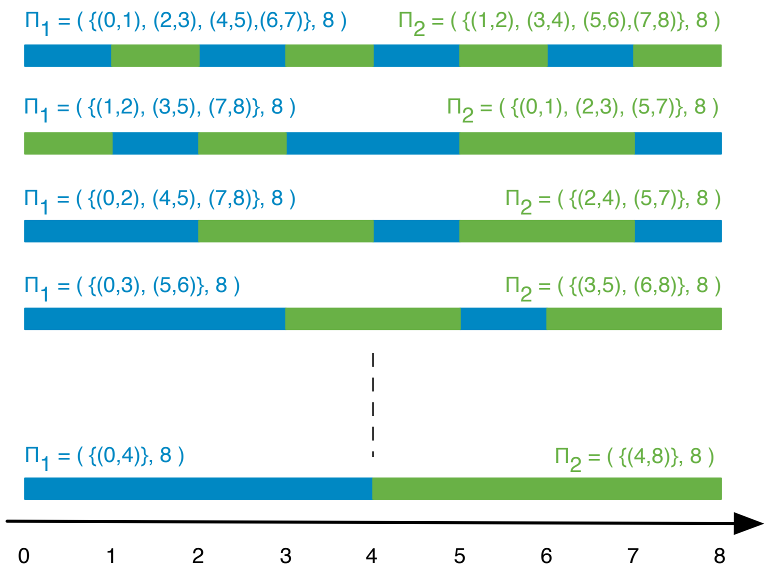

Π1 = ({(0,1), (2,3), (4,5),(6,7)}, 8)
Π2 = ({(1,2), (3,4), (5,6),(7,8)}, 8)
Π1 = ({(1,2), (3,5), (7,8)}, 8)
Π2 = ({(0,1), (2,3), (5,7)}, 8)
Π1 = ({(0,2), (4,5), (7,8)}, 8)
Π2 = ({(2,4), (5,7)}, 8)
Π1 = ({(0,3), (5,6)}, 8)
Π2 = ({(3,5), (6,8)}, 8)
Π1 = ({(0,4)}, 8)
Π2 = ({(4,8)}, 8)
0
1
2
3
4
5
6
7
8

Figure 59: Example for scheduling activation slots of a two periodic resource partitions Π_1 and Π_2 with period $P_1 = P_2 = 8$ and $U_1 = U_2 = \frac{1}{2}$.

$\Pi_i, t)$ at time $t = T_k$ is smaller than the required utilization $\alpha_i \cdot U(RTVM_i)$.

$$\exists \tau_k \in RTVM_i(\Gamma) : U_A(\Pi_i, T_k) < \alpha_i \cdot U(RTVM_i)$$

Due to T_k being a divider of P_i theorem 4.1 can be applied. Thus, $U_A(\Pi_i, T_k)) = \alpha_i \cdot U(RTVM_i)$. Inserting this into the claim leads to

$$\alpha_i \cdot U(RTVM_i) < \alpha_i \cdot U(RTVM_i)$$

at time $t = T_k$, which is a contradiction. Thus, the given taskset $RTVM_i(\Gamma)$ is schedulable on the static resource partition Π_i by the scheduling algorithm $RTVM_i(\Sigma)$. □

4.5.3 Schedule

Up to now, the questions on how to derive the activation slots and the period of a static resource partition guaranteeing the real-time constraints of a single RTVM have been answered. The question is how to schedule the activation slots of the RTVMs within the virtual real-time system to guarantee that the real-time constraints for all RTVMs will be answered. Figure 59 depicts multiple examples of scheduling the activation slots of two periodic resource partitions with same period and equal utilization. Thus, each periodic resource partition needs to provide half of the period as computation time. Each line represents a schedule, which can either be a MTSPP with two, three or four activation slots, or a STSPP with only one activation slot.

As shown in section 4.5.2, the period of a static resource partition is chosen as the greatest common divisor of all deadlines of the taskset executed by the RTVM to ensure the schedulability of a single RTVM when the RTVM is derived by the transformation presented in section 4.4. This period is determined independently of the type of static resource partition. Thus, the partition may either be a STSPP or a MTSPP. Nevertheless, STSPP offer less switching overhead when having a period $P_{STSPP} > \frac{P_{MTSPP}}{N}$. Since the derivation of the period is independent of the partition type, the period would be equal for either a STSPP or a MTSPP, such that $P_{STSPP} = P_{MTSPP}$ and $N > 1$ for a MTSPP. Thus, the choice falls on STSPPs realizing the static resource partitions derived in section 4.5, as they introduce less switching overhead within the same period as a MTSPP.

However, it may be possible to create MTSPPs with longer periods than derived in section 4.5.2 and less switching overhead than a STSPP providing the same bandwidth. The problem then is to find such a appropriate MTSPP. Up to now, there is no better way known than using the schedulability test presented by Mok and Feng (see section 4.2.1) on each possible MTSPP for a RTVM.

The number of possibilities to generate a MTSPP for $RTVM_i$ with a given period P_i is given by

$$\binom{P_i}{\alpha_i \cdot U(RTVM_i) \cdot P_i}. \tag{4.26}$$

The term $\alpha_i \cdot U(RTVM_i) \cdot P_i$ represents the time to be allocated within period P_i according to theorem 4.1. Thus, this would be a factorial complexity of $O(P_i!)$ schedulability tests to be performed for each RTVM. An example for placing the activation

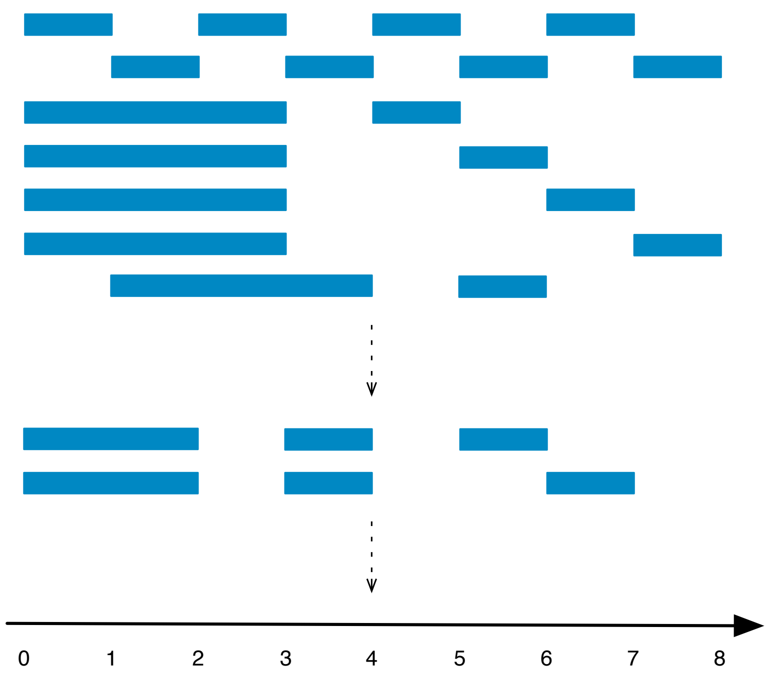


Figure 60: Example for placing the activation slots of a MTSPP

slots within the first RTVM is depicted in figure 60. Note that when placing multiple MTSPPs, the already allocated activation slots need to be considered to prevent a multiple allocation of a resource. When allocating the same activation slot to multiple MTSPPS, the number of MTSSPs sharing this slot defines the number of processor cores needed.

In the case of STSPPs, the number of placing a single STSPP within period P is restricted to $P-(E_1-S_1)$ (see section 4.5.1) possibilities. A test of all possibilities for n STSPPs would result in a complexity of $O(n \cdot P)$, as P_i of a real-time virtual machine $RTVM_i$ can be calculated as the gcd of its tasks deadline. Figure 61 shows an example for placing the activation slots of a STSPP.

As the resource being partitioned by the STSPPs must be allocated by only one partition at a time in case of a single core system. There might be no solution for STSPP combinations when the periods P_i for the STSPPs are determined independently us-

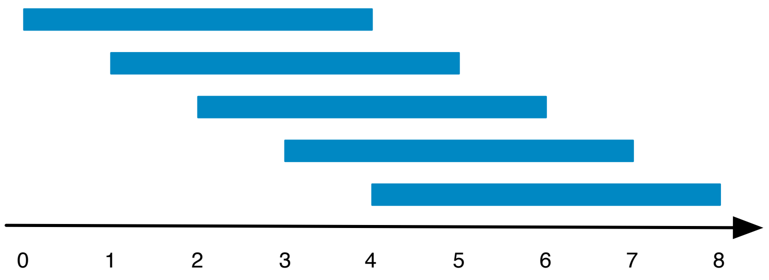


Figure 61: Example for placing the activation slots of a STSPP

ing the gcd for the local tasksets only. According to Lemma 4.2, it is possible to derive P_i as gcd of all deadlines in the system:

$$P_i = \gcd(\{T_k | \tau_k(T_k, C_k) \in \bigcup_{i=1,\ldots,n} RTVM_i(\Gamma)\}) \tag{4.27}$$

Thus, $\forall i = 1, \ldots, n : P = P_i$ with n being the total number of RTVMs within the VRS. This means each period P_i of each $STSPP_i$ is equal to the the period P. The period P is a divisor of all deadlines in the system and $\frac{\sum_{\gamma_i} E_j - S_j}{P_i} = \alpha_i \cdot U(RTVM_i)$ thus, due to theorem 4.2, all RTVMs are schedulable on the virtual real-time system VRS. Now, the remaining question is how to derive the activation slot lengths of each STSPP and how to schedule them.

This is achieved by starting with the first STSPP at the beginning and determining the length of the activation slot by multiplying the utilization of its RTVM by the length of the period P. Additionally this value also needs to be aligned relative to the utilization bound of the applied scheduling algorithm. Describing it in other words, the activation slots are concatenated one after another with their length being proportional to their utilization relative to the whole system.

$$S_1 = 0 \tag{4.28}$$

$$E_i = S_i + \alpha_i \cdot U(RTVM_i) \cdot P \tag{4.29}$$

$$S_i = E_{i-1} \tag{4.30}$$

Figure 62 shows a small example for three STSPPs with $P = \gcd \bigcup_{i=1}^{3} \{T_k | \tau_k \in RTVM_i(\Gamma)\} = 6$, $\alpha_1 \cdot U(RTVM_1) = \frac{1}{2}$, $\alpha_2 \cdot U(RTVM_2) = \frac{1}{6}$ and $\alpha_3 \cdot U(RTVM_3) = \frac{1}{3}$. The order in which

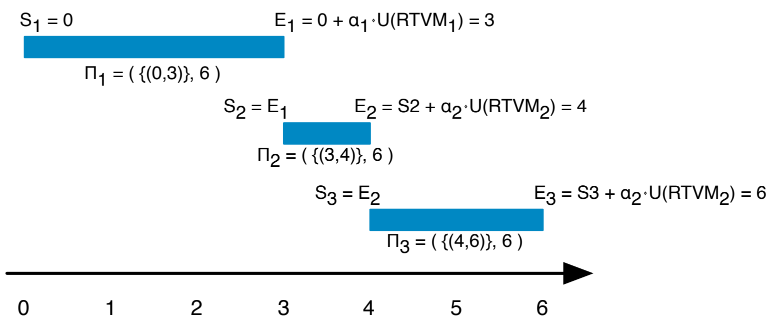

$S_1 = 0$
$E_1 = 0 + \alpha_1 \cdot U(RTVM_1) = 3$
$\Pi_1 = (\{(0,3)\}, 6)$
$S_2 = E_1$
$E_2 = S2 + \alpha_2 \cdot U(RTVM_2) = 4$
$\Pi_2 = (\{(3,4)\}, 6)$
$S_3 = E_2$
$E_3 = S3 + \alpha_2 \cdot U(RTVM_2) = 6$
$\Pi_3 = (\{(4,6)\}, 6)$
0
1
2
3
4
5
6

Figure 62: Example for placing the activation slots of three STSPPs to ensure the schedulability of the whole system

the STSPPs are placed is not relevant, as for each $RTVM_i$, the required utilization has to be provided not later than P. Reconsidering the example given for the invalid choice of the period P in figure 57 of section 4.5.2, it is now easily possible to derive a solution for this scenario which is depicted in figure 63.

4.5.4 Summary

The integration of software components into a big integrated system is the main goal of this thesis. The previous chapter built the infrastructural fundament by providing a highly deterministic VMM with extensible scheduler interface and the possibility to determine the WCET of the virtualized guests, which is required for the scheduling of the RTVMs.

Within this section, the derivation of the CPU requirements and the derivation of the root-level schedule have been presented under the assumption that only full virtualization is allowed due to licensing restrictions and that all real-time tasks meet their deadlines to realize the temporal isolation property. The related work presented in section 4.2 lacks this complete derivation of the root-level schedule from given real-time systems under the assumption of full virtualization.

First the derivation the CPU requirements of the virtual real-time system hosting the RTVMs was realized in section 4.4 in such a way that the system is not overloaded. A general transformation approach based on utilization bounds has been introduced to address this problem. As RTOS schedulers EDF and

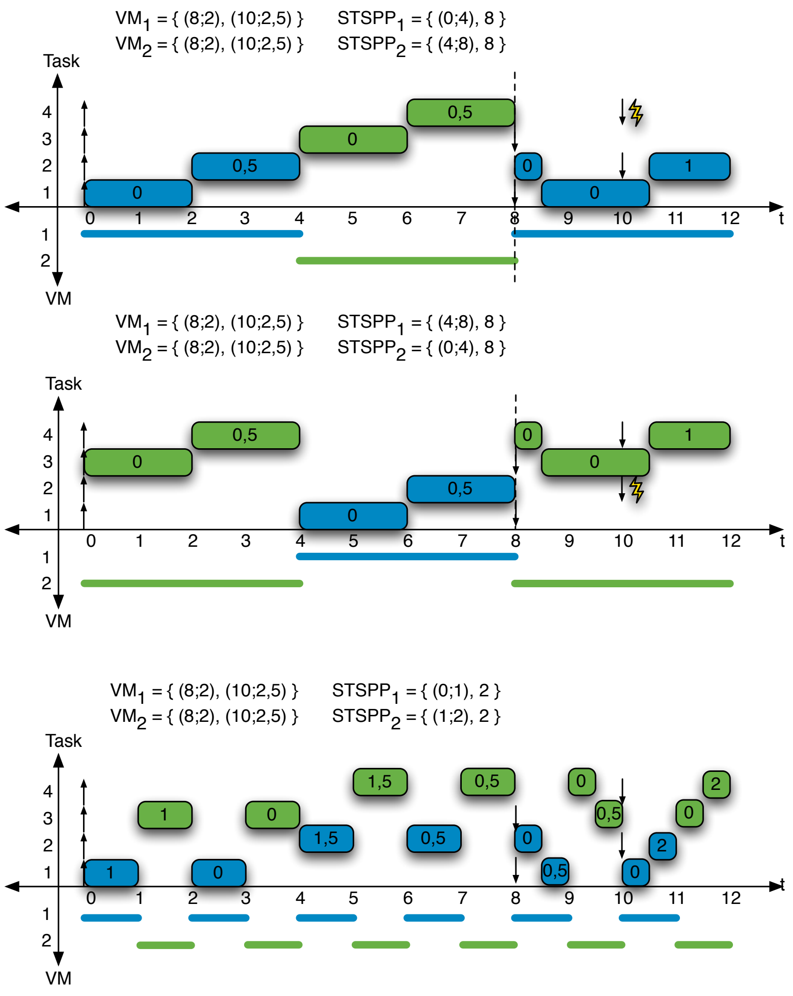

VM1 = { (8;2), (10;2,5) }
VM2 = { (8;2), (10;2,5) }
STSPP1 = { (0;4), 8 }
STSPP2 = { (4;8), 8 }
VM1 = { (8;2), (10;2,5) }
VM2 = { (8;2), (10;2,5) }
STSPP1 = { (4;8), 8 }
STSPP2 = { (0;4), 8 }
VM1 = { (8;2), (10;2,5) }
VM2 = { (8;2), (10;2,5) }
STSPP1 = { (0;1), 2 }
STSPP2 = { (1;2), 2 }
Task
VM
t

Figure 63: Example of a valid partitioned schedule with the period being the GCD of all deadlines instead of P = 8. The activation slots have been determined according to equation 4.28.

RM have been exemplarily analyzed and a scaling factor being applied to a normalized system was calculated. With this scaling factor it is possible to derive the minimal CPU speed f_{VRS} of the virtual real-time system. This derived CPU speed can be considered as a hint for the system designer, as in general there hardly is any hardware available with the desired clock rate and the assumed linear scaling behavior does not reflect accurately the reality[3]. Nevertheless the calculated speedup factor is still valid, even when its application as a linear factor to the clock rate is restricted.

Finally, the derivation of a root-level schedule was presented in section 4.5 by deriving the parameters of STSPP partitions. Each partition executes a RTVM and defines an interval within which this RTVM is executed. This interval is repeated periodically with the period of all STSPPs being the GCD of all task deadlines available in all RTVMs. The schedule of the intervals within the global period is derived as described in section 4.5.3. As already mentioned, the derivation of this partitioning is not covered in the presented related work. So the main contribution of this section is the possibility of automatically deriving a fulyl virtualized virtual real-time system from given real-time systems without requiring the interaction of the system designer.

4.6 EVALUATION

The evaluation of FVBTP from given real-time systems presented in this chapter is first addressed by summarizing the algorithmic complexity for each step of this approach. This algorithmic complexity is compared to the current state of the art approach from academia namely the open system environment (see 4.2.1). At first glance this seems to be a comparison of apples and oranges, since both approaches use different virtualization paradigms. A valid comparison would be to compare FVBTP with another methodology that derives the schedule of a fully virtualized hierarchical real-time system.The related work showed that there does not exist such a solution that completely derives a hierarchi-

3 There are a lot of effects affecting the speed of a program, such as pipelining and caching. Even when assuming they are not available, a linear scaling behavior for identical ISAs is restricted on the speed the memory bus, which defines the maximum speed of memory accesses.

Task	Complexity	
	Open System Environment	FVBTP
Initialization	O(1)	O(j)
Runtime	$O(n^2)$	O(1)

Table 10: Comparison of the algorithmic complexity of the open system environment and FVBTP

cal schedule for fully virtualized systems. Another question may be why not comparing this approach to existing static schedulers like the ARINC 653. This question is easy to answer, because FVBTP does not compete with this kind of schedulers due to the reason of essentially being an extension of this type of schedulers. In general, FVBTP is an abstraction of this class of schedulers providing a methodology to derive the execution parameters as extension. This has been reasoned in section 4.2.3. The main difference is the extension allowing the derivation of the complete virtual real-time system, what has not been addressed in this field up to now. The result of this approach can thus be transferred to an ARINC 653 Plattform. Thus, it is more interesting to see the strengths and weaknesses of the approach presented in this thesis compared to the open system environment as this approach uses paravirtualization, which is in general applied for improving virtualization performance (see 3.2.1). This is realized by evaluating the real execution of tasksets in the open system environment and with FVBTP. This evaluation was the focus of the master thesis[Grö10] of Stefan Groesbrink which was supervised by me. The results presented in this section are based on this work. Before presenting the evaluation based on the real execution hardware, the algorithmic complexity and the distribution of the GCD is discussed to get a feeling about the expectation of the real execution result.

4.6.1 Algorithmic complexity

The algorithmic complexity is determined for the initialization process and the runtime overhead of the open system environment and the approach presented in this thesis. Let n denote the number of RTVMs being executed and let j denote the number of all tasks available in the virtual real-time system. The transformation of the tasksets, namely the adaption of the worst case execution times to the underlying hardware (see 4.4), has a complexity of $O(j)$, as each task needs to be scaled to the target virtual real-time system. The derivation of the root-level schedule requires the calculation of j greatest common divisors to calculate the global period P and the activation slots for each STSPP. The worst case execution time of determining the gcd of two k-bit numbers which can be found in literature [Sed09] is $O(k^2)$. Current hardware architectures provide either 32 Bit or 64 Bit numbers. The gcd of j numbers can be determined by hierachical application of the gcd.

$$\begin{aligned} gcd(j_1, ..., j_n) = \\ gcd(j_1, gcd(j_2, gcd(j_3, ..., gcd(j_{n-1}, gcd_j)))) \end{aligned} \tag{4.31}$$

The number of iterations is bounded by O(j) [Bra70]. Thus, the total complexity of determining the gcd of j k bit numbers is $O(jk^2)$. Since k is constant, by either being 32 Bit or 64 Bit when considering actual hardware architectures, the complexity finally is $O(j)$. Both the transformation and the derivation of the root-level schedule have a complexity of $O(j)$ and thus, the total complexity of the initialization step is also $O(j)$.

The initialization of the partitioning policy of the open system environment only requires the initialization of the CBS servers what requires a constant time for each server. Thus, the complexity of the initialization process of the open system environment is $O(n)$ while being $O(j)$ for the FVBTP. Note that in general, there are more tasks than virtual machines $n \leqslant j$.

Finally, the complexity of the scheduler at runtime is determined. For the open system environment, this complexity is given by the maintainance of n servers and the determination of the next deadline among all servers. The maintenance can be realized in constant time, while the determination of the next deadline is given by the complexity of inserting one element into a sorted list. Using a linear list the complexity is given by $O(n)$. Thus, the total complexity is $O(n^2)$ for the runtime overhead of the open system environment. In case of FVBTP this complexity is constant, as the schedule can be stored as an array of activation slots being repeated after P time units. Thus, the runtime complexity is $O(1)$.

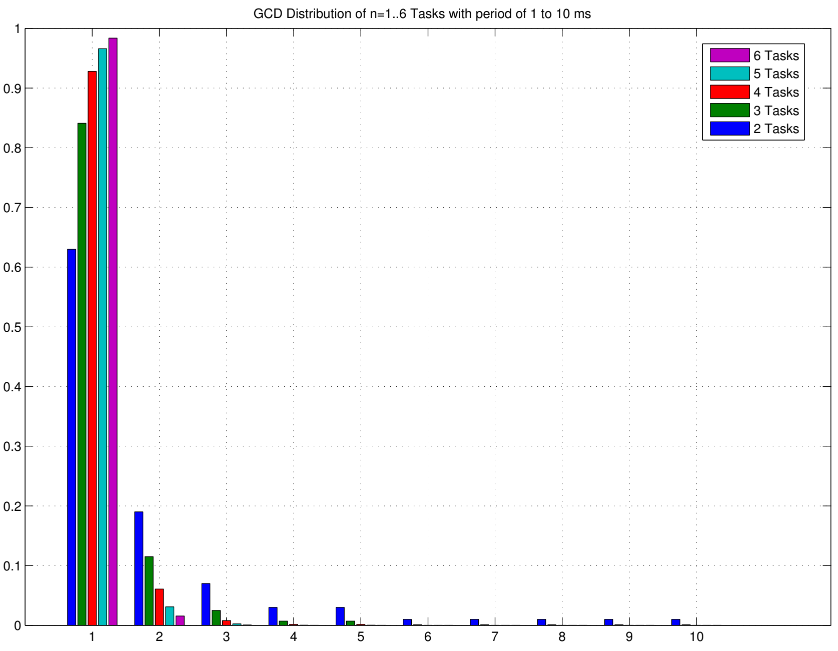


Figure 64: Distribution of the GCD for up to 6 tasks with periods up to 10ms

4.6.2 Distribution of the GCD

The performance of the FVBTP is heavily dependent on the result of the GCD of all deadlines available in the system, as small GCDs heavily increase the switching overhead within the schedule. Intuitively, one expects a lot of low GCD results for the possible combinations of deadlines that exist. Unfortunately, this expectation is correct, and figure 64 shows the distribution of the GCD for $n = 2, ..., 6$ tasks where all possible combinations of deadlines for periods $T = 1ms, ..., 10ms$ have been considered. As one can see, the distribution has its maximum at a GCD value of one and it rapidly decreases when the GCD is getting larger. When increasing the number of tasks, the probability of getting a GCD of one increases, while the probability of getting a higher GCD decreases. So theoretically, the approach of FVBTP seems to be unsuitable for most of the cases and seems to work only for seldom special cases. Fortunately, these special cases are not as seldom for hard-real-time systems as one would expect. This will be shown in the following sections where the evaluation on the real execution, platform with realistic scenarios is presented.

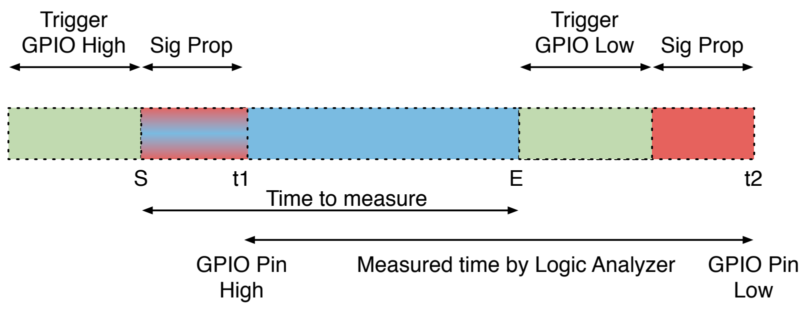


Figure 65: Possible measurement error due to signal propagating time.

4.6.3 Experimental Setup

As execution platform for the real execution the PowerPC405FX system as presented in section 3.4 was used, since the designed VMM was implemented on this platform. As a difference, the measurements in this section are determined by using an Agilent logic analyzer connected directly to 20 general purpose I/O pins (GPIOs) of the hardware platform. These GPIOs have been used to signal the different start and stop times of the evaluated parts of the VMM. The logic analyzer is able to detect rising and falling edges on these GPIOs with a sampling frequency of 800 MHz being more than two times higher than the frequency of the evaluated hardware. This ensures that the required property of the sampling frequency is at least two times higher than the sampled signal stated by the *Nyquist Shannon Sampling Theorem*[Sha49]. Thus, the worst case error induced due to the logic analyzer would be equal to $\pm 1.25ns$ for each detection of a falling or raising edge, as the sampled signal may just be hit shortly before or after the transition occurs. This error is equal to approximately a third of a processor cycle and is thus negligible for the following measurements. [Grö10]

Besides the error induced by the logic analyzer, the signal propagation delay from the triggering instruction to the GPIO needs to be determined when measuring time intervals. Figure 65 shows the effects occuring when measuring a specific time interval with the Logic Analyzer. [Grö10]

To determine the length of the interval, E and S have to be determined. The direct measurements of the GPIO signal lead to a delay between the finish time of the instruction triggering the

GPIO to high and the time at which the GPIO pin changes its state due to the signal propagation. Thus, the value of the GPIO pin raises to high when the time to measure may have already started. The instruction triggering the GPIO to change its state to low is executed directly after the time to measure, and again the delay due to signal propagation time delays the state change of the GPIO pin. Thus, the difference between S and E can be calculated as:

$$\begin{aligned} E - S &= t_2 - SigProp - TriggerInstr - (t_1 - SigProp) \\ E - S &= t_2 - t_1 - TriggerInstr \end{aligned}$$

So the signal propagation time has no influence on measuring the time interval. The only error is given by the execution time of the instruction that triggers the GPIO. The worst case would be the alignment of this instruction on the beginning of a cache line that needs to be fetched for execution. The WCET of handling the line fill has been already determined in section 3.4.2 and is equal to 52 cycles. Additionally the access to the GPIO is realized by memory mapped I/O which introduces an additional overhead of 52 cycles for the memory access. Thus, the error is 78 cycles $\pm$ 26 cycles, because the memory access is always required while the line fill may not occur. This error needs to be considered for each taken measurement performed in the following analysis.

4.6.4 Scheduling performance

For determining the scheduler performance, the execution times of the methods specified in the extensible scheduler interface (see 3.3.5) have been determined using the experimental setup described in the previous section. The measurements can be found in A.1 and are taken from [Grö10].

The method *sched_init* is responsible for the initialization of the schedulers. As expected, the overhead induced by determining the schedule offline in case of FVBTP is significantly larger than in case of the Open System environment initialization phase. The complexity of determining the root-level schedule is $O(j)$ with j being the number of available real-time tasks in the system (see 4.6.1). In case of the Open System Environment, the initializa-

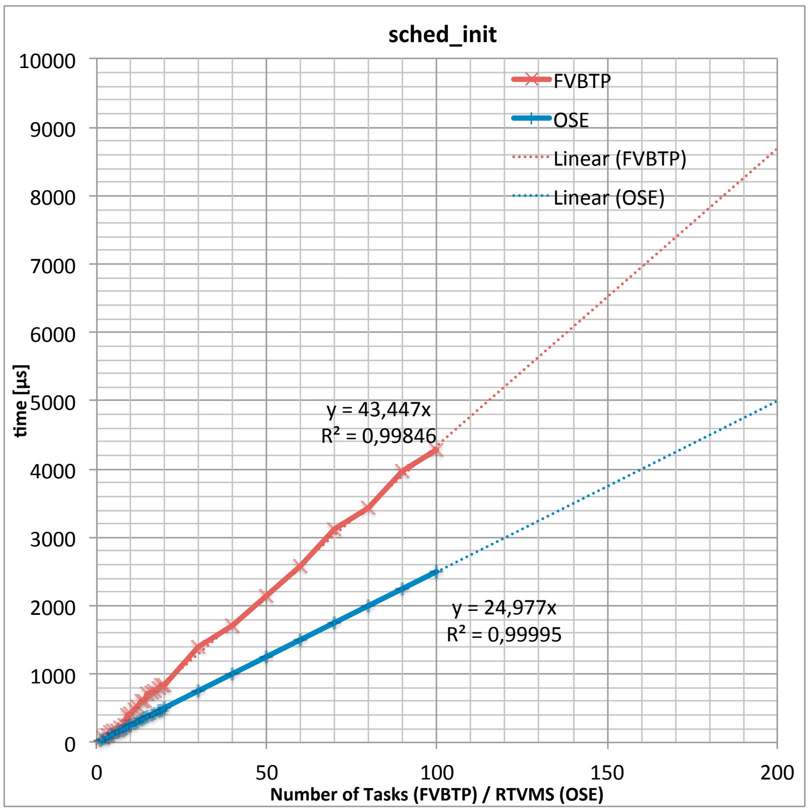


Figure 66: Scaling behavior of sched_init. For measurement values see table 22, 23 and 26.

tion step is $O(n)$ with n being the number of RTVMs (see 4.6.1). Figure 66 shows the scaling behavior of *sched_init* with an increasing number of tasks/RTVMs. As one can see, the expected linearity of the initialization phase is also represented in the measurements. To give a prediction, a linear regression was applied to the available measurements, and as a result, a gradient of $a = 42.663[\frac{\mu s}{Task}]$ was determined with a coefficient of determination of $R^2 = 0.99744$, which implies a very good predictability.

As expected, the offline overhead of FVBTP is higher than the overhead of the open system environment. Nevertheless, even for a large virtual real-time system with 1000 tasks, the FVBTP offline derivation of the schedule would only need about 42 milliseconds on a 300MHz PowerPC. Thus, even a reset of the system requiring a recalculation of the schedule seems to be realizable without any problem of violating the timing constraints for resetting such a system.

Scheduler	Min[μs]	Max[μs]	Mean[μs]	Confidence Interval$_{0.95}$
FVBTP	0.917	1.240	1.033	[1.017; 1.050]
OSE	0.920	1.200	1.037	[1.021; 1.054]

Table 11: Performance of scheduler interface routine sched_getNextTimerEvent [Grö10].

The next method of the extensible scheduler interface to be analyzed is the method *sched_getNextVMIndex*, which is the essential scheduling method that determines the next VM to be executed out of the ready queue. In the case of FVBTP, this method is expected to show constant execution time behavior, while in the case of the OSE this method is expected to show $O(n^2)$. Figure 67 shows the measured behavior for a growing number of virtual machines. As one can see, OSE shows quadratic behavior for a growing number n of RTVMs with a coefficient of determination of $R^2 = 0.9996$ The FVBTP *sched_getNextVMIndex* shows the expected constant behavior and is even in the case of a small set of RTVMs (n=6) significantly lower (10 times) than the *sched_getNextVMIndex* of the OSE.

As already stated at the beginning of this evaluation section, this comparison is quite unfair, as OSE is an online and FVBTP is an offline scheduler. Nevertheless, FVBTP had to be compared to an existing approach, and as FVBTP is the only known offline approach, a comparison to the state of the art online approach was obvious.

Finally, the performance of the method *sched_getNextTimerEvent* had to be analyzed. In both cases, this method returns the point in time where a timer interrupt is set to reactivate the VMM for scheduling reasons. In the case of OSE, this is the next deadline which is accessible in $O(1)$ by reading the relative deadline from the first tasks PCB of the ready queue and adding this value to the current time. In the case of FVBTP, the next timer event is determined by the endpoints of the activation slots, which are also accessible in $O(1)$. Table 11 shows the result of the measurements for FVBTP and OSE. Both methods show a nearly identical performance at $0.9\mu s$.

As expected, FVBTP shows the better runtime performance for the scheduling decision and a worse performance for the ini-

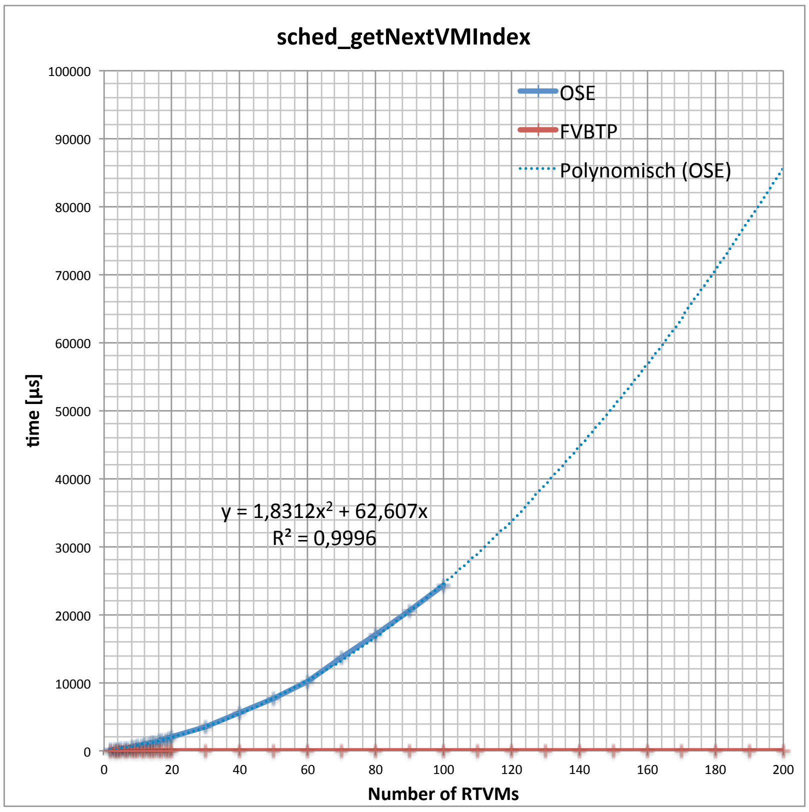


Figure 67: Scaling behavior of sched_getNextVMIndex

tialization phase when comparing it to OSE. However, there is still an issue not addressed up to now that has significant influence on the overhead introduced by those schedulers. This issue is the amount of overhead caused by VM context switchings. Therefore, two values need to be determined. The first one is the introduced overhead of a context switch and the second is the number of context switches caused by the scheduler. The first question will be answered in the following section and the second question will be addressed by performing a scenario based evaluation using realistic scenarios.

4.6.5 Context Switching

Like any other system software enabling the use of multiprogramming on a single core system, virtualization adds overhead due to switching between the virtualized environments. This switching process is in general identical to a classical context switch known from operating systems and consists of the following steps

1. Save context (Hardware Registers, Virtual Registers, Virtual memory data structures)
2. Scheduling decision
3. Restore context of next VM

Such a virtual machine context switch (VMCS) needs to be performed every time a VM is suspended and another VM becomes active. The VMCS starts from the point in time where the last VM was suspended and ends when the next VM starts its execution. During this time, only the VMM is active. The steps 1 and 3 are independent from the applied root-level scheduling policy, while step 2 is directly influenced by the scheduling policy. To quantify the VMCS, the save and restore process needs to be determined. With this result, the VMCS can be derived for a specific scheduling scenario. Table 12 shows the result of the VMCS measurements performed for the OSE scheduler with two virtual machines, and table 13 shows the results for the FVBTP scheduler. As one can easily see, the save and restore process needs the same amount of time of about $33\mu s$ and only differs in orders of $500ns$ in the mean which can occur due to code positioning. The time needed for the scheduling

OSE	Min[μs]	Max[μs]	Mean[μs]	Confidence Intervall$_{0.95}$
Total	129.123	130.042	129.545	[129.512;129.579]
context	33.087	33.969	33.716	[33.687; 33.746]
schedule	95.522	96.162	95.823	[95.764; 95.852]

Table 12: VMCS for Open System Environment for $n = 2$. [Grö10]

includes the calls of the extensible scheduler interface methods *sched_getNextVMIndex* and *sched_getNextTimerEvent*. The difference of about 5μs to the sum of the measured values is in both cases caused by the code embedding the calls to these methods. The scheduling decision itself is the dominating part in case of OSE. Already in the case of two virtual machines, the scheduling overhead is about three times larger than the overhead induced by saving and restoring the context. In case of FVBTP, the schedule decision takes about a third of the time to save and restore the context. Thus, the complexity of VM switching is in the same order of magnitude of a normal context switch. The main problem of OSE is the fact that the scheduling decision in the worst case grows quadratically in the number of virtual machines, what leads to an even higher impact of the scheduling decision upon a VMCS. The main problem of FVBTP is a small GCD. To show the impact of the GCD on the VMCS overhead, figure 68 shows the expected overhead for a GCD ranging from $1\mu s$ to $10ms$ for 2 up to 10 VMs. This overhead was determined by

$$\text{Overhead}(GCD, VMs) = \frac{(VMs - 1) \cdot VMCS_{FVBTP}}{GCD} \tag{4.32}$$

It is easy to see the high impact of GCDs in the magnitude of μs on the switching overhead for the used evaluation platform. When restricting the granularity in time to milliseconds, FVBTP seems to be applicable in most of the cases. This restriction seems also be valid for OSE, as the complexity for a single VMCS is already in the order of $0.13ms$ in case of two tasks. The final question to answer is how many VMCS are performed by OSE and FVBTP in realistic scenarios, as this allows a final evaluation of the overhead introduced by both scheduling approaches. This discussion will be performed in the following section using different realistic scenarios.

FVBTP	Min[μs]	Max[μs]	Mean[μs]	Confidence Intervall$_{0.95}$
Total	42.480	42.962	42.751	[42.725; 42.776]
context	32.977	33.577	33.281	[33.251; 33.311]
schedule	9.360	9.643	9.465	[9.450; 9.479]

Table 13: VMCS Performance for FVBTP. [Grö10]

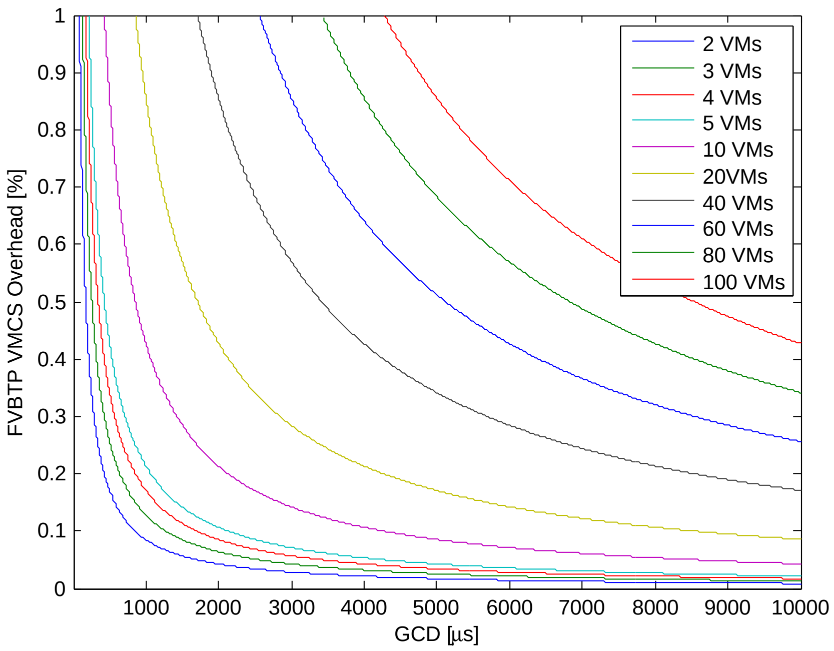


Figure 68: VMCS overhead for periods from 1μs up to 10ms on PowerPC405 @ 300 MHz

4.6.6 Case Study

To sum up the evaluation up to this point, FVBTP introduces significantly less runtime overhead than OSE, but suffers from a very bad GCD distribution (see 4.6.2), which is essential for the total number of VMCS needed. So one may expect that OSE outperforms FVBTP even with its higher runtime overhead due to its expensive scheduling decisions. Nevertheless, this section will show that FVBTP is applicable to realistic scenarios and it even outperforms OSE even when having small numbers of the GCD.

The presented scenarios in [Grö10] cover varying timing granularities of embedded systems ranging from microseconds to seconds. The computation times of the presented scenarios are adopted to be able to consolidate them on the evaluation platform presented in 4.6.3, because in some cases, a system with more than 300 MHz would have been necessary due to the calculated required speedup for FVBTP. In this case, it has been assumed that the 300MHz platform was able to execute those tasksets with the same speed as on the required platform.

Electrical Drive Engineering - Linear Motor Control

The first scenario presented in [Grö10] is taken from the CRC614[4] and covers the control of a linear motor developed at the University of Paderborn. It includes two parts. The first part is responsible for controlling the current and the second part is responsible for controlling the speed of the motor. The current control involves two tasks while the speed control only involves one task. The tasksets are listed in 14.

When taking a look at the final results for the depicted scenario shown in table 15, one can see that OSE introduces three times more VMM overhead when the number of VMCS are in the same magnitude. This is due to the VMCS of OSE being three times slower than the VMCS of FVBTP in the case of two virtual machines. FVBTP assigns 57.46% to the guests in case of EDF and 69.13% in case of RM. This is caused by the pessimistic approach of scaling based on the least upper bound for the RM taskset. Here, OSE has an advantage of 11.67% less guest load. Neverthe-

4 The Collaborative Research Centre 614 "Self-Optimizing Concepts and Structures in Mechanical Engineering"

Guest System	Task	Computation Time [ms] *	Period [ms]
(1) Current Control	Controller_{C_1}	0.25	3
	Controller_{C_2}	0.25	3
(2) Speed Control	Controller_S	0.25	42

* On a 1 GHz processor

Table 14: Scenario IV: Electrical Drive Engineering - Linear Motor Control. [Grö10]

less, this advantage is nearly equalized by the overhead caused by VMM.

Figure 69 shows the resulting schedules of OSE and FVBTP. In case of OSE the speed controller is executed by a single RTVM being colored green. The two current controllers are colored blue. At the very beginning the schedule of OSE shows its property of introducing more context switches than FVBTP due to its global EDF nature. The higher amount of VMCS in case of OSE is caused by the aspect that a task that terminated before its deadline returns the control to the VMM to trigger a new scheduling decision, because the head of the scheduling queue has changed. This is not necessary in case of FVBTP leading to a continuous execution within its activation slot even when a task terminated within this activation slot. Thus, an increasing number of tasks in a VMM causes OSE to perform more context switches at VMM level. Considering an additional controller task with the same period and the utilization equally distributed among these three controller tasks, OSE would introduce 15 additional context switches during the hyperperiod. Considering the worst case of the GCD being one millisecond, the VMM overhead of FVBTP would increase to 8.55% as the number of VMCS would increase to 84. Nevertheless, the total VMM overhead in this case is still less than the VMM overhead of OSE. Thus, the granularity of time used for the tasksets is an essential property influencing the performance of FVBTP. As a result, one can see that FVBTP has not been outperformed by OSE in this scenario, even if in the worst case scenario the GCD is one millisecond.

It has to be noted that the original periods of the current control tasks were given as $3.051667ms$ and $3.183013ms$, while the speed control task was given as $43.473157ms$ by the mechanical engineers. Now why is it possible to use $3ms$ and $42ms$? The

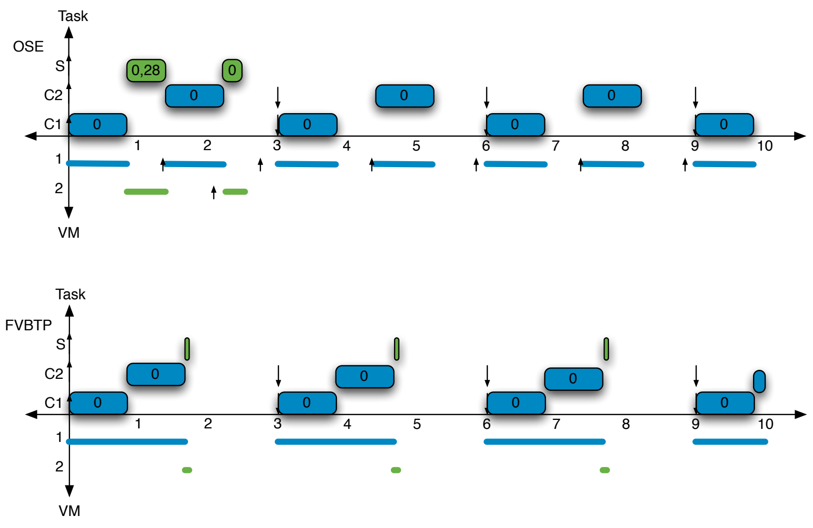


Figure 69: Schedules using OSE and FVBTP

		Illinois		**Paderborn**	
		Guest Scheduler		Guest Scheduler	
Scenario		EDF	RM	EDF	RM
(IV) Motor Control	VMM [%]	9.25	9.25	2.85	2.85
	Guests [%]	57.46	57.46	57.46	69.13
	Idle [%]	33.29	33.29	39.69	28,02
	U [%]	66.71	66.71	60.31	71,98
	VMCS	30	30	28	28

Table 15: Electrical Drive Engineering Evaluation. [Grö10]

property of periods being multiples of each other is quite common in control systems. This is due to the fact that the sampling and control frequency is selectable freely within certain limits dictated by the technical restrictions. Usually, the lower limit is fixed while the upper limit is not. The frequency of the actuators is then optimally been chosen as a multiple of the sampling frequencies [Lit04, Ric08]. Thus, it was possible to increase the rates to get a GCD of 3ms in this case for FVBTP. Nevertheless, FVBTP would have been even capable of handling a GCD of 1ms. So as a final result both approaches were able to consolidate two real-time systems being executed separately before on a 1GHz computer with FVBTP and OSE causing the same number of VMCS. The following scenarios will show a different behavior of OSE, as already mentioned above.

Guest System	**Task**	**Computation Time [ms] ***	**Period [ms]**
(1) Human-Machine Interface	GUI Control	10	100
	GUI Task 1	50	100
	GUI Task 2	5	100
(2) Corporate IT Interface	Industrial Ethernet	1	10
	Database Control	50	100
(3) CNC Control	Servo Control 1	1	10
	Servo Control 2	1	10
	Servo Control 3	1	10
	Sensor Task 1	1	10
	Sensor Task 2	1	10
	Sensor Task 3	1	10
	Emergency Stop Check	1	10

* On a 100 MHz processor

Table 16: Scenario I: Industrial - CNC Machine. [Grö10]

Industrial – CNC Machine

The second scenario presented in [Grö10] covers an industrial automation example. Therefore, three systems of a CNC[5] machine control are consolidated on a single virtualized system. One of the systems is responsible for handling the GUI[6]-based human machine interface. This GUI requires a latency of 100ms to be considered reacting instantaneously [Mil68, Nie93]. Another system is responsible for database access via an industrial ethernet connection to acquire the necessary information of the workpiece in production and another system is responsible for sensing and controlling the actuators of the CNC machine [WLM+10]. The tasksets can be found in table 16.

This scenario shows the impact of multiple tasks within one RTVM on the number of VMCS of OSE. The results are shown in table 17. In this scenario, OSE needs to perform 129 VMCS during the hyperperiod of 100ms, which leads to a total VMCS overhead of 24.58%. In contrast, only FVBTP causes an overhead of 1.29% for the GCD being 10ms. Even in the worst case of the GCD being 1ms, the total VMCS overhead with 12.82% would

5 Computer Numerical Control (CNC) refers to the automation of machine tools that are operated by programmed commands

6 Graphical User Interface

		Illinois		Paderborn	
		Guest Scheduler		Guest Scheduler	
Scenario		EDF	RM	EDF	RM
Industrial	VMM [%]	24.58	24.58	1.29	1.29
CNC	Guests [%]	65.00	65.00	65.00	83.93
Machine	Idle [%]	10.12	10.12	33.71	14.78
	U[%]	89.88	89.88	66.29	85, 22
	VMCS	129	129	30	30

Table 17: Industrial - CNC Machine Evaluation. [Grö10]

be still half of the overhead introduced by OSE. Unfortunately, this would cause a total utilization of 98.05% when using FVBTP with RM as guest schedulers due to the pessimistic activation slot allocation. Thus, the total utilization would be about 8% more than the utilization of OSE in that case.

Medical – X-ray Machine

The third scenario shown in [Grö10] covers the medical domain and includes the subsystems typical for medical applications. The first subsystem is the human-machine interface and the second subsystem is responsible for controlling of the X-ray unit. The human-machine interface has three tasks. The first task is to control the GUI. The second task is to perform some image processing on the gathered image, and the final task is to visualize the resulting image. The frame rate is about one image every half a second. The control subsystem needs to perform sensing and controlling the actuators to position the X-ray beam as required. The AEC[7] ensures the exposition limitation of an object to the X-ray beam when the human operator did not turn off the X-ray beam after a specific period of time [VS06]. The corresponding taskset of this scenario is shown in table 18.

In this scenario, the execution times and their periods of all tasks are quite long. Thus, OSE and FVBTP had no problem coping with this example. Table 19 shows the results for this scenario. One can see that there is still a significant difference in overhead between OSE and FVBTP, what is expected due to the effects described in the previous scenarios, but due to the long execution times and periods, this difference did not have a large impact on

7 Automatic Exposure Control

Guest System	**Task**	**Computation Time [ms] ***	**Period [ms]**
(1) Human-Machine Interface	GUI Control	10	100
	Image Processing	200	500
	Visualization	100	500
(2) X-Ray Control	AEC	50	1000
	Servo Control 1	20	100
	Servo Control 2	20	100
	Sensor Task 1	10	100
	Sensor Task 2	10	100

* On a 150 MHz processor

Table 18: Scenario II: Medical - X-ray Machine. [Grö10]

		Illinois Guest Scheduler		**Paderborn** Guest Scheduler	
Scenario		EDF	RM	EDF	RM
(II) Medical	VMM [%]	0.97	0.97	0.001	0.001
	Guests [%]	67.50	67.50	67.50	88.61
	Idle [%]	31.53	31.53	32.41	11.30
	U [%]	68.47	68.47	67.59	88.70
	VMCS	75	75	20	20

Table 19: Medical Evaluation. [Grö10]

the total overhead introduced by OSE. Thus, both approaches performed very good in this scenario.

Automotive - Airbag control and Driver Assistance

The fourth scenario presented in [Grö10] covers an automotive consolidation of an ACU[8], ABS[9], and ESC[10]. The tasksets of this scenario are listed in table 20. The Airbag control unit runs a detection task which monitors the results of the different sensor tasks and triggers the gas inflation in case of a detected crash. For each of the eight sensors[11] used in the airbag control, a corresponding sensor task is executed to handle the readout [Rei07]. The ABS works at a speed of 10Hz. The sensor task monitors the

8 Airbag Control Unit
9 Anti-lock Braking Systems
10 Electronic Stability Control
11 Accelerometer, Impact Sensor, Pressure Sensors, Wheel Speed Sensor and Gyroscopes

Guest System	Task	Computation Time [ms] *	Period [ms]
(1) Airbag Control Unit	Detection Task	1.5	15
	Sensor Task 1	0.3	15
	Sensor Task 2	0.3	15
	Sensor Task 3	0.3	15
	Sensor Task 4	0.3	15
	Sensor Task 5	0.3	15
	Sensor Task 6	0.3	15
	Sensor Task 7	0.6	15
	Sensor Task 8	0.6	15
(2) Anti-lock Braking System	Brake Pressure Control	20	100
	Sensor Task	10	50
	CAN Communication	1	5
(3) Electronic Stability Control	Detection Task	0.9	5
	Servo Control	0.3	5
	Sensor Task 1	0.3	5
	Sensor Task 2	0.3	5
	Sensor Task 3	0.3	5
	Sensor Task 4	0.3	5
	CAN Communication	1	5

* On a 100 MHz processor

Table 20: Scenario III: Automotive - Airbag Control and Driver Assistance. [Grö10]

rotational speed of the four wheels. This sensor input is used in the brake pressure control task to calculate the applied pressure for the target actuator. The ESC compares the driver steering commands based on a steering angle sensor with the current driving conditions. When necessary, the ESC intervenes keeping the car in a stable and safe driving state. The tasks necessary for the ESC are executed at a rate of 150Hz [Bos07]. To prevent contradictory commands of the ABS and the ESC both components are interconnected using a CAN[12] interface [Ise06]. The consolidation of the ABS and the ESC into a virtualized system eliminates the need of the physical interconnection of those systems using CAN and can be replaced by virtual network interfaces.

This scenario again shows the impact of multiple tasks in a RTVM, what leads to an overload condition for OSE. Due to the

12 Controller Area Network

		Illinois		**Paderborn**	
		Guest Scheduler		Guest Scheduler	
Scenario		EDF	RM	EDF	RM
(III) Automotive	VMM [%]	52.47	52.47	2.57	2.57
	Guests [%]	52.67	52.67	52.70	70.64
	Idle [%]	-	-	44.73	26.79
	U[%]	> 100	> 100	55.27	73.21
	VMCS	826	826	180	180

Table 21: Automotive Scenario Evaluation. [Grö10]

strong time-shared behavior of the tasksets within short periods, a lot of VMCS are necessary for OSE, what creates a required VMM utilzation of 52.47%. Here, FVBTP shows its strength by performing 4.5 times less VMCS than OSE by executing multiple tasks within of a RTVM in a single activation slot. In addition, the impact of the VMCS being around five times more expensive in the case of OSE causes the total VMM utilization difference of 50% compared to FVBTP. Considering again the worst case of a GCD being $1ms$, the VMCS overhead would increase to 12.82% with the scenario still being schedulable by FVBTP in case of EDF and RM.

4.6.7 Summary

The evaluation showed the expected performance advantage of FVBTP in all cases of pure EDF based RTVMs due to significant difference of the VMCS overhead of OSE. This overhead caused by the dynamic scheduling approach enabled FVBTP to outperform OSE even in the case of its worst case of the GCD being $1ms$. This effect is caused, by the fact that to the VMCS overhead uses only 4.28% in an interval of $1ms$ length. In contrast, OSE needs 12.95% in the same interval and in the case of only two RTVMs. With an increasing number of RTVMs the VMCS overhead of OSE quickly grows quadratically to 23.6% in case of four RTVMs, while the VMCS overhead for FVBTP is constant due to its static behavior. Thus, the dynamic behavior comes at a high price when dealing with short periods, which has been shown in the scenarios. Besides the pure values of the VMCS overhead, the number of performed VMCS has been evaluated. For FVBTP, a direct derivation of the number of VMCS was possible and a performance graph for the caused overhead for a given GCD was presented. In case of OSE, the number of VMCS is dependent on the executed tasks in the virtualized system. The scenario evaluation showed a significant higher amount of VMCS for OSE. The reason for this behavior is the preemptive behavior of OSE. OSE needs to perform a scheduling decision in the VMM upon every task finishing time or when a CUS with higher priority becomes active causing the active RTVM to be preempted.

When taking a look at pure RM guests, one notices that FVBTP has a higher guest utilization. This is due to the pessimistic scaling of the activation slots based on the least upper bound for the executed taskset. Due to this decision, it is always ensured that the taskset is schedulable in the RTVM. This pessimistic scaling can be improved by scaling relatively to higher utilizations than the least upper bound, but then a response time analysis (see 2.1.2) is necessary in advance to determine whether the taskset is still schedulable under this conditions or not.

Summing all up FVBTP performed very good in all scenarios demonstrating that the worst case GCD impact can be kept low when the time granularity is in the next magnitude to the VMCS overhead. In the special case of the evaluation platform, this was a time granularity of ms being the next magnitude to the VMCS overhead being in the lower μs magnitude. The impact of the dynamic behavior of OSE is as severe as expected in the case of multiple tasks per RTVM with short periods due to the preemptive behavior. This prevents OSE from being applicable to the automotive scenario. The possibility of adding dynamically RTVMs to OSE makes it attractive for dynamic environments, but it has to be noted that the required server parameter calculation proposed by Bini et. al. (see 4.2.1) does not come without consequences. Thus, the application is restricted to more powerful hardware architectures than the used PowerPC405FX.

5 SUMMARY

The characteristics of embedded systems has changed dramatically within the last two decades. Especially the growing complexity of embedded real-time systems and their demand for high-level functionality typically provided by GPOSes creates different problems that have to be faced when developing, assembling and deploying them. One of the main problems of building such complex systems is the integration of software components into a big integrated system as described by Broy in [Broo6]. The integration of such complex systems can easily lead to unmodelled and thus to unintentional feature interactions between the integrated components. In a distributed system where every task is placed on a single ECU, the unintentional interaction or a hardware failure are fairly the only issues endangering a components functionality. But this kind of feature distribution is currently something up for discussion, as the trend is to have less dedicated ECUs in favor of more centralized multi-functional hardware, due to the increasing need for high-level APIs like graphical user interfaces or multimedia functionality. The trend towards more centralized multi-functional hardware boosts the problem of unintentional interaction of software components as they share the processor, memory and I/O devices in this case. To isolate the different components spatial and temporal, the approach of virtualization is suitable. The main problem currently is the virtualization of multiple system either requiring hard, soft or non real-time using full virtualization instead of the widely applied paravirtualization approach. The paravirtualization approach requires access to the source code of the virtualized OS, which is in general not accessible due to licensing restrictions of the different vendors. Thus a VMM providing the possibility to use full virtualization and paravirtualization at the same time is required to offer the possibility to virtualized closed source guests as well as paravirtualizable guests like Linux, which offer high-level functionality for graphical user interfaces. The presented Proteus VMM was designed and implemented to face these problems and offers the possibility to execute full virtualized and paravirtualized guests at the same time while the provided ABI is configurable to keep the code overhead of the VMM as small as possible. To be able to handle hard real-time guests Proteus VMM provides an extensible scheduler interface and is strictly kept deterministic. Thus it is easily possible to determine the WCET of its different virtualization components. With the knowledge of WCET analysis approaches this allows, the approximation of the WCET of the virtualized execution of a guest application. The WCET of the virtualized guest is required for the schedulability analysis. Due to the integration of different systems into a single virtualized systems it is very useful to provide a methodology that automatically derives all execution parameters of the virtual real-time systems that executes given real-time systems as real-time virtual machines. Those parameters are especially the required performance and the schedule of the real-time virtual machines. The existing state of the art lacks the possibility of automatically deriving those parameters under the constraint of full virtualization. The presented approach called FVBTP[1] addresses this problems and allows for deriving automatically those parameters from a set of given systems. The derived schedule depends on the correlation of the deadlines being in the best case divided by large GCD to reduce the switching overhead of the VMM. This drawback can be smoothed by restricting the time granularity to the next magnitude. In addition, hard real-time systems used for controlling sensors and actuators very often have the property of their task periods being multiples of each other and the possibility of adjusting the periods in certain limits what mostly leads to good results for the GCD. The evaluation demonstrated the applicability of the presented approach and showed good results compared to the open system environment. The open system environment is based on paravirtualization and is a dynamic scheduling approach in contrast to the presented FVBTP which uses a static scheduling strategy and is based on full virtualization. The comparison to the open system environment was performed, as this

1 Full Virtualization by Temporal Partitioning

is the state of the scheduling approach and the only approach which is able to schedule multiple hard real-time systems.

To conclude, it has been shown that the defined goal of providing a configurable and deterministic VMM being able to handle paravirtualized and full virtualized guests was reached by implementing the presented VMM design into the VMM called Proteus. To ensure the schedulability under the constraint of full virtualization, the FVBTP approach was presented and evaluated. The results showed the applicability of the approach to representative scenarios. Further research may be focused on the support for multicore systems, as those systems seem to be the trend for future multi-functional hardware even in the field of embedded hard real-time systems. This especially includes the questions of how to distribute the real-time virtual machines in such a system. Another interesting area is the extension of the presented transformation of the given real-time system tasksets to derive the parameters of the virtualized real-time system. Currently, this transformation is kept quite simple as it is a proof of concept. The extension to transform the taskset between different ISAs and hardware architectures is quite interesting for making much more accurate approximations on the required execution platform for the host system.

A

EVALUATION

A.1 MEASUREMENTS

Tasks	Min[µs]	Max[µs]	Mean[µs]	Confidence Intervall$_{0.95}$
2	75.795	75.765	75.825	[75.765; 75.825]
3	111.448	111.421	111.476	[111.421; 111.476]
4	144.455	144.411	144.498	[144.411; 144.498]
5	172.491	172.475	172.507	[172.475; 172.507]
6	183.051	183.009	183.092	[183.009; 183.092]
7	214.671	214.615	214.727	[214.615; 214.727]
8	251.289	251.245	251.334	[251.245; 251334]
9	410.597	410.525	410.669	[410.525; 410.669]
10	424.146	424.068	424.225	[424.068; 424.225]
11	475.634	475.595	475.673	[475.595; 475.673]
12	520.839	520.773	520.905	[520.773; 520.905]
13	604.713	604.601	604.825	[604.601; 604.825]
14	617.245	617.226	617.264	[617.226; 617.264]
15	715.053	715.024	715.081	[715.024; 715.081]
16	728.816	728.726	728.906	[728.726; 728.906]
17	759.352	759.252	759.452	[759.252; 759.452]
18	792.982	792.888	793.076	[792.888; 793.076]
19	830.144	830.097	830.190	[830.097; 830.190]

Table 22: Performance of scheduler interface routine sched_init in case of FVBTP. [Grö10]

Tasks	Min[μs]	Max[μs]	Mean[μs]	Confidence Intervall$_{0.95}$
20	843.301	843.270	843.331	[843.270; 843.331]
30	1401.315	1401.276	1401.353	[1401.276; 1401.353]
40	1715.308	1715.241	1715.375	[1715.241; 1715.375]
50	2145.459	2145.327	2145.591	[2145.327; 2145.591]
60	2573.147	2573.017	2573.276	[2573.017; 2573.276]
70	3115.312	3115.196	3115.428	[3115.196; 3115.428]
80	3429.558	3429.386	3429.731	[3429.386; 3429.731]
90	3972.970	3972.840	3973.100	[3972.840; 3973.100]
100	4286.520	4286.500	4286.541	[4286.500; 4286.541]

Table 23: Performance of scheduler interface routine sched_init in case of FVBTP. [Grö10]

Tasks	Min[µs]	Max[µs]	Mean[µs]	Confidence Intervall$_{0.95}$
1	29.523	30.080	29.835	[29.812; 29.857]
2	54.680	55.240	55.010	[54.986; 55.033]
3	78.840	79.440	79.113	[79.089; 79.138]
4	104.600	105.123	104.841	[104.802;104.879]
5	130.083	130.882	130.604	[103.570; 130.638]
6	155.040	155.723	155.365	[155.332; 155.398]
7	177.160	177.803	177.443	[177.413; 177.472]
8	203.400	204.365	203.910	[203.855; 203.966]
9	229.363	230.405	230.020	[229.974; 230.066]
10	254.403	255.005	254.637	[254.601; 254.672]
11	275.603	276.363	276.054	[276.019; 276.088]
12	302.082	303.045	302.589	[302.516; 302.663]
13	329.402	330.485	329.912	[329.859; 329.965]
14	355.725	356.765	356.475	[356.409; 356.541]
15	374.202	374.885	374.498	[374.457; 374.539]
16	402.285	403.005	402.533	[402.501; 402.565]
17	431.405	432.165	431.806	[431.762; 431.850]
18	454.725	455.805	455.509	[455.456; 455.561]
19	472.645	473.325	472.944	[472.897; 472.992]
20	501.365	502.365	501.976	[501.914; 502.038]
30	753.648	753.970	753.787	[753.753; 753.821]
40	998.133	999.090	998.616	[998.512; 998.719]
50	1256.215	1257.135	1256.765	[1256.681; 1256.849]
60	1493.180	1494.497	1493.815	[1493.657; 1493.973]
70	1756.900	1757.500	1757.169	[1757.072; 1757.266]
80	1991.463	1991.985	1991.723	[1991.633; 1991.814]
90	2243.425	2247.825	2246.326	[2245.335; 2247.317]
100	2488.347	2488.710	2488.491	[2488.412; 2488.571]

Table 24: Performance of scheduler interface routine sched_init in case of OSE. [Grö10]

VMs	Min[µs]	Max[µs]	Mean[µs]	Confidence Intervall$_{0.95}$
2	85.882	86.882	86.792	[86.763; 86.820]
3	154.563	156.882	156.857	[156.812; 156.903]
4	199.322	202.245	202.213	[202.156; 202.270]
5	306.243	309.565	309.408	[309.342; 309.474]
6	365.965	369.885	369.613	[369.532; 369.694]
7	435.125	439.527	439.364	[439.277; 439.450]
8	498.005	503.408	503.333	[503.226; 503.440]
9	669.648	676.170	675.432	[675.290; 675.573]
10	749.050	756.570	755.932	[755.760; 756.104]
11	839.690	846.410	846.079	[845.945; 846.212]
12	921.970	930.693	930.107	[929.934; 930.281]
13	1028.172	1037.333	1036.800	[1036.610; 1036.990]
14	1118.573	1128.253	1127.983	[1127.797; 1128.169]
15	1219.255	1229.295	1228.616	[1228.411; 1228.821]
16	1313.375	1324.098	1323.314	[1323.097; 1323.531]
17	1603.898	1615.780	1615.260	[1615.027; 1615.493]
18	1716.940	1730.860	1729.660	[1729.403; 1729.918]
19	1841.223	1853.863	1853.360	[1853.117; 1853.603]
16	1958.303	1972.385	1972.223	[1971.948; 1972.499]
30	3389.000	3408.600	3407.685	[3407.308; 3408.062]
40	5520.302	5545.505	5545.026	[5544.536; 5545.517]
50	7722.488	7755.010	7754.684	[7754.050; 7755.318]
60	10181.683	10221.400	10220.492	[10219.721; 10221.264]
70	13690.518	13739.800	13737.246	[13736.323; 13738.168]
80	16896.037	16947.315	16945.127	[16944.159; 16946.094]
90	20485.518	20543.520	20542.339	[20541.213; 20543.465]
100	24481.280	24485.522	24483.402	[24482.890; 24483.913]

Table 25: Performance of scheduler interface routine sched_getNextVMIndex in case of OSE. [Grö10]

Min[µs]	Max[µs]	Mean[µs]	Confidence Intervall$_{0.95}$
2.918	3.440	3.054	[3.035 ; 3.073]

Table 26: Performance of scheduler interface routine sched_getNextVMIndex in case of FVBTP. [Grö10]

A.2 SCENARIO I: ELECTRICAL DRIVE ENGINEERING – LINEAR MOTOR CONTROL

Both subsystems are executed on a dSPACE DS1005 PPC board[1]. The processor of this platform is a PowerPC 750GX, with a processor speed of 1 GHz. If one executes these controller tasks on a PowerPC with 300 MHz, the execution times have to be multiplied by 1000 MHz/300 MHz $\approx$ 3.333.

$RTVM_1(\Gamma) = \{\ (3,\ 0.833),\ (3,\ 0.833)\ \}$
$RTVM_2(\Gamma) = \{\ (42,\ 0.833)\ \}$

$U(RTVM_1(\Gamma)) \approx 0.556$
$U(RTVM_2(\Gamma)) \approx 0.020$

(a) EDF

Activation Slots:

$\Pi_1 = (\ (0,\ 1.668),\ 3\ )$

$\Pi_2 = (\ (1.668,\ 1.724),\ 3\ )$

(b) RM

$U_{lub}(\Gamma_1) \approx 0.828$

$U_{lub}(\Gamma_2) = 1$

Activation Slots:

$\Pi_1 = (\ (0,\ 2.014),\ 3\ )$

$\Pi_2 = (\ (2.014,\ 2.074),\ 3\ )$

1 http://www.dspace.com/

A.3 SCENARIO II: INDUSTRIAL – CNC MACHINE

All three subsystems are assumed to be executed on a 100 MHz processor, and are now consolidated on a PowerPC with 300 MHz.

$RTVM_1(\Gamma) = \{\ (100,\ 3.333),\ (100,\ 16.667),\ (100,\ 1.667)\ \}$

$RTVM_2(\Gamma) = \{\ (10,\ 0.333),\ (100,\ 16.667)\ \}$

$RTVM_3(\Gamma) = \{\ (10,\ 0.333),\ (10,\ 0.333),\ (10,\ 0.333),\ (10,\ 0.333),$
$(10,\ 0.333),\ (10,\ 0.333),\ (10,\ 0.333)\ \}$

$U(RTVM_1) \approx 0.217$
$U(RTVM_2) \approx 0.2$
$U(RTVM_3) \approx 0.233$

(a) EDF

Activation Slots:

$\Pi_1 = (\ (0,\ 2.170),\ 10\ )$

$\Pi_2 = (\ (2.170,\ 3.870),\ 10\ )$

$\Pi_3 = (\ (3.870,\ 6.200),\ 10\ )$

(b) RM

$U_{lub}(RTVM_1(\Gamma)) \approx 0.780$

$U_{lub}(RTVM_2(\Gamma)) \approx 0.828$

$U_{lub}(RTVM_3(\Gamma)) \approx 0.729$

$\Pi_1 = (\ (0,\ 2.782),\ 10\ )$

$\Pi_2 = (\ (2.782,\ 5.197),\ 10\ )$

$\Pi_3 = (\ (5.197,\ 8.393),\ 10\ )$

Utilization G = 83.93 %

A.4 SCENARIO III: MEDICAL – X-RAY MACHINE

Both subsystems are assumed to be executed on a 150 MHz processor, and are now consolidated on a PowerPC with 300 MHz.

$RTVM_1(\Gamma) = \{ (100,\ 5.000),\ (500,\ 100.000),\ (500,\ 50.000) \}$

$RTVM_2(\Gamma) = \{ (1000,\ 25.000),\ (100,\ 10.000),\ (100,\ 10.000),$
$(100,\ 5.000),\ (100,\ 5.000) \}$

$U(RTVM_1(\Gamma)) \approx 0.350$
$U(RTVM_2(\Gamma)) \approx 0.275$

(a) EDF

Activation Slots:

$\Pi_1 = (\ (0,\ 35.000),\ 100\ )$

$\Pi_2 = (\ (35.000,\ 62.500),\ 100\ )$

(b) RM

$U_{lub}(RTVM_1(\Gamma)) \approx 0.780$

$U_{lub}(RTVM_2(\Gamma)) \approx 0.743$

Activation Slots:

$\Pi_1 = (\ (0,\ 44.872),\ 100\ )$

$\Pi_2 = (\ (44.872,\ 81.833),\ 100\ )$

A.5 SCENARIO IV: AUTOMOTIVE – AIRBAG CONTROL AND DRIVER ASSISTANCE

All three subsystems are assumed to be executed on a 100 MHz processor, and are now consolidated on a PowerPC with 300 MHz.

$RTVM_1(\Gamma) = \{ (15, 0.500), (15, 0.100), (15, 0.100), (15, 0.100), (15, 0.100), (15, 0.100), (15, 0.100), (15, 0.200), (15, 0.200) \}$

$RTVM_2(\Gamma) = \{ (100, 6.667), (50, 3.333), (5, 0.333) \}$

$RTVM_3(\Gamma) = \{ (5, 0.300), (5, 0.100), (5, 0.100), (5, 0.100), (5, 0.100), (5, 0.100), (5, 0.333) \}$

$U(RTVM_1(\Gamma)) \approx 0.100$
$U(RTVM_2(\Gamma)) \approx 0.200$
$U(RTVM_3(\Gamma)) \approx 0.227$

(a) EDF

Activation Slots:

$\Pi_1 = ((0, 0.500), 5)$

$\Pi_2 = ((0.500, 1.500), 5)$

$\Pi_3 = ((1.500, 2.635), 5)$

(b) RM

$U_{lub}(RTVM_1) \approx 0.721$

$U_{lub}(RTVM_2) \approx 0.780$

$U_{lub}(RTVM_3) \approx 0.729$

Activation Slots:

$\Pi_1 = ((0, 0.693), 5)$

$\Pi_2 = ((0.693, 1.975), 5)$

$\Pi_3 = ((1.975, 3.532), 5)$

BIBLIOGRAPHY

[Atm08] Atmel: *Atmega8: 8-bit with 8k bytes in-system programmable flash.*, Juli 2008. http://www.atmel.com/dyn/resources/prod_documents/doc2486.pdf.

[Aud91] Audsley, N: *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times*. Real-Time Systems, Jan 1991.

[BA00] Buttazzo, G and L Abeni: *Adaptive rate control through elastic scheduling*. Decision and Control, 2000. Proceedings of the 39th IEEE Conference on, 5:4883 – 4888 vol.5, Jan 2000.

[BA01] Bennett, M. D. and Neil C. Audsley: *Predictable and efficient virtual addressing for safety-critical real-time systems*. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 183–190. IEEE Computer Society, 2001.

[Bak05] Baker, T: *An analysis of EDF schedulability on a multiprocessor*. IEEE Transactions on Parallel and Distributed Systems, Jan 2005. http://oz.nthu.edu.tw/~d918323/real-time/01458691.pdf.

[Bal09] Baldin, Daniel: *Entwurf und Implementierung einer komponentenbasierten Virtualisierungsplattform für selbstoptimierende eingebettete mechatronische Systeme*. Diplomarbeit, Universität Paderborn, March 2009.

[BB01] Bini, E. and G. Buttazzo: *A hyperbolic bound for the rate monotonic algorithm*. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 59 –66, 2001.

[BB08] Baruah, S and T Baker: *Schedulability analysis of global edf*. Real-Time Systems, Jan 2008.

[BBB09] Bini, E, M Bertogna, and S Baruah: *Virtual multiprocessor platforms: Specification and use*. Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE, pages 437 – 446, Dec 2009. http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=5368130&isnumber=5368116&punumber=5368115&k2dockey=5368130@ieeecnfs.

[BDF$^+$03] Barham, Paul, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield: *Xen and the art of virtualization*. SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, Dec 2003. http://portal.acm.org/citation.cfm?id=945445.945462.

[Bel73] Bell, J: *Threaded code*. Communications, Jan 1973. http://portal.acm.org/citation.cfm?doid=362248.362270.

[BLA98] Buttazzo, G, G Lipari, and L Abeni: *Elastic task model for adaptive rate control*. Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE, pages 286 – 295, Dec 1998.

[BLCA02] Buttazzo, G, G Lipari, M Caccamo, and L Abeni: *Elastic scheduling for flexible workload management*. Computers, IEEE Transactions on, 51(3):289 – 302, Mar 2002.

[Bos07] Bosch, Robert: *Autoelektrik Autoelektronik*. Vieweg, 2007.

[Bra70] Bradley, G: *Algorithm and bound for the greatest common divisor of n integers*. Communications of the ACM, Jan 1970. http://portal.acm.org/citation.cfm?id=362694.

[Bro02] Broad, William J.: *For parts, NASA boldly goes on ebay*, May 12 2002. http://www.nytimes.com/2002/05/12/us/for-parts-nasa-boldly-goes-on-ebay.html(access,09/28/2010).

[Bro06] Broy, M: *Challenges in automotive software engineering*. Proceedings of the 28th international conference on Software engineering (ICSE '06), Jan 2006. http://portal.acm.org/citation.cfm?id=1134292.

[Bry09] Brygier, Jacques: *Extending military software life expectancy through safe and secure virtualization*. Military Embedded Systems, June, 2009.

[But04] Buttazzo, Giorgio C.: *Hard Real-Time Computing Systems*. Springer, 2004.

[But05] Buttazzo, G: *Rate monotonic vs. EDF: Judgment day*. Real-Time Systems, Jan 2005. http://www.springerlink.com/index/Q210434KLR294437.pdf.

[But06] Buttazzo, Giorgio: *Research trends in real-time com-*

puting for embedded systems. SIGBED Review, 3(3), Jul 2006. http://portal.acm.org/citation.cfm?id=1164050.1164052.

[BVZB05] Berndl, M, B Vitale, M Zaleski, and A Brown: *Context threading: a flexible and efficient dispatch technique for virtual machine interpreters*. Code Generation and Optimization, 2005. CGO 2005. International Symposium on, pages 15 – 26, Feb 2005. http://ieeexplore.ieee.org/search/srchabstract.jsp?arnumber=1402073&isnumber=30441&punumber=9631&k2dockey=1402073@ieeecnfs.

[BYMK...06] Ben-Yehuda, M, J Mason, O Krieger, and J Xenidis ...: *Utilizing IOMMUs for virtualization in Linux and Xen*. Proceedings of the Ottawa Linux Symposium (OLS '06), Jan 2006. http://landley.net/kdocs/ols/2006/ols2006v1-pages-71-86.pdf.

[CEG07] Casey, K, M Ertl, and D Gregg: *Optimizing indirect branch prediction accuracy in virtual machine interpreters*. portal.acm.org, Jan 2007. http://portal.acm.org/citation.cfm?id=1286821.1286828.

[CGV07] Cherkasova, Ludmila, Diwaker Gupta, and Amin Vahdat: *Comparison of the three CPU schedulers in xen*. SIGMETRICS Performance Evaluation Review, 35(2), Sep 2007. http://portal.acm.org/citation.cfm?id=1330555.1330556.

[CHL06] Chantem, T, Xiaobo Sharon Hu, and M Lemmon: *Generalized elastic scheduling*. Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International, pages 236 – 245, Dec 2006.

[Ciu08] Ciufo, Chris A.: *Virtualization yields hardware optimization and new embedded architectures*. Military Embedded Systems, July, 2008.

[CK97] Channon, D. and D. Koch: *Performance analysis of reconfigurable partitioned tlbs*. In *HICSS 97: Proceedings of the 30th Hawaii International Conference on System Sciences*, page 168, Washington, DC, USA, 1997. IEEE Computer Society.

[CM96] Cifuentes, C and V Malhotra: *Binary translation: static, dynamic, retargetable?* Software Maintenance 1996, Proceedings., International Conference on, pages 340 – 349, Oct 1996.

[CN01] Chen, Peter M. and Brian D. Noble: *When Virtual Is*

Better Than Real. 2001.

[DCSH05] D. C. Snowdon, S. Ruocco and G. Heiser: *Power management and dynamic voltage scaling: Myths and facts*. In *Proceedings of the 2005 Workshop on Power Aware Real-time Computing*, 2005.

[DCSH07] D. C. Snowdon, S. M. Petters and G. Heiser: *Accurate on-line prediction of processor and memory energy usage under voltage scaling:*. In *Proceedings of the 7th International Conference on Embedded Software*, 2007.

[Dew75] Dewar, Robert: *Indirect threaded code*. Communications of the ACM, 18(6), Jun 1975. http://portal.acm.org/citation.cfm?id=360825.360849.

[Dit98a] Ditze, Carsten: *A customizable library to support software synthesis for embedded applications and microkernel systems*. EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications, Sep 1998. http://portal.acm.org/citation.cfm?id=319195.319209.

[Dit98b] Ditze, Carsten: *A step towards operating system synthesis*. In *In Proc. of the 5th Annual Australasian Conf. on Parallel And Real-Time Systems (PART). IFIP, IEEE*, 1998.

[DK89] Demers, A and S Keshav: *Analysis and simulation of a fair queueing algorithm*. SIGCOMM 1989 Symposium proceedings on Communications architectures and protocols, Jan 1989.

[DL97] Deng, Z and J Liu: *Scheduling real-time applications in an open environment*. Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE, pages 308 – 319, Nov 1997.

[DLS97] Deng, Z, J Liu, and J Sun: *A scheme for scheduling hard real-time applications in open system environment*. Real-Time Systems, 1997. Proceedings., Ninth Euromicro Workshop on, pages 191 – 199, May 1997.

[EG01] Ertl, M and D Gregg: *The behavior of efficient virtual machine interpreters on modern architectures*. Lecture Notes in Computer Science, Jan 2001. http://www.springerlink.com/index/1EUU8KJJ7TXRR1WE.pdf.

[EGKP02] Ertl, M, D Gregg, A Krall, and B Paysan: *Vmgen-a generator of efficient virtual machine interpreters*. Software: Practice and Experience, Jan 2002. http://doi.

wiley.com/10.1002/spe.434.abs.

[EPF06] EPFL, S: *Optimizing network virtualization in Xen*. USENIX Annual Technical Conference, 2006. http://www.usenix.org/events/usenix06/tech/menon/menon.pdf.

[Ert99] Ertl, M: *Optimal code selection in DAGs*. POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Jan 1999. http://portal.acm.org/citation.cfm?id=292540.292562.

[EW01] Ertl, M and T Wien: *Threaded code variations and optimizations*. EuroForth 2001 Conference Proceedings, Jan 2001.

[FH04] Ferdinand, Christian and Reinhold Heckmann: *aiT: Worst-Case Execution Time prediction by static program analysis*. In Jacquart, Renè (editor): *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 377–383. Springer Boston, 2004.

[FM02] Feng, Xiang and A Mok: *A model of hierarchical real-time virtual resources*. Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE, pages 26 – 35, Nov 2002.

[GL94] Greene, Robert and George Lownes: *Embedded CPU target migration, doing more with less*. Proceedings of the conference on TRI-Ada '94, 1994.

[Gre10] Greenhills: *Real-Time Operating Systems (RTOS), Embedded Development Tools, Optimizing Compilers, IDE tools, Debuggers - Green Hills Software*. Website, June 2010. http://www.ghs.com/.

[Grö10] Grösbrink, Stefan: *Comparison of alternative hierarchichal scheduling techniques for the virtualization of embedded real-time systems*. Diplomarbeit, Universität Paderborn, Dec 2010.

[Hei07] Heiser, Gernot: *Virtualization for embedded systems*. Technical report, Open Kernel Labs, Apr 17 2007.

[Hei08a] Heinz, Thomas: *Preserving temporal behaviour of legacy real-time software across static binary translation*. IIES '08: Proceedings of the 1st workshop on Isolation and integration in embedded systems, Apr 2008. http://portal.acm.org/citation.cfm?id=

1435458.1435459.

[Heio8b] Heiser, Gernot: *The role of virtualization in embedded systems*. IIES '08: Proceedings of the 1st workshop on Isolation and integration in embedded systems, Apr 2008. http://portal.acm.org/citation.cfm?id=1435458.1435461.

[HM80] Horspool, R and N Marovac: *An approach to the problem of detranslation of computer programs*. The Computer Journal, Jan 1980. http://comjnl.oxfordjournals.org/cgi/content/abstract/23/3/223.

[IBM05] IBM: *PPC405Fx Embedded Processor Core Users Manual*, January 2005.

[Inc08] Inc., Wind River: *Arinc 653 - an avionics standard for safe, partitioned systems*. In *IEEE-CS Seminar*, June 4 2008.

[Int07] Intel: *Virtualization brings real benefits*. Whitepaper - www.intel.com, July 2007.

[Int09] Intel: *Computing technologies for medical equipment*. Medical Equipment Whitepaper - www.intel.com, 2009.

[IO07] Ito, M and S Oikawa: *Mesovirtualization: Lightweight virtualization technique for embedded systems*. Lecture Notes in Computer Science, Jan 2007. http://www.springerlink.com/index/y7228126v5933720.pdf.

[IO08] Ito, M and S Oikawa: *Improving real-time performance of a virtual machine monitor based system*. Lecture Notes in Computer Science, Jan 2008. http://www.springerlink.com/index/10xr208023j16574.pdf.

[Ise06] Isermann, Rolf: *Fahrdynamik-Regelung: Modellbildung, Fahrerassistenzsysteme, Mechatronik*. Vieweg+Teuber, 2006.

[JM98a] Jacob, B. L. and T.N. Mudge: *Virtual memory in contemporary microprocessors*. IEEE Micro, 18(4):33–43, 1998.

[JM98b] Jacob, Bruce L. and Trevor N. Mudge: *A look at several memory management units, TLB-Refill mechanisms, and page table organizations*. In *Proceedings of the Eighth International Conference on Architectural Supportfor Programming Languages and Operating Systems*, pages 295–306, 1998.

[JSMA98] Jeffay, K, F Donelson Smith, A Moorthy, and J Anderson: *Proportional share scheduling of operating system services for real-time applications*. Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE, pages 480 – 491, Nov 1998.

[Kai08a] Kaiser, Robert: *Alternatives for scheduling virtual machines in real-time embedded systems*. IIES '08: Proceedings of the 1st workshop on Isolation and integration in embedded systems, Apr 2008. http://portal.acm.org/citation.cfm?id=1435458.1435460.

[Kai08b] Kaiser, Robert: *Applying virtualization to real-time embedded systems*. 1. GI/ITG KuVS Fachgespr"ach Virtualisierung, 2008.

[KB09] Kerstan, Timo and Daniel Baldin: *ORCOS*. Website, 2009. https://orcos.cs.uni-paderborn.de/orcos/.

[KL99] Kuo, Tei Wei and Ching Hui Li: *A fixed-priority-driven open environment for real-time applications*. Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE, pages 256 – 267, 1999.

[LB00] Lipari, G and S Baruah: *Efficient scheduling of real-time multi-task applications in dynamic systems*. Real-Time Technology and Applications Symposium, 2000. RTAS 2000. Proceedings. Sixth IEEE, pages 166 – 175, Jan 2000.

[LB05] Lipari, G and E Bini: *A methodology for designing hierarchical scheduling systems*. Journal of Embedded Computing, Jan 2005. http://iospress.metapress.com/index/A60KEB3BB9C9CPTN.pdf.

[LCB00] Lipari, G, J Carpenter, and S Baruah: *A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments*. Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE, pages 217 – 226, 2000.

[LCH06] Laune C. Harris, Barton P. Miller: *Practical Analysis of Stripped Binary Code*. Technical report, Computer Sciences Department, University of Wisconsin, 2006.

[Lit04] Litz, Lothar: *Grundlagen der Automatisierungstechnik: Regelungssysteme - Hybride Systeme*, Seiten 145–146. Oldenbourg, 2004.

[LL73] Liu, C and J Layland: *Scheduling algorithms for multiprogramming in a hard-real-time environment*. Journal of the Association for computing Machinery, Jan 1973. http://portal.acm.org/citation.cfm?doid=321738.321743.

[LSD89] Lehoczky, J, L Sha, and Y Ding: *The rate monotonic scheduling algorithm: exact characterization and average case behavior*. Real Time Systems Symposium, 1989. Proceedings., The 10th IEEE, pages 166 – 171, 1989.

[LUC+06] LeVasseur, Joshua, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser: *Pre-Virtualization: soft layering for virtual machines*. Technical report 2006-15, Fakultät für Informatik, Universität Karlsruhe (TH), July 2006.

[Lyn09] LynuxWorks: *Embedded Hypervisor and Separation Kernel for Operating-system Virtualization: LynxSecure*. Website, February 2009.

[Mai09] Main, Chris: *Allowing for GPOS and RTOS: The unique virtualization needs of mission-critical embedded systems*. Military Embedded Systems, September, 2009.

[MFC01] Mok, A, X Feng, and D Chen: *Resource partition for real-time systems*. Proc. of IEEE Real-Time Technology and Applications ..., Jan 2001. http://ieeexplore.ieee.org/iel5/7401/20109/00929867.pdf?arnumber=929867.

[Mil68] Miller, Robert: *Response time in man-computer conversational transactions*. Proc. AFIPS Fall Joint Computer Conference, 33:267–277, Jan 1968. http://portal.acm.org/citation.cfm?id=1476589.1476628.

[MMH08] Murray, Derek, Grzegorz Milos, and Steven Hand: *Improving Xen security through disaggregation*. VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, Mar 2008. http://portal.acm.org/citation.cfm?id=1346256.1346278.

[Mod09] *ModelSim - a comprehensive simulation and debug enivronment for complex ASIC and FPGA designs*, 2009. http://www.model.com/.

[Nea06] Neumann, D. and D. Kulkari et. al.: *Intel virtualization technology in embedded and communication infrastructure applications*. Intel Technology Journal, 10(3), 2006.

[Nie93] Nielsen, J: *The usability engineering life cycle*. Computer, 25(3):12–22, 1993.

[Obj08] Objective Interface Systems: *MILS Technical Primer*. Website, 2008. http://www.ois.com/Products/MILS-Technical-Primer.html.

[Oer09] Oertel, Markus: *Entwurf und prototypische Implementierung eines echtzeitfähigen ATmega8 Emulators*. Diplomarbeit, University of Paderborn, 2009.

[OIN06] Oikawa, Shuichi, Megumi Ito, and Tatsuo Nakajima: *Linux/RTOS hybrid operating environment on gandalf virtual machine monitor*. 4096, Jan 2006. http://www.springerlink.com/content/j580077065882221/.

[PB00] Puschner, Peter and Alan Burns: *A Review of Worst-Case Execution-Time Analysis*. Journal of Real-Time Systems, 18(2/3):115–128, May 2000.

[PG74] Popek, G and R Goldberg: *Formal requirements for virtualizable third generation architectures*. Communications of the ACM, Jan 1974. http://dforeman.cs.binghamton.edu/~foreman/552pages/Readings/popek74formal.pdf.

[Pit87] Pittman, T: *Two-level hybrid interpreter/native code execution for combined space-time program efficiency*. ACM SIGPLAN Notices, Jan 1987. http://portal.acm.org/citation.cfm?id=960114.29666.

[PLW+06] Peng, Jinzhan, Guei Yuan Lueh, Gansha Wu, Xiaogang Gou, and Ryan Rakvic: *A comprehensive study of hardware/software approaches to improve TLB performance for java applications on embedded systems*. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 102–111, New York, NY, USA, 2006. ACM Press.

[PR98] Piumarta, Ian and Fabio Riccardi: *Optimizing direct threaded code by selective inlining*. PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, May 1998. http://portal.acm.org/citation.cfm?id=277650.277743.

[Reg01] Regehr, John: *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. PhD thesis, University of Virginia, 2001.

[Rei07] Reif, Konrad: *Automobilelektronik*. Vieweg, 2007.

[Ric08] Richter, K: *Echtzeitfähigkeit im verteilten Regler-Entwurf*. Elektronik Automotive, (9):42–45, 2008.

[RRW+03] Regehr, J, A Reid, K Webb, M Parker, and J Lepreau: *Evolving real-time systems using hierarchical scheduling and concurrency analysis*. Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE, pages 25 – 36, Jan 2003.

[RS07] Raj, H and K Schwan: *High performance and scalable I/O virtualization via self-virtualized devices*. Proceedings of the 16th international symposium on High performance distributed computing (HPDC '07), Jan 2007. http://portal.acm.org/citation.cfm?id=1272390.

[SAWJ+96] Stoica, I, H Abdel-Wahab, K Jeffay, S Baruah, J Gehrke, and C Plaxton: *A proportional share resource allocation algorithm for real-time, time-shared systems*. Real-Time Systems Symposium, 1996., 17th IEEE, pages 288 – 299, Nov 1996.

[SB09] Sabeghi, M and K Bertels: *Toward a runtime system for reconfigurable computers: A virtualization approach*. Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09., pages 1576 – 1579, Apr 2009.

[SBR05] Schnerr, Jurgen, Oliver Bringmann, and Wolfgang Rosenstiel: *Cycle accurate binary translation for simulation acceleration in rapid prototyping of SoCs*. DATE '05: Proceedings of the conference on Design, Automation and Test in Europe, 2, Mar 2005. http://portal.acm.org/citation.cfm?id=1048925.1049217.

[SCEG08] Shi, Y, K Casey, M Ertl, and D Gregg: *Virtual machine showdown: stack versus registers*. portal.acm.org, Jan 2008. http://portal.acm.org/citation.cfm?id=1328197.

[SCK+93] Sites, Richard, Anton Chernoff, Matthew Kirk, Maurice Marks, and Scott Robinson: *Binary translation*. Communications of the ACM, 36(2), Feb 1993. http://portal.acm.org/citation.cfm?id=151220.151227.

[SD98] Saumya Debray, Robert Muth, Matthew Weippert: *Alias Analysis of Executable Code*. Technical report, Department of Computer Science, University of Arizona, 1998.

[Sed09] Sedjelmaci, S: *The mixed binary euclid algorithm*. Electronic Notes in Discrete Mathematics, Jan 2009. http://linkinghub.elsevier.com/retrieve/pii/S1571065309001826.

[SH02] Shapiro, J and N Hardy: *EROS: A principle-driven operating system from the ground up*. IEEE software, Jan 2002.

[Sha49] Shannon, C.E.: *Communication in the presence of noise*. Proceedings of the IRE, 37(1):10 – 21, jan 1949, ISSN 0096-8390.

[SL03] Shin, I and I Lee: *Periodic resource model for compositional real-time guarantees*. Proceedings of the 24th IEEE International Real-Time . . . , Jan 2003.

[SL04] Shin, I and I Lee: *Compositional real-time scheduling framework*. Proc. of IEEE Real-Time Systems Symposium, Jan 2004.

[SL08] Shin, I and I Lee: *Compositional real-time scheduling framework with periodic model*. ACM Transactions on Embedded Computing Systems . . . , Jan 2008. http://portal.acm.org/citation.cfm?id=1347383.

[SN05a] Smith, J and R Nair: *The architecture of virtual machines*. Computer, Jan 2005.

[SN05b] Smith, James E. and Ravi Nair: *Virtual Machines - Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[Sou09] *Winavr*, 2009. http://sourceforge.net/projects/winavr/.

[SS97] Suzuki, S. and K. Shin: *On memory protection in real-time os for small embedded systems*. In *Fourth International Workshop on Real-Time Computing Systems and Applications (RTCSA97)*, page 51, 1997.

[SSF$^+$04] Seyer, Reinhard, Christian Siemers, Rainer Falsett, Klaus H. Ecker, and Harald Richter: *Robust partitioning for reliable real-time systems*. In *Workshop on Parallel & Distributed Real-Time Systems – WPDRTS, 18th International Parallel and Distributed Processing Symposium (18th IPPDS'04)*. IEEE Computer Society, 2004.

[Sys10] Sysgo AG: *PikeOS manual*, January 2010.

[Tan07] Tanenbaum, Andrew S.: *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007, ISBN 9780136006633.

[TH94] Talluri, Madhusudhan and Mark D. Hill: *Surpassing the TLB performance of superpages with less operating system support*. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 171–182, 1994.

[Tij80] Tijdeman, R.: *The chairman assignment problem*. Discrete Mathematics, 32(3):323 – 330, 1980, ISSN 0012-365X. http://www.sciencedirect.com/science/article/B6V00-48PVXSR-C/2/9d4146c9c8b36a222cdd992dac3af767.

[TKHP92] Talluri, M., S. Kong, M. Hill, and D. Patterson: *Tradeoffs in supporting two page sizes*. In *Proceedings of the 19th ISCA*, pages 415–424, 1992.

[UNRS05] Uhlig, R, G Neiger, D Rodgers, and A Santoni: *Intel virtualization technology*. Computer, Jan 2005.

[Vir09] VirtualLogix: *VirtualLogix - Real-time Virtualization for Connected Devices :: Products - VLX for Embedded Systems:*. Website, February 2009. http://www.virtuallogix.com/.

[VmW09] VmWare: *TRANGO Virtual Prozessors: Scalable security for embedded devices*. Website, February 2009. http://www.trango-vp.com/.

[VS06] Varjonen, Mari and Pekka Strömmer: *Anatomically adaptable automatic exposure control (AEC) for amorphous selenium (a-Se) full field digital mammography (FFDM) system*. In Flynn, Michael J. and Jiang Hsieh (editors): *Medical Imaging 2006: Physics of Medical Imaging*, 2006.

[Wal95] Waldspurger, C: *Stride scheduling: Deterministic proportional-share resource management*. Technical report, 1995.

[WEE$^+$08] Wilhelm, Reinhard, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström: *The worst-case execution-time problem—overview of methods and survey of tools*. Trans. on Embedded Computing Sys., 7(3):1–53, 2008, ISSN 1539-9087. http://dx.doi.org/10.1145/1347375.1347389.

[Win10] *Windriver hypervisor*, 2010. http://www.windriver.com/products/hypervisor/.

[WLM$^+$10] Wan, J. F., D. Li, Y. Ming, F. Ye, Y. Quin Tu, and C. Hua Zhang: *A performance analysis model for real-time ethernet-based CNC machines*. Journal of Central South University of Technology, 2010.

[WW94] Waldspurger, Carl A. and William E. Weihl: *Lottery scheduling: flexible proportional-share resource management*. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association. http://portal.acm.org/citation.cfm?id=1267638.1267639.

[Xil09] Xilinx: *PowerPC 405 Processor Block Reference Guide*, 2009.

[YLLT10] Yang, L., H. Lin, J. Li, and Y. Tao: *The architecture and real-time communication of CNC systems based on switched ethernet*. 2nd International Conference on Computer Engineering based on switched ethernet, pages V1–169–V1–173, Dec 2010.

[ZP05] Zhou, Xiangrong and Peter Petrov: *Arithmetic-based address translation for energy-efficient virtual memory support in low-power, real-time embedded systems*. In *Proceedings of the 18th Annual Symposium on Integrated Circuits and Systems Design, SBCCI 2005, Florianolopolis, Brazil, September 4-7, 2005*, pages 86–91. ACM, 2005.