Adaptive Virtual Machine Scheduling and Migration for Embedded Real-Time Systems

Stefan Groesbrink, M.Sc.

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
doctor rerum naturalium (Dr. rer. nat.)

2015

Abstract

Integrated architectures consolidate multiple functions on a shared electronic control unit. They are well suited for embedded real-time systems that have to implement complex functionality under tight resource constraints. Multicore processors have the potential to provide the required computational capacity with reduced size, weight, and power consumption. The major challenges for integrated architectures are robust encapsulation (to prevent that the integrated systems corrupt each other) and resource management (to ensure that each system receives sufficient resources). Hypervisor-based virtualization is a promising integration architecture for complex embedded systems. It refers to the division of the hardware resources into multiple isolated execution environments (virtual machines), each hosting a software system of operating system and application tasks.

This thesis addresses the hypervisor's management of the resource computation time, which has to enable multiple real-time systems to share a multicore processor with all of them completing their computations as demanded. State of the art approaches realize the sharing of the processor by assigning exclusive processor cores or fixed processor shares to each virtual machine. For applications with a computation time demand that varies at run-time, such static solutions result in a low utilization, since the pessimistic worst-case demand has to be reserved at all times, but is often not needed. Therefore, adaptability is desired in order to utilize the shared processor efficiently, but without losing the real-time capability as a prerequisite for the integration.

The first contribution of this thesis is an algorithm for the partitioning of virtual machines to homogeneous cores, which produces mappings that support adaptive scheduling and the protection of safety-critical systems. The second contribution is a virtual machine scheduling architecture that combines real-time guarantees with an adaptive management of the computing power. The third contribution is a technique for real-time virtual machine migration. Together, these contributions enable the integration of independently developed and validated systems on top of a hypervisor. The processing time redistribution in case of mode changes and execution time variations follows the varying demand effectively. Adaptive measures are taken as well to protect critical systems. In case of a worst-case execution time overrun of a critical system, it is attempted to protect its execution by stealing computation time from non-critical systems. In case of a hardware failure, migration is performed to continue the operation of systems on other processors. A prototype demonstrates the feasibility.

Zusammenfassung

Integrierte Architekturen konsolidieren mehrere Funktionen auf einem gemeinsam genutzten Steuergerät. Sie sind für eingebettete Echtzeitsysteme geeignet, die komplexe Funktionalität ressourceneffizient implementieren müssen. Mehrkernprozessoren bergen das Potential die erforderliche Rechenleistung bei reduzierter Größe, Gewicht und Leistungsaufnahme zu bieten. Die größten Herausforderungen integrierter Architekturen sind eine robuste Isolation der integrierten Systeme und eine Ressourcenverwaltung, die jedem System die Erfüllung ihrer Anforderungen garantiert. Hypervisor-basierte Virtualisierung ist eine vielversprechende Integrationsarchitektur für komplexe eingebettete Systeme. Es bezeichnet die Aufteilung der Hardwareressourcen in mehrere isolierte Ausführungsumgebungen (virtuelle Maschinen), von denen jede ein Softwaresystem aus Betriebssystem und Anwendungen beinhaltet.

Diese Dissertation befasst sich mit der Verwaltung der Ressource Rechenzeit durch den Hypervisor, so dass alle Systeme die einen Prozessor gemeinsam nutzen ihre Berechnungen wie erforderlich durchführen können. Stand der Technik ist das Teilen des Prozessors durch die Zuweisung exklusiver Prozessorkerne oder festgesetzter Ausführungszeitanteile zu allen virtuellen Maschinen. Solch statische Ansätze führen jedoch bei Anwendungen deren Bedarf zur Laufzeit schwankt zu einer geringen Auslastung, da der Bedarf für den ungünstigsten Fall zu jeder Zeit reserviert werden muss, oft aber nicht benötigt wird. Aus diesem Grund ist Anpassungsfähigkeit für die effiziente Nutzung des geteilten Prozessors wünschenswert, ohne die Echtzeitfähigkeit als Voraussetzung für die Integration zu verlieren.

Der erste Beitrag dieser Dissertation ist ein Algorithmus für die Aufteilung der virtuellen Maschinen auf homogene Prozessorkerne. Er produziert Zuweisungen, die adaptive Ablaufsteuerungen und den Schutz sicherheitskritischer Systeme unterstützen. Der zweite Beitrag ist eine Technik zur Ablaufsteuerung von virtuellen Maschinen, welche Antwortzeitgarantien mit einer adaptiven Verwaltung der Prozessorleistung verbindet. Der dritte Beitrag ist eine Technik zur echtzeitfähigen Migration virtueller Maschinen. Zusammen ermöglichen es diese Beiträge unabhängig voneinander entwickelte Systeme mithilfe eines Hypervisors zu integrieren. Die Neuverteilung der Prozessorleistung im Falle von Betriebsmoduswechseln und Ausführungszeitschwankungen reagiert effektiv auf Veränderungen des Bedarfs. Adaptive Maßnahmen werden zudem zum Schutz sicherheitskritischer Systeme durchgeführt. Wenn ein solches System die reservierte Ausführungszeit überschreitet, wird versucht durch das Stehlen von Ausführungszeit von unkritischen Systemen das kritische System zu schützen. Im Falle von Hardwarefehlern wird Migration zur Fortsetzung des Betriebs auf einem anderen Prozessor angewandt. Ein Prototyp demonstriert die Machbarkeit.

To my parents.

ACKNOWLEDGEMENTS

I express my deep gratitude to my advisors Prof. Franz-Josef Rammig and Prof. Luis Almeida. Their support, ambition, dedication, tenacity, and expertise are the foundation of this work.

I would like to thank my committee Prof. Marco Platzner, Prof. Christian Plessl, and Dr. Stefan Sauer.

I am also grateful to my brilliant colleagues in Paderborn and Porto.

Foremost, I thank my family for their unwavering support, understanding, and encouragement. I am deeply thankful to Anne, for all the love, support, and patience.

Contents

Contents **vii**

Abbreviations . . . xi

Symbols . . . xi

1 Introduction **1**

1.1 Hypervisor-based Integration . . . 1

1.2 Application Example . . . 4

1.3 Adaptive Scheduling of Virtualized Real-Time Systems . . . 7

1.4 Outline and Contributions . . . 9

2 Fundamentals: Hypervisor-based Multicore Virtualization for Embedded Real-Time Systems **13**

2.1 Embedded Real-Time Systems . . . 14

2.1.1 Embedded Systems . . . 14

2.1.2 Real-Time Computing . . . 15

2.1.3 Mixed-Criticality Systems . . . 21

2.2 Hypervisor-based Virtualization . . . 23

2.2.1 System Virtualization . . . 23

2.2.2 Processor Virtualization . . . 25

2.2.3 I/O Virtualization . . . 28

2.2.4 Virtualization for Mixed-Criticality Systems . . . 29

2.3 Multicore Processors . . . 32

2.3.1 Multicore Scheduling . . . 34

2.3.2 Multicore and Predictability . . . 35

2.4 Virtual Machine Scheduling . . . 37

2.4.1 Hierarchical Scheduling . . . 37

2.4.2 Virtual Processor and Virtual Time . . . 38

2.4.3 Classification and Common Solutions . . . 40

2.5 Summary 43

3 **A Multicore Hypervisor for Embedded Real-Time Systems** **45**
3.1 Problem Statement 46
3.2 Related Work 48
3.3 Proteus Multicore Hypervisor 50
3.3.1 Architecture 51
3.3.2 Configurability 52
3.3.3 Processor Virtualization 55
3.3.4 Paravirtualization Interface 56
3.3.5 Multicore 57
3.3.6 Memory Virtualization 59
3.3.7 Virtualization of Timer and I/O Devices 59
3.4 Evaluation 60
3.4.1 Evaluation Platform: IBM PowerPC 405 60
3.4.2 Execution Times 61
3.4.3 Memory Footprint 65
3.5 Summary 66

4 **Models** **69**
4.1 Workload Model 70
4.1.1 Task Model 70
4.1.2 Virtual Machine Model 73
4.2 Resource Model 76
4.3 Schedulability Analysis 77
4.4 Suitability of the Model 79
4.5 Related Work 81
4.6 Summary 85

5 **Partitioning** **87**
5.1 Problem Statement 89
5.2 Related Work 90
5.3 Branch-and-Bound Partitioning 93
5.3.1 Pruning & Server Transformation 94
5.3.2 Optimization Goals 99
5.3.3 The Algorithm 100
5.3.4 Example 103
5.4 Evaluation 103

5.5 Summary 108

6 Adaptive Partitioned Hierarchical Scheduling **109**
6.1 Problem Statement 110
6.2 Related Work 112
6.3 Scheduling Architecture 114
6.3.1 Server-based Virtual Machine Scheduling 115
6.3.2 Fixed Priority Virtual Machine Scheduling 117
6.4 Adaptive Bandwidth Distribution 118
6.4.1 Distributing Structural Slack 119
6.4.2 The Algorithm and its Computational Complexity 121
6.4.3 Protection under Overload Conditions 123
6.5 Correctness of Bandwidth Distribution 126
6.5.1 Steady State: Temporal Isolation and Minimum Bandwidth Guarantee 127
6.5.2 Correctness during Mode Transitions 129
6.5.3 Correctness of Redistribution of Dynamic Slack 133
6.5.4 Handling of Multiple Mode Change Requests 140
6.6 The Case for Paravirtualization 140
6.7 Integration into Hypervisor and Operating System 143
6.8 Evaluation 144
6.8.1 Scheduling Simulator 144
6.8.2 Execution Times 146
6.8.3 Overhead versus Benefit: Threshold for Slack Redistribution . 148
6.8.4 Memory Footprint 149
6.8.5 Paravirtualization Effort 150
6.8.6 Comparative Evaluation 150
6.9 Summary 158

7 Real-Time Virtual Machine Migration **161**
7.1 Problem Statement 163
7.2 Related Work 164
7.3 Design 167
7.3.1 Migration Policy 167
7.3.2 Integration into the Hypervisor 168
7.3.3 Protocol 169
7.3.4 Migration Test 171

7.3.5 Integration into Real-Time Virtual Machine Scheduling . . . 173
7.4 Evaluation 175
7.4.1 Experimental Setup 175
7.4.2 Memory Footprint & Paravirtualization Effort 176
7.4.3 Execution Times & Downtime 177
7.4.4 Reliability Analysis 178
7.4.5 Case Study: Autonomous Rail Vehicle 182
7.5 Summary 186

8 Conclusion & Future Work **189**
8.1 Summary of Results 189
8.2 Outlook 191

A Publications **193**

List of Figures **195**

List of Tables **199**

List of Algorithms **201**

Bibliography **203**

Abbreviations

AMP Asymmetric Multiprocessing 33

CPU Central Processing Unit 15

ECU Electronic Control Unit 1

EDF Earliest Deadline First Scheduling 20

GPOS General Purpose Operating System 48

I/O Input/Output 28

ISA Instruction Set Architecture 23

ISR Interrupt Service Routine 18

IVCM Inter Virtual Machine Communication Manager 51

MMU Memory Management Unit 31

MPU Memory Protection Unit 31

OS Operating System 15

PIT Programmable Interval Timer 52

RM Rate Monotonic Scheduling 20

RTOS Real-Time Operating System 17

SMP Symmetric Multiprocessing 33

VM Virtual Machine 23

WCET Worst-Case Execution Time 18

Symbols

bdf bandwidth demand factor 154

C computation time of a task 17

χ criticality level 70

dbf demand bound function 74

Δ service delay of a periodic resource 77

δ relative error of budget allocation 154

Δ_i^{max} maximum service delay of virtual machine V_i 97

Γ periodic resource 76

Ω task set 73

P processor (P_i: processor core) 76

p mode change probability 154

PCB periodic capacity bound 96

Π period of a periodic resource 76

sbf supply bound function 77

σ scheduling algorithm 73

T period of a task 18

τ task 18

t_{down} downtime caused by migration 172

t_{expiry} expiry time of dynamic slack 136

Θ allocation of a periodic resource (execution time per period) 76

U utilization of a task set 19

U_{add} additional utilization 120

U_{lax} utilization laxity 71

U_{min} utilization minimum 71

V virtual machine 73

Ξ mapping of virtual machines to processor cores 89

Z criticality distribution 100

Chapter 1

Introduction

Contents

1.1	**Hypervisor-based Integration**	**1**
1.2	**Application Example**	**4**
1.3	**Adaptive Scheduling of Virtualized Real-Time Systems**	**7**
1.4	**Outline and Contributions**	**9**

1.1 Hypervisor-based Integration

The number of functions of complex embedded systems is constantly increasing, and thus, as well the demand for computing power. Traditionally, each function is realized on a dedicated Electronic Control Unit (ECU), an integrated hardware/software component that is typically connected to a bus to exchange data. This federated approach has the disadvantage that each new function requires an additional ECU, resulting in 70 to 100 ECUs for modern cars [Broy et al., 2007, Hergenhan and Heiser, 2008], with a significant impact on hardware costs, space and weight (as well for connecting cables), and power consumption. Buses have limitations regarding number of nodes and cable length, which have to be addressed by multiple buses and gateways between them, increasing the network complexity significantly [Natale and Sangiovanni-Vincentelli, 2010].

Therefore, there is a trend reversal in many industries such as the automotive [Navet et al., 2010, Obermaisser et al., 2009, Reinhardt and Kucera, 2013] or aerospace industry [Filyner, 2003, Watkins and Walter, 2007, Littlefield-Lawwill and Ramanathan, 2007]: multiple functions are consolidated on more powerful ECUs. Especially multicore processors provide an ideal hardware platform to reconcile the diverging goals of realizing additional features, but at the same time reducing the

number of ECUs [Gut et al., 2012]. Multiple stand-alone ECUs are replaced by an integrated modular architecture with a single ECU, reducing the number of ECU boxes, communication nodes (or traffic), cabling, connectors, and power supplies as well as increasing power efficiency (hardware consolidation).

But this integration of independently developed systems on a shared hardware platform must not lead to a loss of functional and time correctness [Natale and Sangiovanni-Vincentelli, 2010]. Many embedded systems are safety-critical and sensitive to time, which is why reliable mechanisms are needed to protect their execution. The integration requires a safe control of the hardware resources and an encapsulated execution of the integrated systems, so that they do not interfere with each other and functional and timing faults are not propagated. It must be prevented that one of the integrated systems corrupts the execution of another system, for example by manipulating private data of the other guest or by not leaving any resources for it.

Encapsulation is particularly important, since the integration of multiple software systems leads in many cases to mixed criticality systems. The criticality of a function or component refers to the severity of failure and might be directly related to a functional safety certification level. In many application domains, one distinguishes between multiple criticality levels, which are characterized by a differing importance for the safety of the system itself and its environment. It must be shown that safety-critical functions are protected and cannot be compromised by other functions. Otherwise, certification results cannot be reused and all functions have to be certified to the highest level.

System virtualization is a technique to integrate multiple software systems in an encapsulated manner. The integrated software systems consist of operating system and application tasks, so that existing legacy software can be reused. A software abstraction layer called hypervisor manages the hardware resources and provides multiple execution environments. It ensures that these so-called virtual machines are isolated from each other. The encapsulation implements a freedom from interference, which includes the integrity of exclusive address spaces and that each integrated system receives the demanded computation time service regarding duration, rate, and maximum time without service. Independently developed software (potentially by different vendors) can be integrated and share the hardware resources with maintained fault containment as well as functional and timing isolation.

System virtualization has the following benefits for embedded systems:

Consolidation. System virtualization replaces multiple processing units by a single (typically more powerful) processing unit. The reduction of the number

of processing units can result in a reduction of the costs for hardware and maintenance as well as of the required space, weight, and power consumption.

Migration to multicore. Multicore technology is a major enabler for virtualization, since virtualization supports the migration of single-core software and use of the full potential of multicore architectures by effective resource management. Virtualization's architectural abstraction enables to present a single-core environment to legacy single-core software and the concurrent hosting of essentially unmodified single-core software stacks.

Operating system heterogeneity. Many embedded systems are characterized by differing operating system requirements (system services, device drivers, non-functional requirements), which are difficult to satisfy by a single operating system [Oikawa et al., 2006, Augier, 2007]. The hypervisor-based integration includes operating systems and continues to provide the adequate operating system, for example, an efficient and highly predictive real-time operating system for safety-critical control tasks and a feature-rich general purpose operating system for a human-machine interface, web protocols, and middleware.

Security. There is a trend towards open embedded systems, which allow the user to add software on his own (field-loadable software), in contrast to software loaded by the manufacturer and remaining unchanged for the entire lifetime of the system. Not knowing which potentially faulty or malicious applications the user will load, a hypervisor can execute it in an isolated virtual machine (sandbox), control the communication, and prevent that it endangers the other software of the system [Brakensiek et al., 2008].

Portability and reusability. One of the key challenges of embedded systems development is to implement increasingly complex functionality in a short time to market. An increased reusability of legacy software addresses this issue. Virtualization's integration by hosting a legacy operating system instead of porting application tasks and its possibility to realize cross-platform portability by applying emulation enables to combine legacy software and new software on state-of-the-art hardware. The hypervisor provides on different hardware an interface that is consistent with the original configuration, so that the legacy software does not have to be modified. Time to market can be reduced by building a system of (validated) systems.

Incremental certification. Virtualization can provide isolated execution environ-

ments that prevent unwanted interactions between software of differing criticality level. This enables the certification of safety-critical software (potentially by a supplier) independently of the coexisting software. It obviates an expensive re-certification when the non-critical software is modified, which tends to happen much more frequently. If hardware and hypervisor are certified, including the encapsulation mechanisms, and the supplier provides certification artifacts in a format that can be merged by the system integrator into the overall safety case, a re-use of certified software without re-certification becomes possible. [Bate and Kelly, 2003, Wilson and Preyssler, 2008]

Of course, system virtualization has drawbacks as well. The integration results in less physical redundancy and a hardware failure will impact all functions [Williston, 2009]. A distributed architecture can be desired in order to place the computation close to actuators and sensors. The hypervisor as an additional software layer increases the system's complexity, especially since it has to be executed in the highest privileged mode of the processor. The indirection layer involves an execution time overhead and increases the interrupt handling latencies compared to the native execution. If Input/Output (I/O) devices are shared, I/O processing is in many cases a bottleneck.

Hypervisor-based virtualization is state of the art for servers in data centers [Nanda and Chiueh, 2005, Smith and Nair, 2005a]. The application of this technique to embedded systems has to consider the domain-specific requirements real-time behavior (determinism), tight resource limitations (memory, computation time, energy), and functional safety.

1.2 Application Example

The number of functions of automotive systems is constantly increasing, e.g., by adding systems for advanced driver assistance, sophisticated infotainment, or vehicular communication [Obermaisser et al., 2009]. This leads to the consideration of system architectures that integrate multiple functions to more centralized systems [Reinhardt and Kucera, 2013], as introduced in the previous section.

One automotive use case for applying virtualization is the head unit [Pelzl et al., 2008, Thiebaut and Gerlach, 2012, Hergenhan and Heiser, 2008]. It has typically a powerful processor and its functionality increased significantly in the last years. Figure 1.1 lists candidate functions that could be integrated on a head unit multicore ECU, subdivided into four categories.

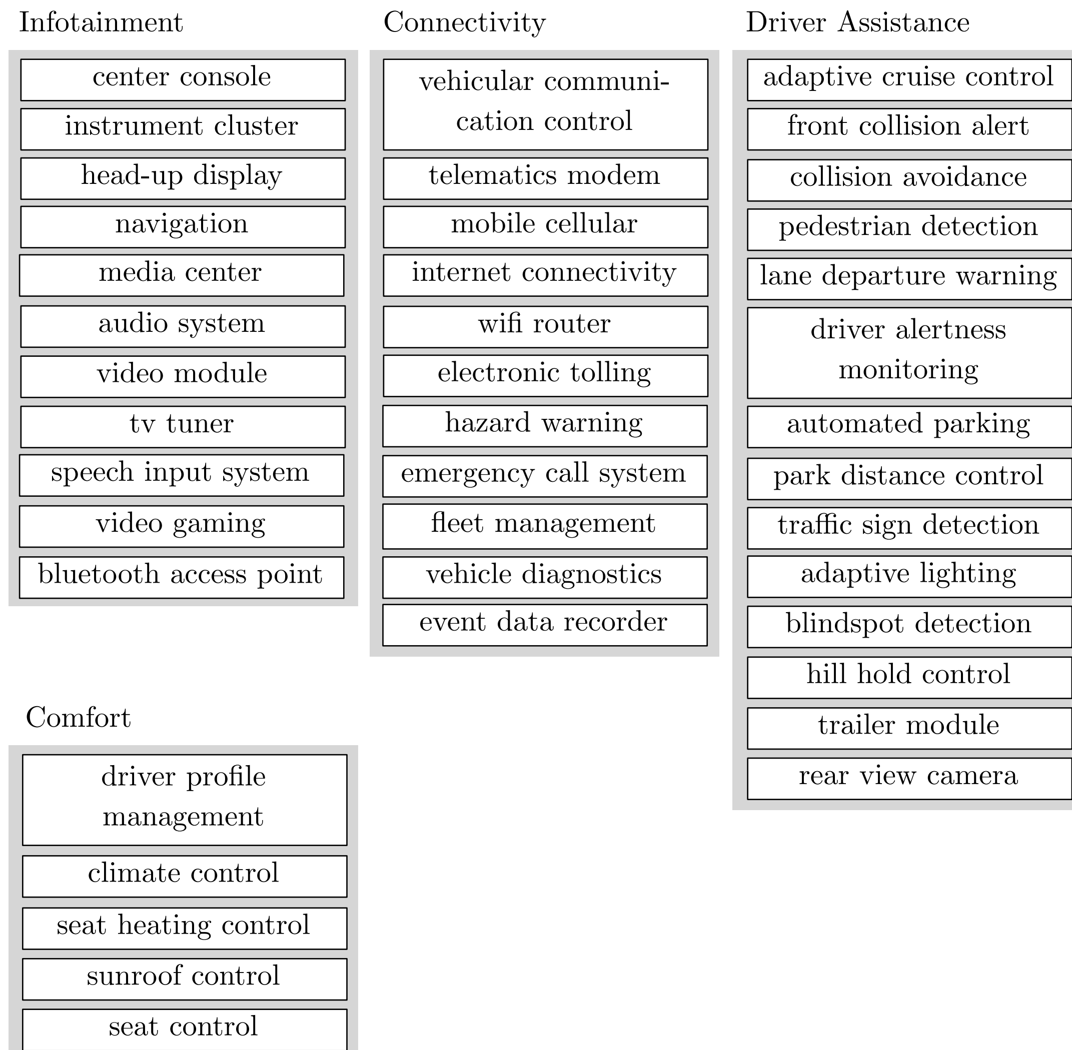


Figure 1.1: Candidate functions for hypervisor-based head-unit integration

First, infotainment functions could be integrated. This is the classic functional category of the head unit and includes navigation, media players, various displays (center console, instrument cluster, head-up display), and potentially sophisticated rear seat entertainment such as video gaming. The second category are vehicular communication control systems, i.e., systems for the communication from vehicle to vehicle, roadside stations, cellular, satellite, or internet [Hasan et al., 2013].

The third category are advanced driver assistance systems, especially those with camera-based object recognition. These systems process images and detect objects. Examples are systems for adaptive cruise control, collision alert and avoidance, pedestrian detection, lane departure warning, driver alertness monitoring, or traffic sign detection. Many of these systems operate on the same sensor data and they can share the results of the image preprocessing [Bucher et al., 2003]. Finally, comfort functions such as driver profile management (stores car settings) or systems for the control of climate, seat position, or sunroof might be integrated.

For multiple reasons, these categories are well suited for an integrated architecture. The number of features in these categories increased significantly in the last years. Moreover, many functions are dependent and/or cooperate, within and across categories. Data from the lane departure warning system is for example input for the driver alertness monitoring. The displays are controlled by multiple functions, resulting in synchronization challenges. There is already a connection from cameras to the head unit in order to display the images, e.g., for parking distance control, which is why camera-based driver assistance systems could be executed on the head unit ECU. There is a trend towards open infotainment systems, which enable the customer to load software, inspired by the world of mobile apps. Virtualization's encapsulation and fault containment protect the other functions from faulty or malicious software.

In addition, the head unit requires support of multiple operating systems, an important feature of hypervisor-based virtualization. Different functions are best served by different operating systems. A general purpose operating system provides lots of support for the development of software with human-machine interface (e.g., graphical user interfaces and touch screens) and a well-known look and feel. A system for in-car communication or connectivity to the outside requires a deterministic real-time operating system that guarantees response times. Safety-critical functions require a certified real-time operating system, typically based on AUTOSAR (AUTomotive Open System ARchitecture [Fürst et al., 2009, Bunzel, 2011]). AUTOSAR and virtualization partially share the common motivation to increase software reusability

and reduce hardware dependency, but apply integration at different levels. The hosting of multiple operating systems is not in the scope of AUTOSAR. An AUTOSAR software stack might be executed as a guest system within a virtual machine.

The integration of these candidate functions leads to a coexistence of systems with different criticality levels. Driver assistance systems are safety-critical with hard real-time requirements, especially if they influence the driving speed, as it is the case for adaptive cruise control or collision avoidance systems. Connectivity or infotainment systems are non-safety-critical and typically quality-of-service driven. An additional observation is that many of the candidate systems are characterized by varying resource requirements, especially the driver assistance systems whose main source of input data is image processing. The required execution time of these systems depends on illumination and weather conditions as well as on the specific driving situation, since it determines the number of objects to detect.

The resource requirements vary as well based on the mode of the systems. The requirement of an infotainment system depends on the requested activity. Video gaming is for example considerably more resource-intensive than playing music. And there is an increased potential to temporarily deactivate functionality that is not in constant use while the car is operational [Liebetrau et al., 2012]. This is obvious for infotainment functions, which can be turned off when not used by any passenger. Subject to the driving situation, adaptive cruise control, lane departure warning, park distance control, or hill hold control can be enabled or disabled. The rear view camera system must only be enabled when the car is reversing. The speech input system becomes active only when enabled by a manual control input such as pressing a button on the steering wheel.

1.3 Adaptive Scheduling of Virtualized Real-Time Systems

The hypervisor is responsible for the management of the hardware resources. This includes the management of the resource CPU, which is known as scheduling. In the context of this thesis, the considered processors are homogeneous multicore processors, with uniformly shared main memory. If the number of virtual machines exceeds the number of processor cores, hypervisor-based virtualization implies a two-level hierarchical scheduling: the hypervisor schedules the virtual machines and the hosted operating systems schedule the application tasks.

Hierarchical scheduling is a direct consequence of the integration level of system

virtualization (software stacks including operating system). The reuse of software and certification results requires isolation at virtual machine level and separation of scheduling concerns. The operating systems should schedule the applications according to their specific scheduling policy and without insight into the scheduling of other virtual machines.

In the following, we derive requirements for a scheduling technique for integrated architectures as introduced in the last two sections. These architectures can be described by the following characteristics:

- **C1** coexistence of independently developed guest systems of operating system and application tasks,
- **C2** coexistence of guests of different criticality levels, especially safety-critical (e.g., driver assistance systems) and non-critical (e.g., infotainment systems),
- **C3** coexistence of guests of different resource requirement characteristics, especially hard real-time and QoS-driven,
- **C4** existence of guests with real-time requirements, which benefit from additional resource allocations (e.g., computer vision systems),
- **C5** guests with multiple operational modes, incl. deactivation,
- **C6** guests with varying execution time demand.

C1 defines the separation of scheduling concerns. The desired composability of software from previous projects and different vendors requires that it is possible to analyze the schedulability of a guest system independent of the other systems. **C2** asks for criticality-aware scheduling. The remaining characteristics **C3-C6** ask for a scheduling technique that addresses different and at runtime varying resource demands (incl. varying execution times and operation modes).

Two major requirements for the virtual machine scheduling can be derived from the characteristics of the hosted guest systems: temporal isolation and adaptability. Temporal isolation is crucial for the ability to integrate real-time systems, especially safety-critical ones. The scheduling has to ensure that no guest system compromises the real-time behavior and safety characteristics of the alongside hosted guest systems. Each guest system must receive a guaranteed share of the processor time. The bandwidth must be sufficiently large and be provided in an appropriate frequency, so that the guest system meets its real-time constraints.

Temporal isolation is a must, but adaptability is desired. The scheduling should be able to adapt to varying computation time requirements of the guest systems. Computation time variations are caused by mode changes (incl. enabling/disabling) and by guest systems temporarily not needing their reserved worst-case demand. Mode changes often impact the resource requirements significantly and for some time, especially when functions are disabled, and the scheduling should react by redistributing processor shares according to the new situation.

Due to dynamic environments, it is a complex task to predict characteristics of the computational load at design time [Buttazzo, 2006]. In addition, the reserved computation time is often not needed. This is especially problematic for safety-critical systems, since it is necessary to allocate computation time according to the worst-case assumptions. The worst-case execution time is in general very pessimistic, in particular for multicore processors, since the on-chip shared resource contention (e.g., memory bus contention) has to be considered [Kotaba et al., 2013]. However, worst-case scenarios occur rarely and a fraction of the reservation is wasted in all other cases, leading to a low average processor utilization.

Static resource allocation is in general very inefficient for systems that are characterized by varying computation time demand and inherently leads to resource fragmentation. Reserved but unused computation time cannot be reclaimed to improve the performance of other guests, but is wasted as idle time. A reaction to mode changes is not possible and by consequence the hardware has to be dimensioned for the situation that all subsystems are enabled and operating in the mode with the highest resource requirements. An adaptive scheduling is of great potential for hypervisor-based systems, since it is able to redistribute on mode change and since it can make the unneeded fraction of the worst-case reservations available to other guest systems.

The challenge of this thesis is the reconciliation of these two conflicting requirements: temporal isolation is a prerequisite for the integration of real-time systems and the certification of safety-critical systems, but adaptability is desired in order to utilize the shared processor cores efficiently and avoid the waste of resources. How much adaptability are we able to afford without losing temporal isolation?

1.4 Outline and Contributions

System virtualization provides opportunities for system design, but as well challenges with regard to software engineering (e.g., modeling of functional and non-functional properties on system and subsystem level), dependability, safety, and deterministic

timing behavior (predictable composition). New models, methods, and tools for design, development, and verification have to be developed for the integration of multiple software systems. In this context, this thesis focuses on two aspects, which can be entitled as "the adaptive management of the resource computation time: the scheduling of virtual machines and runtime virtual machine migration."

Target architectures are embedded systems with real-time constraints, limited resources, and dynamic behavior, which is hardly predictable due to complex dependencies with the environment and shared on-chip resource contention of a multicore processor. The required worst-case design approach results for such systems in many cases in a very low resource utilization. Instead of statically allocating the resource computation time, the proposed adaptive scheduling technique reacts to mode changes and updates the allocations. Slack, which is generated if a guest system does not need its worst-case reservation, is reclaimed and made available to other guests. Different types of resource requirement characteristics can be modeled and the scheduling meets the specific constraints. A guaranteed minimum share of the computation time guarantees temporal isolation, but adaptive measures enable the efficient use of the processor, without leading to a violation of real-time constraints.

Adaptive measures are taken as well to increase the reliability of safety-critical guest systems. In case of a worst-case execution time overrun of a critical guest system, it is attempted to protect its execution by assigning additional computation time, which is stolen from non-critical guests. In case of certain hardware failures, guest systems can be transferred to a different electronic control unit, where their operation is continued (migration). Next to the algorithms and policies, the co-design of hypervisor and operating system and the implementation of a prototype on actual embedded hardware is a focus of this thesis and demonstrates the feasibility of the developed concepts.

This thesis makes the following major scientific contributions:

An algorithm for the partitioning of virtual machines to processor cores. (Chapter 5) An algorithmic solution is in contrast to the manual mapping of virtual machines to cores, which is state of the art. The algorithm includes the correct dimensioning of periodic resources, a model for the computational power supplied by a shared processor. The algorithm systematically generates candidate solutions and tests their schedulability by comparing the computation time demand of the guest systems and the computation time supply of the shared processor. It minimizes the overall required computation bandwidth by exploiting the freedom of periodic resource design to create favorable period

relationships. It considers criticality levels and produces mappings that provide more possibilities to protect safety-critical guest systems and to benefit from an adaptive scheduling (as for example done by the second contribution).

An adaptive virtual machine scheduling architecture. (Chapter 6) It advances the state of the art by combining temporal isolation and real-time guarantees with adaptive scheduling. The technique overcomes the limitations of static resource allocation. It performs a redistribution of computing power in case of mode changes and varying execution times and attempts to protect critical guest systems in case of a worst-case execution time overrun. A novel elastic bandwidth management algorithm is non-iterative, in contrast to existing ones, and therefore characterized by a smaller and more predictable execution time overhead.

A technique for real-time virtual machine migration. (Chapter 7) The migration approach is applied in order to continue the functioning of guest systems despite certain hardware failures. The technique advances the state of the art in that it is aware of real-time requirements and addresses the real-time scheduling issues service outage (non-execution) due to the network transfer to the target ECU and integration into the scheduling on the target. A migration protocol and a co-design of hypervisor and paravirtualized guest operating system are presented.

This thesis is structured as follows. Chapter 2 introduces the background with fundamental results of prior work and characteristics of the field of application. It discusses embedded systems and real-time computing, functional safety and criticality levels, multicore processors and the related predictability issues, hypervisor-based virtualization in general and virtual machine real-time scheduling in specific.

Chapter 3 lists functional and non-functional requirements for a hypervisor for embedded real-time systems and presents the multicore hypervisor *Proteus* that meets the requirements of this application domain. This hypervisor is used as a platform for the prototype-based evaluation of the scheduling technique of Chapter 6 and the migration approach of Chapter 7. Design and evaluation of the hypervisor were published in [Gilles et al., 2013].

Chapter 4 defines the used models for workload and processor platform. A demand bound function denotes the maximum cumulative computation time demand of a virtual machine and is used as a temporal interface. The periodic resource model provides a formalization of the minimum cumulative computation time supply of a

shared homogeneous multicore processor. Schedulability analysis is performed based on the comparison of demand bound function and supply bound function.

The two subsequent chapters study an adaptive partitioned multicore scheduling technique for the hypervisor-based integration of multiple real-time systems. Chapter 5 defines the problem of mapping virtual machines that host real-time systems onto a multicore processor and presents a partitioning algorithm. Chapter 6 introduces a dynamic server-based hierarchical scheduling policy including a bandwidth distribution algorithm, which is applied after the partitioning. The combination of these two contributions guarantees that all guest systems meet their real-time requirements, but enables as well adaptive measures. The results of these chapters were published in [Groesbrink and Almeida, 2014] and [Groesbrink et al., 2014a].

Chapter 7 studies virtual machine migration from one ECU to another one at runtime as a hardware fault reaction measure. The approach is aware of real-time requirements, predicts the service outage due to the network transfer and integrates the migrating VM into the scheduling on the target ECU. This work was published in [Groesbrink, 2014].

Chapter 8 concludes the thesis and gives some pointers for future research.

Chapter 2

Fundamentals: Hypervisor-based Multicore Virtualization for Embedded Real-Time Systems

Contents

2.1 Embedded Real-Time Systems 14
2.1.1 Embedded Systems 14
2.1.2 Real-Time Computing 15
2.1.3 Mixed-Criticality Systems 21
2.2 Hypervisor-based Virtualization 23
2.2.1 System Virtualization 23
2.2.2 Processor Virtualization 25
2.2.3 I/O Virtualization 28
2.2.4 Virtualization for Mixed-Criticality Systems 29
2.3 Multicore Processors 32
2.3.1 Multicore Scheduling 34
2.3.2 Multicore and Predictability 35
2.4 Virtual Machine Scheduling 37
2.4.1 Hierarchical Scheduling 37
2.4.2 Virtual Processor and Virtual Time 38
2.4.3 Classification and Common Solutions 40
2.5 Summary 43

This chapter introduces the background with the major concepts of prior work and characteristics of the field of application. It includes an introduction to embedded systems and real-time computing, criticality levels, multicore processors, as well as hypervisor-based virtualization in general and virtual machine scheduling in specific.

2.1 Embedded Real-Time Systems

2.1.1 Embedded Systems

Embedded systems can be characterized as electronic programmable subsystems that are an integral part of a technical system [Bouyssonouse and Sifakis, 2005]. They are *reactive*, i.e., respond to events and state changes of the environment in which they operate.[1] Sensors and actuators are the embedded system's interfaces. The sensors periodically observe attributes of objects in the environment that are controlled by the system or influence the system. Based on this sensor data and the previous state, the control system computes control points for the actuators in order to influence the environment [Kopetz, 1997]. There are often response time constraints for the computation, as introduced in the following section. In addition, embedded systems can have a human-machine interface, realized for example with a touchscreen.

Definition 1. ***Embedded System.*** *An embedded system is a computer control system (combination of hardware and software) that operates with a dedicated function as an integral part of a larger technical system (often including hardware and mechanical parts).*

Embedded systems are often characterized by strict resource constraints, especially regarding memory and processing time, but as well regarding size and weight (especially for hand-held devices), and power consumption (battery operation or limited cooling possibility). They are often extremely cost-sensitive since they are mass-produced.

Many embedded systems are safety-critical. Human lives or the intactness of facilities or equipment directly depend on the correct operation [Kopetz, 1997]. For this reason, many safety-critical systems need an approval by a certification authority such as governmental agencies. This aspect is introduced in Section 2.1.3.

[1] Reactive systems are seen in contrast to *transformational systems*, which compute the corresponding output to a certain input and then terminate [Olderog and Dierks, 2008].

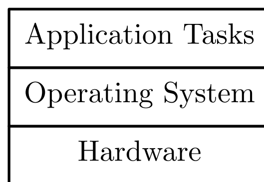


Figure 2.1: Computer system with operating system

2.1.2 Real-Time Computing

Task Scheduling

Most complex computing systems use an Operating System (OS), a software layer between hardware and the functionality-implementing application tasks (see Figure 2.1). The OS manages the computer resources (processors, main memory, disks, I/O devices) and provides services as well as a hardware abstraction that is easier to program [Tanenbaum and Woodhull, 2006].

Modern operating systems are so-called *multitasking systems*, designed to handle multiple application *tasks* (also known as processes) concurrently. The tasks share the hardware and operating system resources in order to increase the resource utilization. Moreover, the division of the functionality into multiple tasks eases software development. Tasks may depend on each other and cooperatively implement functionality, or they may be unrelated except of being executed on the same hardware.

Definition 2. ***Operating System.*** *An operating system is a software layer that manages the hardware resources, controls the execution of tasks, and provides services and hardware abstraction to tasks and programmers.*

The Central Processing Unit (CPU) has to be managed by time-multiplexing: only a single task can use it at any time. The CPU resource management is known as *scheduling.* According to a specific scheduling policy, the OS assigns the processor at each point in time to exactly one of the tasks. The idle task is executed when no task is executable, a special task without functionality. The *scheduler* is the operating system's component that implements the scheduling policy.

Definition 3. ***Task.*** *A task is a schedulable unit of computation that is executed by the CPU in a sequential manner.*

The suspension of the running task in order to execute another task is called *context switch.* The OS saves the necessary information (basically the content of the registers) to allow a future continuation of the task's execution at the exact

same point [Horner, 1989]. A context switch might for example be performed when the running task has to wait for the end of an I/O operation or after a maximum allowable time of uninterrupted execution. A timer is used to determine when the time slot of the currently running task expires [Tanenbaum and Woodhull, 2006]. Schedulers that are able to suspend tasks during their execution in order to make a scheduling decision are called *preemptive*. A non-preemptive scheduler always lets a task run to completion. Preemptive schedulers prevent a task from blocking all other tasks and therefore avert the failure of the whole system if a task is executing an infinite loop [Nutt, 2000]. The interruption of the running task is called preemption.

Different scheduling algorithms pursue different goals. Possible criteria are fairness (even shares of the computation time for the tasks), processor utilization (avoid idle time), or response time (respond to requests quickly). An interactive system should for example minimize the response time to prevent user frustration [Tanenbaum and Woodhull, 2006].

Definition 4. ***Task Schedule.*** *Given a set of tasks* $\{\tau_1, ..., \tau_n\}$*, a schedule is an integer step function s which at any time t assigns a task* τ_k $(1 \leq k \leq n)$ *to the processor:* $s(t) = k$ *(if the processor is idle at time t:* $s(t) = 0$*) [Buttazzo, 2000]. A schedule is generated by the scheduler of an operating system.*

Introduction to Real-Time Computing

Real-time systems are characterized by precise and strict timing constraints for their execution: they are required to guarantee response times. *Real* indicates that these timing constraints are derived from the embedded system's environment, as a system interacts in a well-defined relation to the physical time [Olderog and Dierks, 2008]. A real-time system fails not only if its results are wrong, but also if it cannot provide these results prior to the given deadlines [Kopetz, 1997].

It is important to understand that real-time behavior does not demand that the results have to be produced very fast. Real-time computing should not be mistaken for high performance computing. Real-time expresses the quality of *predictability*, achieved through *deterministic* behavior of the embedded system, i.e., the time required to complete any function must be limited and predictable. As long as the results are provided prior to the deadline, the performance is sufficient. Real-time systems guarantee the adherence to the system's time limits, not extra fast computation [Stankovic, 1988].

Definition 5. ***Real-Time System.*** *A real-time system is a computer system with timing constraints, which "relate the execution of a task to real time, which is physical*

time in the environment of the computer executing the task" [Lee and Seshia, 2011]. These constraints are typically specified in terms of response time deadlines, by which the execution of a task must be completed. The correctness of the system depends not only on the logical results of the computation, but also on the time at which these results are available.

An OS that is designed to guarantee response times of the executed tasks is termed Real-Time Operating System (RTOS). Required are a deterministic behavior of all OS components, keeping of time in well-defined relation to the physical time, and a task scheduler that controls the task execution according to a policy that guarantees compliance of all tasks with their timing constraints.

Definition 6. *Real-Time Operating System.* *A real-time operating system is an operating system that is able to control the execution of tasks with timing constraints and to guarantee that these constraints are met.*

Real-time systems can be categorized into *hard* and *soft* real-time systems. For a hard real-time system, a miss of a deadline (i.e., result not available prior to it) is unacceptable and leads to the failure of the entire system, with catastrophic consequences in a safety-critical system. An example is the control of a car's airbag: if the airbag control unit does not trigger the ignition not later than 30 milliseconds after the first moment of vehicle contact, the airbag is not fully inflated when the passenger's head hits the steering-wheel.

In contrast, the value of a result of a soft real-time system decreases after the deadline, but the system does not automatically fail. It could be demanded that the proportion of tardy operations does not exceed a specific limit. The quality of the system's result would decrease, but without rendering it useless [Freedman et al., 1996]. An example is a video messenger system. If it fails to handle the transfer and processing of some frames on time, the display freezes temporary, lowering the service quality.

Real-Time Task Model

The scheduling decisions of an RTOS have to be based on information about the timing constraints of the real-time tasks. These constraints are specified with the following parameters (Figure 2.2) [Buttazzo, 2000]:

- *Arrival time a*: time at which a task becomes ready for execution (synonym: release time).

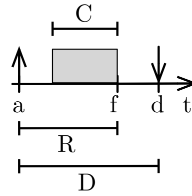


Figure 2.2: Parameters of a real-time task (upward arrow indicates task arrival, downward arrow indicates deadline)

- *Computation time C:* duration needed by the processor to execute the task. The task can run for a single time span of length C or for multiple time spans which add up to a total of C. If the computation time of a task varies, the maximum possible computation time has to be considered, the so-called Worst-Case Execution Time (WCET).
- *Finishing time f:* time at which the task finishes its execution.
- *Absolute deadline d:* time by which the execution of the task must be finished. The difference between absolute deadline d and arrival time a is called *relative deadline D.*
- *Response time R:* time span between finishing time f and arrival time a. A task completes in time if its response time is less than or equal to its relative deadline.
- *Lateness L:* difference between finishing time f and absolute deadline d. It represents the delay of a task's completion, with a negative lateness in case of a completion before the deadline.

One distinguishes between *periodic* and *aperiodic* real-time tasks. Both kinds are characterized by an infinite sequence of instances, but consecutive instances of a periodic task are activated regularly with a time lag of exactly one *period* T. The relative deadline D of a periodic task is often equal to its period [Buttazzo, 2000]. Periodic tasks are used to read or produce data at a given rate, for example the periodic readout and processing of sensor data. Aperiodic tasks are not activated regularly at a constant time. External events cause the activation of an aperiodic task instance to handle it. An Interrupt Service Routine (ISR) is an example for an aperiodic task [Briand and Roy, 1999, Buttazzo, 2000].

Definition 7. ***Periodic Real-Time Task.*** *A periodic real-time task τ is a task with an infinite sequence of identical computation activities (instances), which are activated at a constant rate based on the period T. The first instance is released at the*

phase ϕ. The activation time of the kth instance is given by $\phi + (k-1) \cdot T$ [Buttazzo, 2000]. In the context of this thesis, the real-time constraints are specified by the relative deadline D, which is equal to the tasks period T (implicit deadlines): the execution of the kth instance must be finished at $\phi + k \cdot T$.

Real-time systems typically execute a set of periodic real-time tasks.

Definition 8. ***Task Set.*** *A task set is a set of periodic real-time tasks that is executed by a real-time operating system. The utilization factor U is the fraction of processor time spent in the execution of it. It is a measure of the computational load on the processor and for a task set of n tasks given by [Buttazzo, 2000]:*

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

Real-Time Scheduling

The scheduler of a real-time operating system has to guarantee that all real-time tasks are executed in compliance with their timing constraints. A schedule is *feasible* if all tasks are executed according to their constraints. A task set is *schedulable* if there is an algorithm that can produce a feasible schedule [Buttazzo, 2000]. The most important scheduling algorithms for real-time systems are presented in this section. Real-time scheduling algorithms can be classified as follows [Buttazzo, 2000]:

Offline and *online* scheduling differ in regard to the time at which the scheduling decisions are taken. Offline algorithms create the entire schedule before runtime. The result is stored and enforced at runtime [Xu and Parnas, 1993]. This is of course only applicable to static systems with complete information at design time about the points in time at which requests for task computation occur. Storing the entire schedule is possible since the schedule of a periodic task set repeats itself every *hyperperiod*, equal to the least common multiple of all task periods. Online algorithms take the scheduling decisions at runtime and can consider dynamic task set changes such as the entering of a new task [Stankovic et al., 1998].

Fixed and *dynamic priority scheduling* differ in regard to whether the scheduling decisions are based on static or dynamic task parameters. Fixed priority scheduling is based on static parameters, which never change during execution (equal for all task instances). Dynamic priority scheduling algorithms determine the task priorities at each scheduling event based on dynamic parameters [Stankovic et al., 1995].

In the following, three common algorithms are presented: Timeline Scheduling as an offline algorithm, Rate Monotonic Scheduling as a fixed-priority online algorithm, and Earliest Deadline First as a dynamic-priority online algorithm.

Timeline Scheduling (TS). Timeline Scheduling (also known as Cyclic (Executive) Scheduling [Baker and Shaw, 1989]) is an offline clock-driven scheduling algorithm, which divides the time axis into slices of equal length, the so-called minor cycle. One or more tasks are assigned to each minor cycle. This assignment must follow the frequencies of the tasks. For example, if a task's frequency is twice as high as the frequency of another task, it is assigned to twice as many minor cycles. The tasks within a minor cycle are scheduled successively in a non-preemptive manner (resulting in a low implementation complexity). The optimal length for the minor cycle is the greatest common divisor of the task periods. The schedule repeats itself after each hyperperiod, called major cycle. TS guarantees a feasible schedule if the sum of the worst-case execution times of all tasks within each time slice is less than or equal to the minor cycle [Buttazzo, 2000].

Rate Monotonic Scheduling (RM). RM is a preemptive online algorithm for periodic tasks and schedules according to fixed priorities, which are assigned based on the rate of their requests: the shorter the period (equal for all task instances and known at design time), the higher the task's priority [Liu and Layland, 1973a]. RM preempts the running task at any time when a task with a higher priority becomes ready to run. The produced schedules are optimal for the class of fixed-priority scheduling algorithms for periodic tasks with hard deadlines: if RM fails to produce a feasible schedule for a specific task set, no other algorithm of this class can. RM guarantees the schedulability of a set of n periodic tasks if the utilization factor U is less than $n(2^{1/n} - 1)$, which converges for large n to $ln\ 2 \approx 69\%$ [Liu and Layland, 1973a]. This bound is sufficient, but not necessary, and the schedulable utilization is for many task sets higher [Lehoczky et al., 1987b].

Earliest Deadline First Scheduling (EDF). EDF is a preemptive online algorithm and schedules based on dynamic priorities: the earlier the absolute deadline of the current task instance, the higher its priority [Liu and Layland, 1973a]. EDF is optimal for the class of dynamic priority scheduling algorithms: it minimizes the maximum lateness [Dertouzos, 1974], implicating that it guarantees to produce a feasible schedule if one exists. EDF is applicable to periodic and aperiodic tasks since it does not depend on periodicity [Horn, 1974].

Many real-time systems are characterized by a *hybrid task set* of periodic hard real-time and aperiodic soft real-time tasks. The periodic tasks might be scheduled by RM or EDF. The aperiodic tasks are often scheduled *in background*: whenever no periodic task is running, the idle processor is used to execute aperiodic tasks. Background scheduling can be realized without modification of the periodic task scheduler

by two ready queues: a high-priority queue for periodic tasks and a low-priority queue for aperiodic tasks. The head of the low-priority queue is only dispatched when the high-priority queue is empty [Buttazzo, 2000, Stallings, 2005]. However, a high utilization of the periodic task set leaves in many cases infrequent background service opportunities, resulting in a large average response time for aperiodic requests.

Servers are a technique to improve the average response time of aperiodic tasks. A server is a periodic task with the purpose to service aperiodic requests. It is scheduled like any other periodic task and uses its execution time to run aperiodic tasks. In contrast to background scheduling, aperiodic tasks do not have automatically the lowest priority, but are executed with the priority of the server. The service of the server is limited by its budget, which is reduced whenever the server runs. If none is left, the server cannot execute aperiodic tasks until its capacity is replenished [Lehoczky et al., 1987a]. There are different types of servers, varying in when and to which amount the budget is replenished. For a detailed overview see [Buttazzo, 2000].

2.1.3 Mixed-Criticality Systems

The *criticality* of a function refers to the severity of failure and the potential negative impact on the intended functionality and the system's environment. In general, the implementing component (e.g., task or subsystem) inherits the criticality of the function [Papadopoulos et al., 2010]. In many application domains, one distinguishes between multiple criticality levels, which are characterized by a differing importance for the safety of the system itself and its environment.

Intuitive criticality levels are safety-critical, mission-critical, and non-critical, introduced in the following with the example of an avionics system. The failure of a *safety-critical* function such as the engine control of the flying aircraft can impact human safety. Other functions such as navigation or communication to the ground control are not critical for the welfare of the system and its environment, but for the mission success (purpose of the system). Finally, there are functions that are not critical for safety or mission success, such as the in-flight entertainment system.

Criticality is directly related to *functional safety*. Functional safety is achieved if a system is free from unacceptable risk of injury to the health of people or damage to the environment [Storey, 1996], as it is of high importance for example for transportation systems (aircraft flight control, train control, automotive systems), medical devices, or nuclear systems.

In industries that are subject to certification of functional safety, standards such

as ISO 26262 for automotive systems [ISO, 2011] or DO-178B for airborne systems [RTCA/DO, 2012] specify criticality levels based on risk classification schemes and needed risk reduction factors. The criticality level indicates safety requirements (usually defined in terms of quantitative targets [Papadopoulos et al., 2010]) and for each function the requested level has to be identified in order to design and implement the system to comply with this level. The lower the required failure probability, the higher the criticality level, and usually the higher the development costs.

ISO 26262 specifies five Automotive Safety Integrity Levels (ASILs) based on the exposure (frequency of occurrence), controllability (ability to avoid a harm through the timely reactions of the persons involved), and severity (estimate of the extent of harm to individuals) of any critical hazard. The lowest ASIL is A, while the highest is D, plus ASIL QM for non-safety-critical functions (no safety requirements). For example, a component whose failure will (1) result in a common event that (2) will lead with a high probability to a loss of control of the vehicle and (3) may cause great harm (serious injury or death) will require certification according to ASIL D (e.g., electric steering component).

Functional safety standards specify for each criticality the necessary level of assurance against failure. A certified system can then be claimed to be safe to a particular criticality level. A criticality-level-specific rigor in development is required and different processes, rules, and tools are recommended in order to achieve the demanded safety margins. Important for real-time systems is the determination of the WCETs, which becomes dependent on the task's criticality level. A high criticality level might require the determination of worst-case execution paths by a static code analysis and a subsequent counting of processor cycles under extreme pessimistic assumptions regarding cache state. For a certification according to a lower criticality level it might be sufficient to run experiments and measure the runtimes. By consequence, the higher the criticality level, the higher the required confidence, the larger the required safety margin, the more pessimistic the obtained WCET [Baruah et al., 2010b].

As motivated in the previous chapter, an increasingly important trend for embedded real-time systems is the co-existence of multiple criticality levels:

Definition 9. ***Mixed-Criticality System.*** *A mixed-criticality system is a computer system that executes software components of different distinct criticality levels (safety-critical and non-critical or of different certification levels) in an integrated manner on a common hardware platform.*

In the context of this thesis, we consider computer systems with guest systems (operating system and application tasks) of differing criticality level executed by a

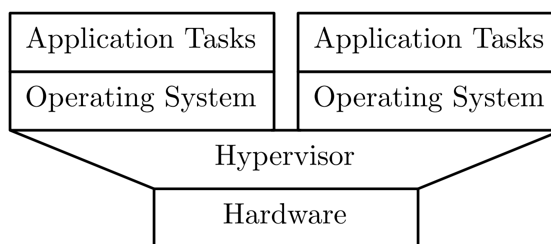


Figure 2.3: Platform replication by system virtualization

hypervisor.

2.2 Hypervisor-based Virtualization

2.2.1 System Virtualization

Virtualization is a classic computer science concept and refers to the abstraction of the physical characteristics of a given resource in order to allow for a transparent and highly flexible resource sharing among multiple subsystems. The basic concepts are about 50 years old. For example, a multitasking operating system virtualizes the computer resources (first of all the CPU, often as well the memory) and creates the illusion that multiple tasks are run at the exact same time. The task programmer can assume that the task gets exclusive hardware access, however, multiple tasks share the resources.

Virtualization can be applied on different levels. *System virtualization* refers to dividing the resources of the entire computer hardware into multiple execution environments (platform replication) [Smith and Nair, 2005b]. It lets a real system appear as a different virtual system or even multiple virtual systems. The interface of the virtualized system and its resources are mapped to those of the real system. The virtualization layer, the so-called *hypervisor* or *virtual machine monitor (VMM)*, is added in between hardware and OS, and enables the sharing of the underlying physical hardware among multiple software stacks of OS and application tasks (Figure 2.3). Each OS runs as a *guest system* within a Virtual Machine (VM), an isolated duplicate of the real machine [Popek and Goldberg, 1974a]. The real machine is the set of hardware resources, including processor, memory, and I/O devices.

The hypervisor provides the abstraction of the real machine, manages the mapping of hardware resources to the VMs, and controls the execution of the VMs. A VM is the interface provided by the hypervisor to the guest systems. It can differ from the interface of the real machine, regarding Instruction Set Architecture (ISA)

and available resources. The ISA (composed of the native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, external I/O, and the set of assembler opcodes) provides the interface for the system software. The virtualization layer has to map the ISA that is provided to the VM to the ISA of the physical machine. An instruction that cannot be executed directly on the physical machine has to be interpreted [Heiser, 2007].

A VM does not have to offer the exact same resources as the real machine, neither in quantity nor in type. Some physical resources such as memory or disk storage can be partitioned, so that each virtual machine uses a fraction. Other resources have to be shared by time-division multiplexing, e.g., the CPU (if the number of VMs exceeds the number of available processor cores). A fundamental requirement is that the hypervisor is in control of the system resources. When a guest OS performs an operation that directly involves shared hardware resources, the hypervisor intercepts the operation and handles it. When, for example, a guest OS tries to set a processor control flag, the hypervisor intercepts, stores the current value of the flag, sets the control flag and resumes the execution of the VM, but resets it before a different VM is executed.

Definition 10. ***Virtual Machine.*** *A virtual machine is an execution environment as created by the hypervisor's virtualization of the physical hardware. It provides a complete substitute for the real machine and enables the execution of an operating system (system virtual machine).*

Definition 11. ***Hypervisor.*** *A hypervisor (or virtual machine monitor) is a piece of computer software, hardware, or combination of software and hardware that provides and controls multiple virtual machines by virtualizing the hardware. Each virtual machine executes a guest system.*

Definition 12. ***Guest System.*** *A guest system is a software stack consisting of operating system and application tasks that is executed by a virtual machine on top of a hypervisor (the host).*

Popek and Goldberg defined three fundamental requirements for a hypervisor [Popek and Goldberg, 1974a]:

Equivalence. A virtual machine is required to be able to run all software which can be executed on the real machine - and vice versa. The execution of software on the hypervisor is identical to its execution on hardware, apart from a possibly slower performance and lower resource availability.

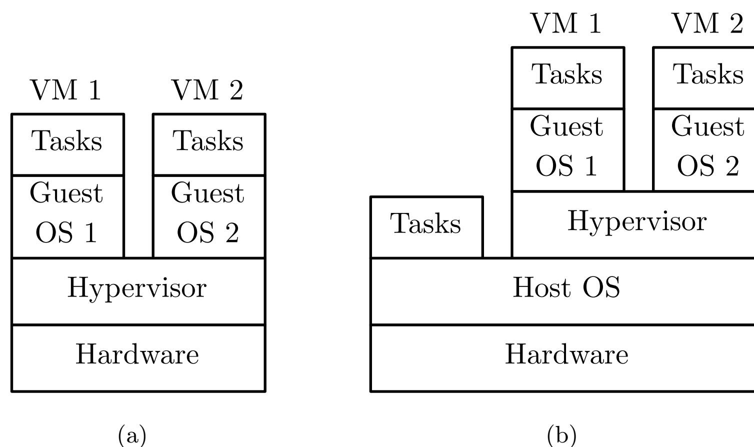


Figure 2.4: (a) Native hypervisor (type I) vs. (b) hosted hypervisor (type II)

Resource Control. The hypervisor retains control of the hardware resources. It must not be possible for guest software to affect the system resources directly. Software cannot break out of the VM.

Efficiency. The vast majority of the guest software's instructions must be executed natively on the hardware, without an interpretation by the hypervisor.

Smith and Nair demand from hypervisors only to satisfy the conditions equivalence and resource control, and call hypervisors that additionally satisfy the third condition *efficient hypervisors* [Smith and Nair, 2005a].

Figure 2.4 shows two different types of hypervisors. A *type I hypervisor* (native hypervisor, bare-metal hypervisor) runs directly on the bare hardware. A *type II hypervisor* (hosted hypervisor) runs on top of an operating system, the host OS. The low-level functionality of the host OS can be used to control the hardware. However, the scheduling of the hypervisor is dependent on the scheduling of the host OS.

2.2.2 Processor Virtualization

Modern processor architectures feature two different execution modes. In the *problem mode* for application tasks, also known as user mode, only a subset of the instruction set can be executed. In the *supervisor mode* for system software, also known as kernel mode or privileged mode, the entire instruction set can be executed. The subset of the instruction set that is only available in supervisor mode is called the set of *privileged instructions* and it provides full access to processor state and functionality. [Stallings, 2005] Robust virtualization requires that the hypervisor retains control of the hardware, so only the hypervisor can be executed in supervisor mode.

The guest systems have to be executed entirely in problem mode, including guest operating system. [Smith and Nair, 2005a]

When executed in problem mode, a privileged instruction causes a *trap*, a synchronous interrupt that results in a switch to supervisor mode, wherein the system software handles the privileged instruction. An example for a privileged instruction from the Power ISA is `mtmsr` (move to machine state register) [IBM, 2010].

An instruction is *sensitive*, if it either affects the state of the hardware or if its execution directly depends on the processor mode or the memory location where it is executed. An example for a sensitive instruction is again `mtmsr`, since it directly affects the machine state register. Popek and Goldberg defined privileged and sensitive instructions in a formal manner and stated the following fundamental theorem:

Theorem 2.2.1. ***(Trap-and-emulate) Virtualizable Instruction Set Architecture.*** *For any conventional third generation computer* [2]*, 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 [Popek and Goldberg, 1974b].*

Figure 2.5 depicts the condition of this theorem. On the right, the ISA is virtualizable according to their theorem. The ISA on the left is not. Non-virtualizable are instructions that are sensitive, but non-privileged: executed by a guest system, they can manipulate the hardware state without being noticed by the hypervisor, since they do not trap when executed in problem mode. An example for such an ISA is x86 [Robin and Irvine, 2000]: the instruction `popf` (pop flags) may change processor flags (e.g., IF, which controls interrupt delivery), but executed in user modes does not cause a trap [Adams and Agesen, 2006].

If Theorem 2.2.1 is fulfilled, as it is for example the case for the Power ISA, *full virtualization* is possible: unmodified guests (compared to the direct execution on hardware) can be executed on top of the hypervisor. The hypervisor is able to intercept and emulate all sensitive instructions. This has the significant advantage that the guest operating system does not have to be aware of whether being executed on bare hardware or by a hypervisor. The virtualization is transparent to it.

But system virtualization can be applied as well if the condition of Popek and Goldberg's theorem is not fulfilled. It is actually not a theorem for virtualizability in general, but only for the ability to virtualize a system by trap-and-emulate. Binary translation, paravirtualization, and hardware virtualization assistance all make non-virtualizable (i.e., non-privileged and sensitive) instructions trap when executed in

[2]Popek and Goldberg address a computer with a processor with two modes of operation and linear, uniformly addressable memory.

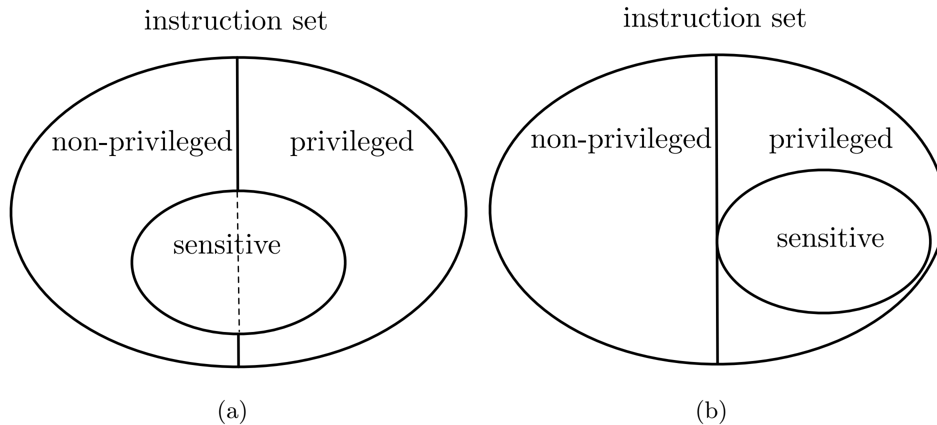


Figure 2.5: Popek and Goldberg's requirement for a virtualizable instruction set architecture: is the set of sensitive instructions a subset of the set of privileged instructions? (a) not fulfilled; (b) fulfilled

user mode.

Binary translation implements the emulation of an instruction set (the source) by another one (the target) by translating executable binary code (sequences of machine code instructions) [Sites et al., 1993]. VMware applied dynamic binary translation for x86 virtualization [Adams and Agesen, 2006]. Source instruction set is the given x86 ISA, translated to a target ISA that does not have x86's obstacles to virtualization. The guest is executed by an interpreter instead of directly by a physical CPU and the interpreter correctly intercepts and implements sensitive and non-trapping instructions like `popf`. The translation replaces these instructions by a sequence of instructions with the intended effect.

Hardware-assisted virtualization enables full virtualization by explicit virtualization support from the processor (e.g., Intel Virtualization Technology [Uhlig et al., 2005]). Architectural extensions include for example a third privileged mode, resulting in different modes for hypervisor (host mode), guest operating system (guest mode), and guest application tasks (user mode). The new data structure virtual machine control block (VMCB) stores control state and the state of a virtual CPU. The hypervisor specifies within the VMCB under which conditions the guest system is interrupted in order to execute the hypervisor. For example, each execution of the sensitive and non-trapping instruction `popf` could cause a context switch from guest to host mode. This overcomes x86's virtualization obstacle, but results in a large overhead. The number of context switches can be reduced by maintaining a guest-specific shadow of the register and including it in the VMCB [Adams and Age-

sen, 2006]. Hardware-assisted virtualization has a great potential to provide efficient virtualization for embedded systems. However, it does not eliminate the challenge to guarantee that the real-time requirements of guest systems are met, which is of major importance for this thesis.

Paravirtualization refers to the porting of the guest OS to an interface provided by the hypervisor, the paravirtualization application programming interface (API) and application binary interface (ABI) [Whitaker et al., 2002, Barham et al., 2003]. The interface is similar but not identical to the ISA. The guest OS is aware of being executed by a hypervisor and uses *hypercalls* to request hypervisor services (e.g., for memory management or inter-VM communication). A hypercall is a trap from guest OS to the hypervisor, just as a system call is a trap from an application task to the OS. The source code of an OS is modified in order to paravirtualize it and sensitive and non-trapping instructions can be replaced by hypercalls to virtualize an ISA that is not virtualizable by trap-and-emulate.

The major drawback compared to binary translation and hardware-assisted virtualization is the lack of transparency. Only ported operating systems can be run. If legal or technical issues preclude the modification of an OS, it is not possible to host it. A specific advantage of paravirtualization for real-time systems is the possibility to schedule in a cooperative manner and according to dynamic policies, which in general requires a passing of scheduling information from guest OS to hypervisor. A fully virtualized OS cannot cooperate with the hypervisor, since it does not even know that there is a hypervisor running below it.

2.2.3 I/O Virtualization

A big challenge for the hypervisor-based integration of multiple software systems is the management of Input/Output (I/O) devices across VMs [Moyer, 2013]. One solution is to avoid the sharing of I/O devices, which is of course only possible if the target hardware features as many devices as there are I/O demanding guest systems. In this case, each guest gets exclusive access to a dedicated I/O device, which therefore does not have to be virtualized (but might be). The hypervisor ensures that only the intended guest can access it (bypassing the hypervisor is usually not possible anyway, since I/O operations require the privileged mode, see below). The guests can use their own device drivers. If the guest does not run continuously but is scheduled, incoming data has to be buffered during periods of time in which the guest is not executed. Interrupt-driven I/O (the I/O controller uses interrupts to inform the system software) requires that the hypervisor can queue up and inject

interrupts into the guest operating system [Smith and Nair, 2005a].

There are different approaches to realize a device sharing. In case of *emulation*, the hypervisor intercepts all accesses to I/O devices and handles them in a manner that realizes a multiplexed sharing (software multiplexing). Interceptions is possible since I/O instructions are usually privileged and in case of memory-mapped I/O (specific region of the physical memory is used for accessing I/O devices), memory addresses used for I/O are usually not accessible in problem mode [Smith and Nair, 2005a]. The guests use virtual devices with an own state, which are presented by the hypervisor. The hypervisor maintains the states of the virtual devices and the guests' operations on it are translated by the hypervisor to operations on the physical device [Smith and Nair, 2005a]. The guests do not have direct access to the physical device and therefore cannot corrupt it. Drawbacks are the overhead on all I/O accesses and the integration of the device driver into the hypervisor (not necessarily in a monolithic design, preferably in user space) [Moyer, 2013]. *Paravirtualized device drivers* that cooperate directly with the hypervisor and are aware of sharing the device can support software multiplexing [Fisher-Ogden, 2006].

Pass-through refers to direct access of the guest OS to the device with its unmodified device drivers, resulting in higher efficiency, but lower robustness. Isolation of the guests is no longer given, since a guest might affect the execution of other guests by improper device accesses. DMA-managed I/O (Direct Memory Access) enables the I/O controller to access the memory directly. An input/output memory management unit (IOMMU) supports the pass-through approach by constraining direct memory accesses. This obviates the need for memory accesses by the guests beyond their allocated memory, which would otherwise be necessary for a guest OS to use DMA [Moyer, 2013].

Hardware multiplexing implements virtualization functionality into the I/O device, enabling it to provide independent I/O resources, e.g., multiple hardware queue and packet buffer rings [Moyer, 2013].

2.2.4 Virtualization for Mixed-Criticality Systems

For a system with virtualization that requires certification of functional safety, the hypervisor has to be certified according to the highest criticality level of the hosted guest systems. This is for example stated by the European Aviation Safety Agency's Certification Memorandum, which provides guidance for compliance demonstration with current standards on specific certification issues, in this case integrated software architectures:

> In cases where there are multiple software levels within a given system and/or component, the protection and associated mechanisms between the different software levels (such as partitioning, safety monitoring, or watchdog timers) should be verified to meet the objectives of the highest level of software associated with the system component. [European Aviation Safety Agency, 2012]

The hypervisor is essential for the correct execution of the guest systems and due to this strong influence on safety inherits the criticality level of the guest. One can derive the requirement for the hypervisor design to be *thin*, i.e., include only mandatory functionality in order to ease certification.

The certification requires a "thorough protection/partitioning analysis" [European Aviation Safety Agency, 2012] and a safe and deterministic partitioning [Littlefield-Lawwill and Kinnan, 2008]. Resource isolation is the key mechanism to enable the integration of guest systems of differing criticality levels on the same platform. A violation of safety requirements arises, if the incorrect behavior of one of the guest systems corrupts the behavior of another guest. In order to certify a system with software partitioning such as a hypervisor, *freedom from interference* between partitions has to be shown, especially between guest systems of differing criticality level. The functional safety standard ISO 26262 for automotive systems defines interference as follows (ISO 26262-9:6):

> the presence of cascading failures from a sub-element with no ASIL assigned, or a lower ASIL assigned, to a sub-element with a higher ASIL assigned leading to the violation of a safety requirement of the element. [ISO, 2011]

It is therefore a major requirement for a hypervisor for mixed-criticality systems to achieve freedom from interference. Otherwise, all hosted software had to be certified according to the highest criticality level of any software component. ISO 26262 differentiates between three types of freedom from interference (ISO 26262, Part 6, 7.4.11):

- spatial freedom from interference,
- temporal freedom from interference,
- exchange of information. [ISO, 2011]

Spatial freedom from interference refers to the protection of the resource memory. The integrity of the address space of each guest system and the hypervisor itself

must be ensured. A unique address space must be statically allocated to each VM, not accessible by other VMs, and the guest system must operate entirely in it. By consequence, a guest system cannot change the software or data of another guest system and cannot command the private devices or actuators of other VMs [Rushby, 1999]. The damage of a faulty or malicious guest system is restricted to its own behavior and data. Alike, the integrity of the hypervisor's address space must be ensured, so that guest systems cannot affect it.

Spatial isolation is typically enforced by a hardware component that performs all memory references and prevents illegal memory accesses, either a Memory Protection Unit (MPU) that enables the hypervisor to partition memory into regions with defined access permissions or a Memory Management Unit (MMU), which provides in addition a translation of virtual to physical memory addresses [Tanenbaum and Goodman, 1998]. With the help of a MMU, virtual memory spaces that are completely isolated from each other are created and assigned to the VMs.

Temporal freedom from interference refers to the protection of the resource CPU, i.e., each VM receives the demanded computation time service regarding duration, rate, and maximum blackout time (time without service) [Rushby, 1999]. The hypervisor must ensure that each guest system is executed in compliance with its real-time constraints, independent from the other guests. When a guest system overruns its computation time, the correct execution of the other guest must nevertheless be assured, i.e., a guest cannot prevent another guest from completing its tasks by depriving it of computing time (time starvation).

The third aspect demanded by ISO 26262 covers the *exchange of information*: safety related shared data and signals have to be protected.

A robust partitioning must include as well that resources that are shared between VMs (or a VM and the hypervisor) are indeed shared in a safe manner, with the result that all guest systems receive access as demanded to fulfill the functional and temporal requirements [Rushby, 1999]. For example, if multiple guest systems share an I/O device, a guest must not be able to prevent another guest from using it, e.g., by a denial-of-service attack or a reconfiguration. The hypervisor has to prevent deadlocks, which occur when two or more guest systems are waiting for each other, since a requested resource is held by another guest, which in turn is waiting for another resource (multiple guest systems are waiting for each other to complete and cannot make any progress) [Stallings, 2005].

2.3 Multicore Processors

The Central Processing Unit (CPU) is the component of a computer system that executes program instructions. After fetching the instructions from main memory, the CPU examines and executes them. A *multicore processor* is a processor with two or more CPUs, typically integrated in a single integrated circuit (chip multiprocessor). These so-called cores execute instructions in parallel, independent from each other (processor-level parallelism). Different cores can execute different instructions on different parts of the memory at the same time (multiple instructions, multiple data), in contrast to array processors that perform the same sequence of instructions on multiple instances of data. The ISA is in general the same as for single-core processors, except of modifications to support parallelism, since this enables the reuse of existing software and development tools. [Tanenbaum and Goodman, 1998] Homogeneous multicore processors feature identical cores (same ISA and frequency). Heterogeneous architectures combine different processing elements, for example a digital signal processor and a general purpose processor [Catanzaro, 1994].

It becomes more and more complicated for processor designers to reach the demanded performance growth by increasing the frequency of single-core processors, since the associated growths in power consumption and heat dissipation become unacceptable [Keckler et al., 2009]. Multicore processors address the power issue and achieve a performance growth by parallelism, i.e., increasing the number of processor cores, instead of increasing the frequency of a single core. The execution of multiple cores at lower frequency results in an increase of the performance in terms of instructions per second with reduced power consumption.

The cores share main memory and peripheral devices. The main memory is shared uniformly: the latency of an access to a specific memory location is the same for all cores. A problem of shared-memory multiprocessing is memory bus contention when the cores try to access the memory over the same bus. The resulting performance degradation can be reduced by caches, significantly faster but smaller memories that buffer data and instructions between CPU and memory [Tanenbaum and Goodman, 1998]. A multi-level cache hierarchy may be present and caches can be core-exclusive or shared. All components (cores, main memory, caches, I/O devices) are connected by a bus. Figure 2.6 shows such a bus-based shared memory multicore processor with per-core private caches.

Private caches raise the challenge of *cache coherence*. If copies of the same memory block are stored in multiple caches, problems may arise with inconsistent shared data. If a core modifies the memory block in main memory, the other cores continue

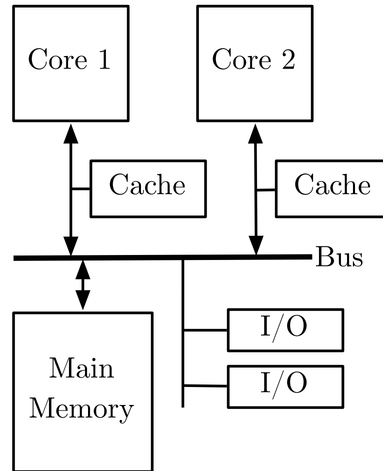


Figure 2.6: Single-bus shared memory multicore with private caches

to access the no longer valid copy in their caches [Culler et al., 1999]. Common mechanisms to ensure coherency are directory-based and snooping protocols. In the first case, a directory stores the information which data is being shared between caches and filters all requests of the cores to load or update memory in their caches. When an entry is modified, the directory either updates or invalidates the copies in other caches [Moyer, 2013]. In case of snooping protocols, the caches monitor a shared bus for accesses from other caches to memory locations of which they have copies. When a write operation is observed to such a location, the cache controller invalidates its own copy. Another possibility is to broadcast all updates to shared data and update the affected caches [Culler et al., 1999, Moyer, 2013].

According to Symmetric Multiprocessing (SMP), a single OS controls the software execution on all cores. All cores share code and data of the OS and execute both the OS (potentially simultaneously) and application tasks. Asymmetric Multiprocessing (AMP) uses a separate OS instance on each core. Hypervisor-based system virtualization has to be considered as a third approach. The hypervisor itself controls the software execution on all cores in a SMP manner and its code and data are shared among multiple cores. But it manages the execution of different operating systems, which operate independently on different cores as it is the case for AMP [Moyer, 2013]. Figure 2.7 illustrates these different approaches to use a multicore processor.

The hypervisor might assign each VM to a single core and make the multicore processor look like a single core to the guest or it might support multicore operating systems by enabling an execution on multiple cores in parallel. Moreover, VMs might be pinned to a certain core (full core affinity) or be scheduled by the hypervisor among

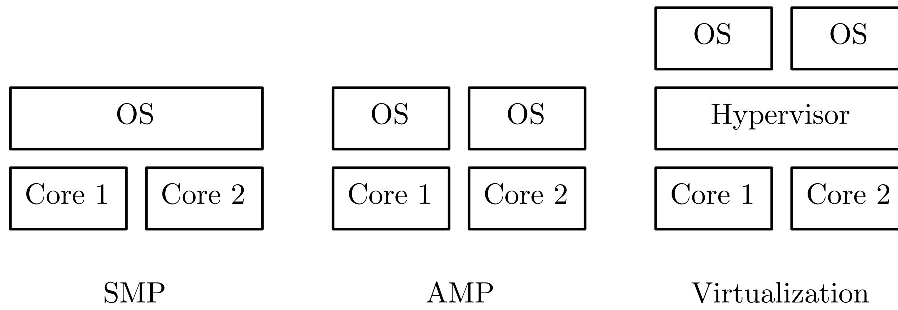


Figure 2.7: System software's core management: symmetric multiprocessing, asymmetric multiprocessing, and hypervisor-based virtualization

the cores at runtime (no affinity).

2.3.1 Multicore Scheduling

Scheduling was defined so far only for uniprocessors (see Definition 4 in Section 2.1.2). As just introduced, a processor might feature multiple cores, each a full CPU. A multicore scheduler has to take an additional decision, the so-called allocation problem: not only which task to execute at any point in time, but also on which core. The first work on multiprocessor real-time scheduling dates back to the late 1960s, when Liu exposed the complexity of the problem [Davis and Burns, 2010]:

> Few of the results obtained for a single processor generalize directly to the multiple processor case; bringing in additional processors adds a new dimension to the scheduling problem. The simple fact that a task can use only one processor even when several processors are free at the same time adds a surprising amount of difficulty to the scheduling of multiple processors. [Liu, 1969]

Multiprocessor scheduling algorithms can be classified according to when the allocation is made (migration based classification [Carpenter et al., 2004]). In *partitioned scheduling*, each task is allocated to a specific processor core and executed only on this core (no migration). The scheduler maintains per core a separate ready queue. Scheduling algorithms where migration is permitted are referred to as *global*. They use a single ready queue and do not require that all jobs of a task execute on the same core. A further differentiation of global scheduling is based on whether migration is only possible at job boundaries [Carpenter et al., 2004, Davis and Burns, 2010]. In case of task-level migration, different jobs of a task can be executed on different

cores, but each job is executed on a single core. Job-level migration permits the preemptive execution of a single job on different processors (but no parallel execution of a job) [Davis and Burns, 2010].

Global scheduling has the following advantages [Davis and Burns, 2010]:

- in many cases fewer preemptions (only required if no core idles) [Andersson and Johnsson, 2000],
- spare bandwidth (when a task does not need its WCET) can potentially be used by all other tasks,
- more appropriate for open systems that permit adding tasks at runtime.

Partitioned scheduling has the following advantages [Davis and Burns, 2010]:

- reduction of the multiprocessor scheduling problem to a set of less complex uniprocessor scheduling problems [Carpenter et al., 2004],
- no migration overhead (e.g., communication load for transfer of the context of job/task, additional cache misses),
- overrun of the WCET by a task affects only the tasks on the same core,
- no excessive overhead of maintenance of a single global ready queue for large systems (scalability).

Partitioned scheduling can reuse well-known uniprocessor scheduling results, e.g., schedule the tasks that are allocated to the same core by RM or EDF; whereas using these optimal uniprocessor scheduling algorithms in a global multiprocessor manner may result in arbitrarily low utilization (the so-called "Dhall effect") [Dhall and Liu, 1978]. The main disadvantage of partitioned scheduling is the complexity of the allocation problem. Finding an optimal allocation of tasks to cores is analogous to bin packing and therefore known to be NP-Hard [Garey and Johnson, 1979]. By consequence, non-optimal heuristic partitioning algorithms are usually applied [Carpenter et al., 2004]. A second disadvantage: there are task sets that are schedulable if and only if migration is permitted [Carpenter et al., 2004].

See Davis and Burns for an extensive survey of both partitioned and global multiprocessor scheduling algorithms [Davis and Burns, 2010].

2.3.2 Multicore and Predictability

Real-time systems require information about the worst-case execution times in order to guarantee a deterministic behavior. The certification of functional safety depends on the ability to determine the system's exact timing behavior and requires to show that the system reaches a safe state within a specified time interval after a hazard.

This is achieved by analyzing the ECU activities and their timing in an end-to-end event chain [Stappert et al., 2010].

Timing of software is highly dependent on the underlying hardware. A precise determination of the WCETs of single-core architectures is already challenging, since processor features such as caches, pipelines, branch prediction, or co-processors evolved in order to maximize the average performance and complicate the timing analysis. But multiple established methods and tools exist. [Wilhelm et al., 2008]

Multicore processors are significantly more difficult to analyze as the sharing of on-chip resources between cores introduces complex timing effects at the machine instruction level or even nondeterminism. Independently executed software permanently competes for accessing shared architectural elements such as:

- system bus,
- memory bus,
- memory controller,
- non-core-private caches,
- DMA controller,
- interrupt controller,
- I/O controller. [Kotaba et al., 2013]

Two issues arise for the determination of WCETs due to shared resource contention: nondeterminism and pessimism. Resource accesses might be arbitrated by certain units in a non-explicit manner, introducing nondeterministic delays and making it impossible to determine the WCET [Kotaba et al., 2013]. Or the complexity of the on-chip dependencies results in very long possible delays for certain operations, and by consequence in extremely pessimistic WCETs, which potentially are no more economically acceptable.

An example is system bus contention: a static worst-case analysis might have to expect that each access is delayed by simultaneous accesses of all other cores plus asynchronous accesses by DMA controllers, as they autonomously access the shared bus, plus potentially asynchronous accesses of additional hardware units. The severeness of these issues is emphasized by the common solution of the avionics domain for the contention on the system bus: all but one core are actually disabled and any asynchronous DMA or I/O traffic is avoided [Kotaba et al., 2013].

Very challenging are shared caches due to the overhead to keep coherency and due to unpredictable inter-core interactions (a cache miss on one core can heavily impact the performance on the other cores in both directions, increasing or decreasing) [Paun et al., 2013]. Modeling the behavior of shared caches is practically impossible [Schoe-

berl, 2009]. Solutions are the disabling of shared caches or a static cache partitioning (each cores gets dedicated cache lines) [Vera et al., 2003, Lin et al., 2009].

Other solutions of the contention problem include capabilities for parallel service of hardware units. Take the example of a memory controller and interleaved accesses from different cores: if the controller is able to handle at the same time at least as many open pages as there are cores, the delay of continuously opening and closing pages is prevented. Another example is the parallelization of memory access by multiple memory banks (partitioning of banks among cores supports in addition isolation [Liu et al., 2012, Yun et al., 2014]). System bus contention can be reduced by separate interconnects for cache coherence protocols. [Kotaba et al., 2013]

Deterministic arbitration policies (and complete information about the exact behavior) are required for all processor elements, e.g., based on FIFO (first in, first out: accesses are served in the order of occurence), TDMA (time division multiple access: the cores get different time slots in which their accesses are served) [Rosen et al., 2007], or static priorities. See for example Paolieri et al. for a predictable round-robin memory bus arbiter [Paolieri et al., 2009]. Nowotsch et al. [Nowotsch and Paulitsch, 2012] and Dasari et al. [Dasari et al., 2011] showed how to bound the impact of system bus contention. Negrean et al. presented a method to analyze the worst-case delay due to accesses to shared bus and memory [Negrean et al., 2009].

2.4 Virtual Machine Scheduling

2.4.1 Hierarchical Scheduling

System virtualization implies an additional scheduling level, the scheduling of the virtual machines, which is required if the number of VMs is greater than the number of processor cores. Two different scheduling decisions have to be made: VM scheduling and task scheduling.

The VM scheduling is responsibility of the hypervisor, as it controls the execution of the guest systems and the resource management. The additional scheduling level is therefore implemented in a different software layer as the task scheduling, which is still performed by the guest operating systems. Scheduling techniques with decisions on different levels are called *hierarchical scheduling*, in this case: the hypervisor schedules VMs, the hosted guest OS of the selected VM schedules the guest's applications tasks (Figure 2.8).

Hierarchical scheduling is a direct consequence of the coarse-grained integration of system virtualization. Goal is the integration of existing software stacks including

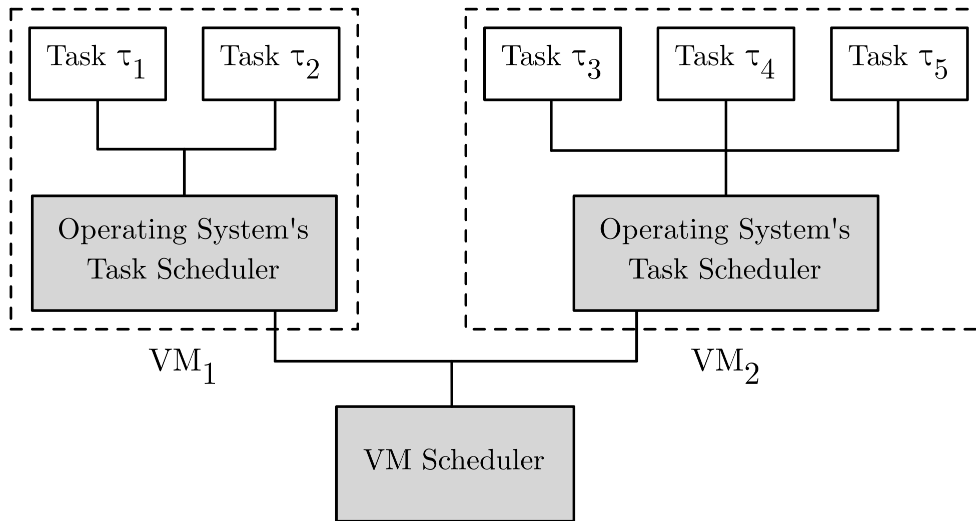


Figure 2.8: System virtualization's hierarchical scheduling: virtual machine scheduling by hypervisor and task scheduling by operating systems

OS with verified (or even certified) characteristics and it is not desired to split these systems up, especially in case of mixed-criticality systems. The intended reuse of software and certification results requires isolation at guest system (i.e., VM) level. This is irreconcilable with a merging of the different task sets and a scheduling based on a global task ready queue. Instead, the guest operating systems should schedule their task set according to their specific task scheduling policy.

This separation of scheduling concerns is an abstraction for both system software entities, hypervisor and operating system. The hypervisor is not responsible for task scheduling, the guest OS does not need any information about the task scheduling of other guests. The hypervisor schedules virtual machines based on a system-level global policy, the guest operating system's task scheduling is based on guest-local information. The guest systems can be developed and analyzed independently from each other. The hypervisor needs information about the guests' timing requirements, otherwise real-time guarantees are impossible, but no global task schedulability analysis is required.

2.4.2 Virtual Processor and Virtual Time

A *virtual processor* is a representation of the physical processor to a virtual machine. A dedicated virtual processor is created and managed by the hypervisor for each VM. It is in general slower than the physical processor to allow a mapping of multiple virtual processors onto a single physical processor (if of equal speed, no abstraction

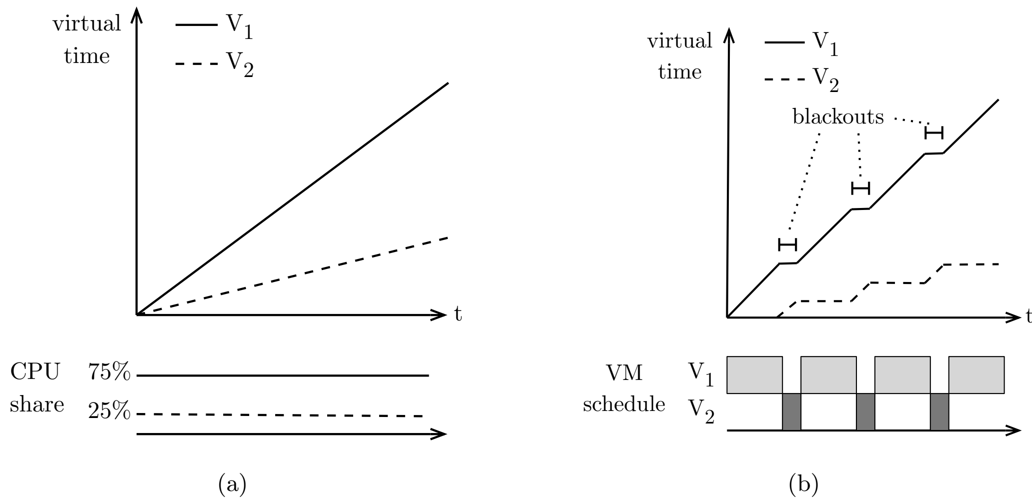


Figure 2.9: Progress of virtual machines: (a) ideal; (b) in practice (cf. [Kaiser, 2008])

to a virtual processor is necessary).

Virtualization for real-time systems requires a deterministic mapping between real world time and virtual time. Only a single virtual machine is active on single-core processors or a specific core of a multicore processor at any time and the other virtual machines experience a blackout. From the point of view of a virtual machine, time only advances while the VM is active. However, the behavior of the guest system and its scheduling is defined with respect to the real world time, derived directly from the environment in which the system operates.

The virtual time t_k^{virt} of a guest system that is executed by a virtual processor of speed v_k is incremented when the corresponding virtual machine is executed [Lipari and Baruah, 2001]:

$$\frac{d}{dt} t_k^{virt} \stackrel{def}{=} \begin{cases} \frac{1}{v_k} & \text{, if virtual machine } V_k \text{ is executed,} \\ 0 & \text{, otherwise.} \end{cases}$$

Figure 2.9 depicts this mapping of virtual time to real world time for an example with two VMs and a single core. The left side shows the mapping of an ideal virtualized system, with different virtual processor speeds, but continuous progress of both VMs. This cannot be achieved in practice, since a processor core has to be shared in a time-divison multiplexing manner, and has to be approximated. The right side shows an exemplary VM schedule and the resulting blackouts, in which the virtual time for the at this time not scheduled VM does not progress. [Kaiser, 2008]

Event-driven real-time systems have to react within specified time spans to external events. If the event occurs during a blackout of the handling VM, the reaction

is delayed until the VM is scheduled again. Compared to a non-virtualized system, the reaction time is increased by the VM's worst-case blackout time. The virtualization of event-driven real-time systems requires therefore that the maximum blackout times are deterministic and small enough to meet the guest systems' reactivity requirements.

2.4.3 Classification and Common Solutions

This section classifies virtual machine scheduling based on different characteristics and gives an overview of common approaches.

Preemptive vs. Non-preemptive. Preemptive VM schedulers are able to suspend the execution of a guest system. A non-preemptive scheduler always lets a guest run until the guest itself suspends, which requires cooperation (to avoid starvation of other guests) and paravirtualization (to suspend) if multiple VMs are executed on the same core. A faulty or malicious guest can starve all other guests.

Single-core vs. Multicore. Single-core VM scheduling assigns a single VM out of the set of hosted VMs at any time to a single available processor. Multicore scheduling manages the execution of a set of VMs on a set of processor cores and decides not only which VM to execute at any point in time, but also on which core (allocation). The allocation can be performed statically (partitioned scheduling: each VM is executed only on a single core) or dynamically (global scheduling: VMs migrate between cores). A hypervisor might support multicore VMs (e.g., allocate two cores to a VM), otherwise, each guest is executed on at most one core at all times. In the first case, the number of allocated cores might be static or dynamic.

Cooperative vs. Non-communicating. The coaction of VM scheduling and task scheduling can be cooperative and make use of explicit communication. The guest OS might pass scheduling related information to the hypervisor and the hypervisor might inform the guest OS about VM scheduling decisions, e.g., the amount and characteristic of the allocated bandwidth. Cooperative scheduling requires paravirtualization, since a guest OS that is unaware of being executed by a hypervisor cannot communicate with it. In case of a non-communicating scheduling architecture, there is no exchange of scheduling related information at runtime.

In the following, common VM scheduling approaches are discussed, namely:

(I) dedicated processor core for each VM (non-preemptive, statically partitioned multicore, non-communicating),

(II) static precedence of a single real-time VM per core (non-preemptive, single-core or statically partitioned multicore, cooperative),

(III) static cyclic schedule with fixed execution time slices (preemptive, statically partitioned multicore, non-communicating),

(IV) execution-time servers (preemptive, single-core or partitioned multicore or global multicore, cooperative or non-communicating).

(I) Dedicated processor core for each VM. Each VM is executed on a dedicated core. This one-to-one mapping of VMs to cores obviates the need for a VM scheduler, since each guest system is executed at all times (non-preemptive). This guarantees that the timing constraints of real-time guests are met (no blackouts). The approach can be applied only to multicore processors, with the required number of cores equal to the number of guests.

(II) Precedence of a single real-time VM per core. A single guest with real-time requirements is executed on each core. In contrast to the first solution, it might share the core with non-real-time VMs, which are executed in background (the scheduler uses the idle time of the real-time guest to run them). The timing constraints are guaranteed since the real-time guest is executed whenever it has a computation demand. If the core is shared, paravirtualization is required for the real-time VM in order to inform the hypervisor about idle intervals. The real-time VM is not preempted by the hypervisor, non-real-time VMs are (they might be scheduled in a round robin manner). This approach requires a multicore processor if there is more than one real-time guest. Non-real-time VMs might be partitioned or migrate among the cores.

(III) Static cyclic schedule. Static cyclic scheduling [Baker and Shaw, 1989] is based on an offline time-division multiplexing schedule, which assigns execution time windows within a repetitive cycle to the VMs [Sha, 2004, Kerstan, 2011]. Simple solutions apply a weighted round-robin approach: time slices with the length determined by the product of utilization and cycle length are assigned to the VMs in circular order. In order to implement a multi-rate cyclic scheduling, a major cycle (the repetitive cycle, equal to the least common multiple of all task periods) is divided into minor cycles (time intervals of fixed lengths, often equal to the greatest common divisor of all task periods) and VMs are statically allocated to these minor cycles based on their required utilization and execution frequency, which have to be derived from the guests' task sets. This time-driven static scheduling approach is for example part of the software specification

ARINC 653 [3] for avionics systems. The schedule is stored in a dispatching table and enforced by the hypervisor at runtime, including preemption.

(IV) Execution time servers. The scheduling is based on servers, a technique to schedule hybrid task sets of periodic and aperiodic tasks, as introduced in Section 2.1.2. Each VM is executed by a dedicated server, which implements a virtual processor by providing a limited computation bandwidth. Period and bandwidth of the server have to be dimensioned based on the computation requirements of the associated guest system. Servers are periodic tasks and, therefore, there are multiple options to schedule them, according to static or dynamic priorities [Buttazzo, 2004], partitioned or global. The hypervisor might preempt a running server at each point in time. As server algorithm, different solutions are possible, e.g., periodic servers [Groesbrink et al., 2014a], constant bandwidth servers [Cucinotta et al., 2011a], work-conserving periodic servers [Lee et al., 2011], or L^4Linux servers [Yang et al., 2011]. A static server-based scheduling does not require cooperation of the guest OS.

[3] ARINC 653 (Avionics Application Standard Software Interface) is a software specification for space and time partitioning in safety-critical avionics real-time operating systems. [Prisaznuk, 2008a]

2.5 Summary

Embedded systems are computer control systems that operate with a dedicated function as an integral part of a larger technical system. Many embedded systems have strict response time constraints, derived from the system's environment. Scheduling refers to the management of the resource processor in a time-multiplexing manner. In case of a real-time system, the major objective of scheduling is to meet the timing requirements, which demands a predictable system behavior.

The criticality of a component refers to the severity of failure. Human lives depend on the correct operation of safety-critical systems. Criticality is therefore directly related to functional safety and the certification according to safety integrity levels. The criticality level defines safety requirements: the higher the required confidence in the component, the more pessimistic is in general the obtained worst-case execution time. Integrated modular systems such as system virtualization lead in many cases to the co-existence of multiple criticality levels.

Multicore processors are characterized by multiple CPUs and a shared main memory. The permanent competition for shared architectural elements such as buses or the memory controller by the software that independently executes on different cores complicates the determination of worst-case execution times significantly.

System virtualization provides platform replication and enables the concurrent execution of multiple software stacks (operating system and application tasks). The hypervisor provides an abstraction of the real machine (virtual machine). It manages the hardware resources and controls the execution of the virtual machines, including their scheduling. Paravirtualization requires a porting of a guest operating system. A paravirtualized guest operating system accesses services provided by the hypervisor via hypercalls (virtualization awareness).

The hypervisor-based integration of safety-critical systems demands freedom from interference between the hosted guest systems. This includes spatial (memory) and temporal isolation (CPU) as well as a safe exchange of information and sharing of other resources. Hierarchical scheduling refers to the fact that system virtualization implies scheduling decisions on two levels. The virtual machines are scheduled by the hypervisor and each guest operating system schedules its tasks. Common virtual machine scheduling techniques are based on a dedicated core for real-time guest systems, static cyclic schedules, or execution time servers.

Chapter 3

A Multicore Hypervisor for Embedded Real-Time Systems

Contents

3.1 Problem Statement **46**

3.2 Related Work **48**

3.3 Proteus Multicore Hypervisor **50**

3.3.1 Architecture 51

3.3.2 Configurability 52

3.3.3 Processor Virtualization 55

3.3.4 Paravirtualization Interface 56

3.3.5 Multicore 57

3.3.6 Memory Virtualization 59

3.3.7 Virtualization of Timer and I/O Devices 59

3.4 Evaluation **60**

3.4.1 Evaluation Platform: IBM PowerPC 405 60

3.4.2 Execution Times 61

3.4.3 Memory Footprint 65

3.5 Summary **66**

3.1 Problem Statement

As introduced in the previous chapter, the hypervisor is the software layer that creates an abstraction of the underlying physical machine in order to provide the operating system with a virtual machine. It separates operating system and hardware and divides the system resources (CPU, memory, I/O devices) into multiple virtual execution environments in order to share the hardware among multiple software stacks of operating system and application tasks. The hypervisor manages the execution of these virtual machines.

Hypervisor technology is state of the art for server architectures for many years [Smith and Nair, 2005c], but this software cannot be applied to embedded systems due to the fundamentally differing requirements and design goals. Most importantly, server technology prioritizes the average throughput over predictable timing, resulting in completely different scheduling strategies. A hypervisor for embedded real-time systems has to meet the following requirements, which can be derived from the use cases in Chapter 1:

Real-time Capability. The hypervisor has to provide deterministic behavior for all of its services and operations as well as bounded interrupt latencies. Each guest system has to be executed in a predictable way. The virtual machine scheduling must guarantee that all guest systems are able to meet their timing requirements.

Safe and Secure Partitioning. The hypervisor must be in complete control of the system resources. It must be protected from the guest systems; virtual machines must be isolated from each other. Spatial isolation refers to the integrity of a VM's address space: a VM must operate completely in a unique and statically allocated address space, not accessible by other VMs. Alike, the integrity of the address space of the hypervisor must be guaranteed. The damage that a faulty or malicious guest system can do, must be restricted to its own data. Temporal isolation refers (in addition to an appropriate hierarchical scheduling) to the possibility to validate the timing requirements of a guest system independently from other guests, and the containment of execution time overruns: if a guest system overruns its computation time, this must under no circumstances provoke that other guests miss a deadline.

Secure Inter-VM Communication. The system architect must be able to explicitly relax the strong partitioning in order to allow communication between guest systems. Communication between guests is mandatory, if one consolidates formerly physically distributed systems that cooperate. The hypervisor must authorize and control the communication channel, so that it is protected against access from unauthorized VMs.

Multicore Support. Multicore processors are increasingly popular for embedded systems in order to provide the required computational power. In fact, multicore processors are a major driver for virtualization, whose architectural abstraction eases the migration of single-core software to multicore platforms and supports the efficient resource usage.

Scalability. To increase the applicability, the number of VMs that can be hosted should be limited only by the system resources (memory and computational power). In particular, the hypervisor should not be restricted to a dual OS configuration and it should be able to host more VMs than processor cores available. In addition, it should be designed considering scalability regarding the number of processor cores.

Efficiency. The overhead of a guest system (execution time and latencies) should be low compared to the native performance of an unvirtualized execution.

Small Memory Footprint. The hypervisor must use memory very efficiently. Embedded systems are often memory-constrained, especially with respect to on-chip memory. Security-critical code such as a hypervisor must be contained in on-chip memory [Heiser, 2009]. The hypervisor has to be executed at the processor's highest privilege level and is part of the system's trusted computing base (TCB, components that are critical to the security of the entire system [Rushby, 1981]). A small hypervisor helps to minimize the size of the TCB, which is important due to the implications on the costs of the quality assurance process and the trustworthiness of the system. The smaller the TCB, the easier its validation or even verification, the more secure and reliable the expected result.

Configurability. Embedded systems are dedicated to a particular functionality. The hypervisor should be configurable based on the application-specific requirements, so that only needed functionality is included. This supports the reduction of the memory footprint.

Support of Paravirtualization and Full Virtualization. Paravirtualization can often be exploited to reduce the overhead of virtualization and ease the sharing of I/O devices. Moreover, it is required by all hierarchical scheduling techniques that are based on an explicit cooperation of hypervisor and OS. A hypervisor should therefore provide an extensible paravirtualization interface. However, paravirtualization's applicability is limited, since it might not be possible to paravirtualize an OS (i.e., port it to the hypervisor's interface) for technical or legal reasons. Therefore, a hypervisor should support the execution of non-modifiable operating systems by full virtualization. The concurrent hosting of both paravirtualized and fully virtualized guests should be possible, since a convenient approach is to host a paravirtualized

RTOS and a fully virtualized General Purpose Operating System (GPOS).

Support of OS-less Guests. Hypervisor-based virtualization is by its nature coarse-grained: entire software stacks including operating system are integrated. There are embedded software applications that do not need an operating system. In order to reduce the overhead regarding latencies and memory footprint, a hypervisor should provide the possibility to host native tasks within a VM, without OS.

After a look at related work, this chapter presents the multicore hypervisor *Proteus* for PowerPC 405 processors, which meets these requirements. It is the evaluation platform for the scheduling technique of Chapter 6 and the migration approach of Chapter 7.

3.2 Related Work

Gu and Zhao [Gu and Zhao, 2012] published a survey of both commercial and academic virtualization solutions for embedded real-time systems. In the academic world, Steinberg and Kauer presented a hypervisor for x86 architectures, relying on hardware assistance for virtualization [Steinberg and Kauer, 2010]. Oikawa, Ito, and Nakajima developed the paravirtualization-based Gandalf Virtual Machine Monitor, which can host multiple instances of Linux and the RTOS μITRON on x86 single-core architectures [Oikawa et al., 2006]. The authors do not address VM scheduling. Sangorrin, Honda, and Takada realized a dual OS architecture of GPOS and RTOS based on ARM TrustZone, a processor extension of high-end ARM embedded processors that provides two virtual CPUs (secure VCPU and non-secure VCPU) mapped to a single physical CPU [Sangorrin et al., 2012]. The GPOS Linux is assigned to the non-secure VCPU and the RTOS TOPPERS/ASP is executed by the secure VCPU. The RTOS schedules the GPOS in an integrated manner as multiple RTOS tasks. Yoo et al. developed MobiVMM, a hypervisor for mobile phones with single-core ARM processors [Yoo et al., 2008]. The GPOS is scheduled in background, i.e., executed when the RTOS is idle. Nakajima et al. presented SPUMONE, a paravirtualization requiring hypervisor for SH-4A multicore processors [Nakajima et al., 2011, Li et al., 2012b]. Guest OSes are Linux and TOPPERS/JSP. A partitioned fixed-priority scheduling is applied, which schedules the GPOS in background when RTOS and GPOS share the physical core. Lin, Mitake, and Nakajima extended this work by runtime migration of virtual CPUs across cores in order to improve the performance of the GPOS (global scheduling of virtual CPUs) [Lin et al., 2013].

Crespo et al. developed for the avionics domain XtratuM, a paravirtualization bare-metal hypervisor for x86, LEON2, LEON3, PowerPC, and ARM [Masmano

et al., 2009, Peiró et al., 2010]. A redesign for multicore processors was recently published [Carrascosa et al., 2013]. VMs can be executed by multiple cores, with fixed cyclic scheduling or fixed priority scheduling. Each core has its own scheduler, the coexistence of both policies on different cores is possible. The static configuration of resource allocation, inter-VM communication, memory layout, temporal requirements of the partitions etc. is done by configuration files, very similar to our approach. SParK by Ghaisas et al. is a hypervisor for PowerPC platforms without hardware assistance for virtualization [Ghaisas et al., 2010]. However, their solution requires paravirtualization and does not support multicore platforms. Closest to our work, Tavares et al. presented an embedded hypervisor for PowerPC 405, which supports full virtualization, but no multicore architectures [Tavares et al., 2012].

Multiple independent projects use L4 microkernels for system virtualization. Heiser and Leslie introduced the OKL4 microvisor for ARM processors, a L4Ka::Pistachio derivative [Heiser and Leslie, 2010]. They evaluated the overhead by comparing virtualized and native Linux. Yang et al. proposed a virtualization solution for x64 uni-processor platforms based on the L4/Fiasco microkernel [Yang et al., 2011]. L4Linux is executed as a paravirtualized guest OS. The microkernel schedules each guest OS as a periodic thread according to scheduling information obtained from the guest OS. Similarly, Bruns et al. evaluated a L4/Fiasco microkernel and a paravirtualized Linux to consolidate subsystems of mobile devices on a single processor [Bruns et al., 2010]. LeVasseur et al. worked with the L4Ka::Pistachio microkernel with a Linux guest and proposed *pre-virtualization*, a semi-automatic preparation of an OS for virtualization [LeVasseur et al., 2008]. They achieve almost the same performance as paravirtualization with significantly lower engineering costs.

There are multiple studies that extend the popular open source hypervisors KVM and Xen for the application to embedded real-time systems. KVM (Kernel-based Virtual Machine) is a mainline kernel module that turns Linux into a hypervisor [Kivity et al., 2007]. It is therefore a hosted hypervisor (type 2). Cucinotta et al. examined hard reservations and an EDF-based soft real-time scheduling policy for KVM to provide temporal isolation among I/O-intensive and CPU-intensive VMs [Cucinotta et al., 2011b]. Their implementation uses only one core. Zhang et al. added support for real-time priorities to KVM [Zhang et al., 2010]. The Linux kernel is patched with PREEMPT-RT, which converts Linux into a fully preemptible kernel and adds hard real-time capabilities [Rostedt, 2007]. Comparably, Kiszka combined KVM and PREEMPT-RT [Kiszka, 2011]. He paravirtualized Linux in order to give the hypervisor a hint about the internal states of its guests and prioritized virtualization

workload over uncritical tasks. Ma et al. proposed a real-time virtualization architecture based on KVM with VxWorks and Linux as guests [Ma et al., 2013]. The SmartVisor [Su et al., 2009] uses KVM and Intel Atom's virtualization hardware extensions for full virtualization of Windows XP.

In contrast to KVM, Xen is a bare-metal hypervisor (type 1), which executes multiple domains (Xen terminology for virtual machines) [Barham et al., 2003]. The domain Dom0 is a privileged domain with direct access to the hardware (containing therefore the device drivers) and the ability to create and terminate other domains. Following the paravirtualization approach, the guests can use hypercalls to invoke hypervisor functions. Xen relies on either paravirtualization or on hardware assistance. PowerPC multicore architectures are supported. Gupta et al. implemented a feedback-controlled EDF scheduler in Xen [Gupta et al., 2006]. Lee et al. added support for soft real-time tasks by modifying Xen's Credit Based Scheduler [Lee et al., 2010]. Masrur et al. implement a novel scheduler in Xen: non-real-time domains are scheduled with a default Xen scheduler, real-time domains are scheduled with fixed priorities and get a higher priority [Masrur et al., 2010]. Xi, Lee et al. developed a real-time scheduling framework for Xen [Xi et al., 2011, Lee et al., 2011]. Their RT-Xen provides fixed-priority hierarchical real-time scheduling with different server types.

None of these hypervisors provides full virtualization on multicore PowerPC platforms without hardware assistance. They rely on either paravirtualization or processor virtualization extensions. Virtualization support was added to the PowerPC architecture with instruction set architecture Power ISA Version 2.06 [IBM, 2010], is however only available for high performance processors. Typical platforms for embedded systems do not feature hardware assistance and many OSes cannot be paravirtualized for legal or technical reasons. By consequence, the applicability of existing PowerPC hypervisors is limited significantly.

3.3 Proteus Multicore Hypervisor

In previous work, as a predecessor, a hypervisor for single-core PowerPC architectures with the same name *Proteus* was developed under the direction of Timo Kerstan in the context of his dissertation [Kerstan, 2011]. The diploma thesis of Daniel Baldin covered the development of the prototype [Baldin, 2009]. The results have been published in [Baldin and Kerstan, 2009]. In this work, we present a redesign for multicore platforms, which reuses fundamental parts of the predecessor. The development of the prototype under my direction was part of Katharina Gilles' master's

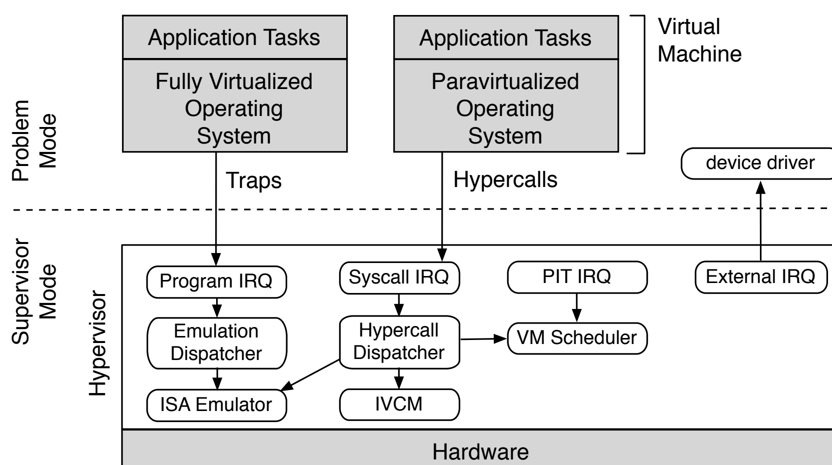


Figure 3.1: Design of the Proteus hypervisor (cf. [Baldin and Kerstan, 2009])

thesis [Gilles, 2012]. The results have been published in [Gilles et al., 2013]. Timo Kerstan, Daniel Baldin, and Katharina Gilles all contributed significantly to the results that are presented in this chapter.

3.3.1 Architecture

The design of the Proteus hypervisor is based on modularity. The functionality is accomplished by a cooperation of multiple modules, each providing a specific sub-functionality. Modules interact via well-defined interfaces and hide implementation details. This design paradigm increases the reusability and maintainability. It supports the development of robust systems, since modules can be validated separately, facilitated by their small size.

Figure 3.1 depicts the basic modules and the control flow based on interrupt handling. The PowerPC 405 features two execution modes [IBM, 2005]. In the *problem mode* only a subset of the instruction set can be executed. It is intended for application tasks. In the more privileged *supervisor mode* for system software, full access to hardware state and functionality is available via privileged instructions. Proteus executes only a near-minimum set of components in supervisor mode: address space management, VM context management, interrupt and hypercall handlers, VM scheduler, and Inter Virtual Machine Communication Manager (IVCM). Device drivers are executed in problem mode, reducing the damage a faulty driver can cause to system stability.

Any occurring interrupt causes a trap to privileged mode and invokes the hypervisor. The hypervisor examines the cause and and forwards the interrupt internally

to the appropriate component or back to the interrupt handler of the guest OS. If the execution of a privileged instruction in problem mode caused the interrupt (Program IRQ), it is forwarded to the emulation dispatcher to identify the corresponding emulation routine. In case of a hypercall or system call (Syscall IRQ), the hypercall handler invokes either the emulator, the inter-VM communication manager, or the VM scheduler. The VM scheduler is called in case of a Programmable Interval Timer (PIT) interrupt raised by the hardware timer device. An external interrupt is forwarded to the responsible device driver. [Kerstan, 2011]

Proteus is a *bare-metal* hypervisor: it runs directly on top of the hardware, in contrast to a *hosted* hypervisor that runs on top of a host operating system [Smith and Nair, 2005c]. A bare-metal design facilitates a more efficient virtualization solution regarding both latencies and memory consumption. The amount of code executed in privileged mode is smaller compared to a hosted hypervisor, since only a (preferably thin) hypervisor and no OS is incorporated in the trusted computing base (for the same reason device drivers are executed as introduced outside of the core hypervisor in problem mode). The attack surface is reduced, both the overall security and the certifiability of functional safety are increased.

A hosted hypervisor leaves resource management and scheduling at the mercy of the host OS. The entire system is exposed to the safety and security vulnerabilities of the underlying OS, which was often not designed to be part of a hypervisor. The architectural abstraction is restricted to the capabilities of the host OS. An example is KVM: it is tightly integrated into Linux and VM's are run and scheduled as Linux host processes by the existing process management infrastructure [Kivity et al., 2007]. Due to those performance and robustness advantages as well as the clearer and more scalable separation, the bare metal approach is more appropriate for embedded systems.

3.3.2 Configurability

In order to obtain a resource-efficient implementation regarding memory footprint, Proteus is configurable based on the specific requirements of an application. This comprises the inclusion/exclusion of entire modules and their parametrization. The preprocessor of the C programming language (`cpp`) [Kernighan and Ritchie, 1988] is used, a popular technique to implement configurability for C/C++ software [Ernst et al., 2002, Liebig et al., 2011]. The C preprocessor is called by the compiler to modify source code before translating it. Proteus' configurability is therefore restricted to compile time: configurability is lost once a program is preprocessed and

a reconfiguration requires re-compiling. [Kerstan, 2011]

The C preprocessor provides the ability to generate different variants of any C program. The source code is annotated with preprocessor directives. The three basic mechanisms are file inclusion (`#include` directive), textual substitution (`#define`), and conditional compilation (`#ifdef`). Most important, conditional compilation defines separate code branches and includes or excludes them dependent on the conditions in the output of `cpp`. If not included in `cpp`'s output, a code branch is not compiled. [Kernighan and Ritchie, 1988]

In order to configure Proteus, the system designer sets the values for the conditions. He modifies a configuration file, but does not have to touch the actual source code. This configuration files has two sections, one for the hypervisor itself and one for the VMs. Regarding the hypervisor, features such as paravirtualization, TLB virtualization, or inter-VM communication can be enabled or disabled. In addition, parameters like the number of processor cores and the initial number of VMs have to be set. For each VM a set of parameters has to be specified, e.g., start and end address of its address space, scheduling parameters, or core affinity.

Based on this configuration file, the preprocessor includes and excludes source code on the granularity of lines of code inside the source files. As a result, no unneeded source code is included in the executable software file. Figure 3.2 depicts the configuration process. In this example, the system designer disabled the feature TLB virtualization in the configuration file. On the left, an excerpt of the source code of the hypervisor function `vm_resume` is given, the function that resumes the execution of a VM. This configuration file and all source files are the input for the preprocessor. As specified in the configuration file, the preprocessor excludes the call of the function `vm_restore_tlb`, which restores the TLB entries back to the values that were stored when the VM was paused. As a consequence, the call is as well not included in the executable file produced by compiler and linker.

The following modules can be included or excluded: virtualization of the TLB, inter-VM communication via shared memory, paravirtualization, driver for Ethernet network, functionality for VM migration as introduced in detail in Chapter 7 (requires Ethernet driver), Innocuous Register File Mapping (see Section 3.4.2), and previrtualization. Pre-virtualization is an approach to paravirtualize guests automatically [LeVasseur et al., 2008]. The source code is analyzed at compile time in order to identify privileged instructions. At load time, the hypervisor replaces privileged instructions by hypercalls. Finally, one of three VM scheduling policies has to be selected: fixed time-slice scheduler, fixed priority scheduler, or server-based

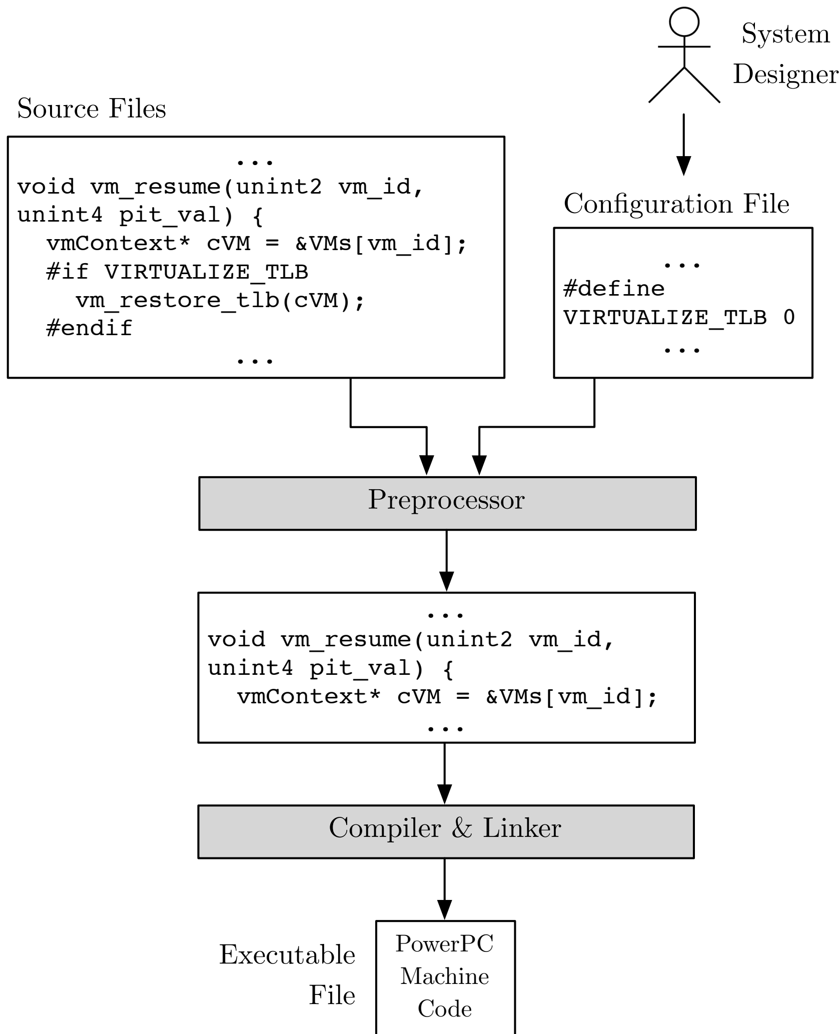


Figure 3.2: Preprocessor-based configuration by conditional compilation

scheduler (see Chapter 6).

Note that configurability does not mean portability. The configurability includes only the selection and parametrization of features, not the inclusion of processor architecture specific source code. In fact, up to now Proteus supports only the PowerPC ISA.

3.3.3 Processor Virtualization

As introduced in Section 2.2.2, an instruction set is virtualizable according to the theorem of Popek and Goldberg if the set of sensitive instructions is a subset of the set of privileged instructions [Popek and Goldberg, 1974b] (more precisely, Popek and Goldberg's theorem defines trap-and-emulate virtualizability). Fortunately, the PowerPC fulfills Popek and Goldberg's requirement and is therefore fully virtualizable, as shown in [Kerstan, 2011]: all sensitive instructions cause an exception if executed in problem mode.

The PowerPC 405 does not provide explicit hardware support for virtualization such as an additional hypervisor execution mode. Only two execution modes are available: the problem mode for tasks and the supervisor mode for system software. Solely the hypervisor is executed in supervisor mode. The guests (both OS and application tasks) are executed in problem mode with no direct access to the machine state. This limitation of the guests' hardware access is mandatory in order to retain the hypervisor's control over the hardware and guarantee the separation between VMs.

However, for the controlled execution of application tasks, guest operating systems rely themselves on an execution-mode differentiation. Therefore, the problem mode has to be subdivided into two virtual execution modes: VM's supervisor mode and VM's problem mode (see Figure 3.3). By virtualizing the machine state register, the hypervisor creates the illusion that a guest OS is executed in supervisor mode, but runs it actually in problem mode. Consequently, the execution of a privileged instruction by the guest OS (e.g., an access to the machine state register) causes a trap and the hypervisor executes the responsible emulation routine. The execution of a privileged instruction or a system call by a task within the VM's virtual problem mode causes a trap that is forwarded by the hypervisor to the guest OS and handled by it. [Kerstan, 2011]

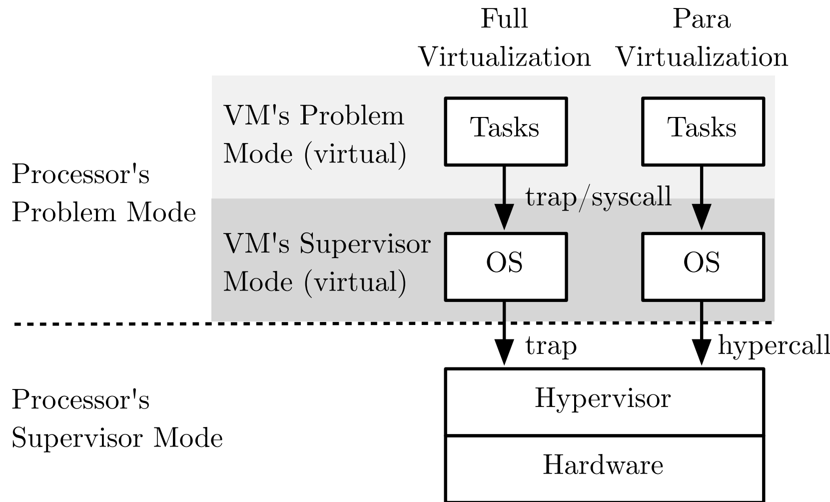


Figure 3.3: Execution mode differentiation and mechanisms for mode transition: extension of the processor's two modes by two virtual modes

3.3.4 Paravirtualization Interface

Next to full virtualization, Proteus supports paravirtualization because of its efficiency (low latencies) and the advantages of an explicit cooperation of OS and hypervisor, for example for hierarchical real-time scheduling or resource sharing. If the modification of an OS is possible, the system designer decides whether the effort of paravirtualization is justified. The concurrent hosting of both paravirtualized and fully virtualized guests is possible without restriction. A natural approach is to host a paravirtualized RTOS and a fully virtualized GPOS, since the benefits of paravirtualization are less important for a GPOS and the ability to paravirtualize it is often restricted. In addition, bare-metal applications without underlying OS can be hosted.

Paravirtualization requires a porting of the guest operating system to the paravirtualization application binary interface (ABI), specifying the set of hypercalls and calling conventions, incl. size and alignment of data types as well as which registers are used to pass arguments and retrieve return values. The hypercall mechanism is implemented by using the available system call infrastructure: they are executed with the system call (`sc`) instruction. In case of a system call interrupt, the hypervisor detects whether it is a hypercall or a system call by analyzing the execution mode of the processor in which the system call instruction was executed. A system call is identified as a hypercall, if the system call instruction was executed in the virtual VM's supervisor mode. A system call is executed in the virtual VM's problem

mode. The hypervisor's hypercall dispatcher calls the associated hypercall handler routine. [Kerstan, 2011]

The paravirtualization interface consists of hypercalls of two categories. First, the interface includes a hypercall for each sensitive instruction. Sensitive instructions have to be emulated by the hypervisor. The hypercalls provide detailed information on how to handle it, which reduces the hypervisor's overhead of analyzing what caused the context switch from guest to hypervisor. Second, the interface provides high-level hypercalls, which can be called by a paravirtualized OS to communicate with other guests, pass scheduling information to the hypervisor, or yield the CPU:

- `ivcm_create_tunnel`: Create a shared memory tunnel to another VM.
- `sched_set_param`: Pass information to the scheduler of the hypervisor.
- `sched_yield`: If a guest idles, it can inform the hypervisor and cooperatively terminate the assigned execution time slice. [Kerstan, 2011]

3.3.5 Multicore

Proteus is a hypervisor for homogeneous multicore PowerPC 405 processors with a shared main memory. It uses the cores in a symmetric manner: all processor cores execute guest systems. When the guest traps or calls for a service, the hypervisor takes over control and its own code is executed on that core. This context switch from guest system to hypervisor can be performed from different guests on different cores at the same time, which is why this design is scalable regarding the number of processor cores.

A design alternative would have been the *sidecore* approach: one dedicated core executes the hypervisor exclusively, the other cores execute guest systems [Kumar et al., 2007]. When an interrupt occurs, the hypervisor on the sidecore handles it and no context switch is invoked on the core that executes the guest. The hypervisor may either be informed via an interprocessor interrupt (not featured by the PowerPC 405) or a notification by the guest OS (either by a hypercall or a write to a specified memory address that is polled by the hypervisor), which requires paravirtualization. To reconcile sidecore approach and full virtualization, a small fraction of the hypervisor could be executed on each core to forward interrupts. The guest OS could run unmodified, but each trap would involve a context switch and thereby a loss of the major benefit. If the sidecore is already serving the request of a guest, other guests have to wait, resulting in a varying interrupt processing time, which is inappropriate for real-time systems. For these reasons, Proteus uses the cores in a symmetric manner.

Proteus provides functionality for both a fixed or a dynamic mapping of virtual machines to cores. In the first case, each VM is executed only on a single core and a ready queue for each core is maintained by the hypervisor to implement a scheduling policy. In the second case, the VMs are managed in a global ready queue and at runtime assigned to one of the cores. The execution of a VM might be paused and resumed on a different processor core (migration across cores). A mixture with VMs that are bound to one specific core and globally scheduled VMs is possible as well. A VM can be bound to a subset of cores. It is not possible to execute a VM in parallel on more than one core at the same time.

Each core has its own heap memory, its own stack, and a private memory section for core-specific global variables of the hypervisor. If, for example, VMs are statically allocated to a core, the ready queue and the VM contexts of momentarily not executed VMs are stored in this memory section. For globally scheduled VMs, the memory that stores the VM context is shared and accessible from all cores.

In a multicore system, the synchronization of access to shared resources is a complex challenge. For example, the access to the UART interface, the scheduler's global ready queue, or the shared memory for inter-VM communication has to be protected since multiple VMs executed on different cores might try to access it at the same time. A common solution are semaphores, accessed under mutual exclusion and assigned exclusively to one VM at any time. Semaphores rely on atomic operations, however, the PowerPC 405 does not feature multicore-safe atomic operations or any other hardware support to realize mutual exclusion in a multicore architecture. Its instructions `lwarx` (load locked) and `stwcx` (store conditional) for atomic memory access do not work across multiple processor cores. Interrupt disabling is a solution for uniprocessor systems, but is as well not safe for multicore systems.

Therefore, Proteus implements a software solution for this synchronization problem of access to resources that are shared across processor cores: Leslie Lamport's Bakery Algorithm [Lamport, 1974]. This algorithm works in two rounds. In the first round, VMs that try to access the critical section receive a number and these numbers increase by one with each request. As this number assignment is not done under mutual exclusion, more than one VM might receive the same number. In this case, the VMs' ID are used as a tie breaker. In the second round, a VM waits until no other VM is just receiving its number and until all VMs with smaller numbers (or the same number, but lower ID) finished their resource access. The Bakery Algorithm does not require atomic operations such as test-and-set, satisfies first-in-first-out fairness and excludes starvation (in which a VM never gets access to the resource), an

advantage over Dijkstra's algorithm [Dijkstra, 1965]. Essential for real-time systems, the waiting time is bounded, as derived in Section 3.4.2. Multiple shared resources protected by semaphores impose the risk of deadlocks. Proteus does not allow to hold more than one semaphore at a time, precluding circular waiting.

3.3.6 Memory Virtualization

Spatial separation is a key requirement for system virtualization, especially for safety-critical embedded systems. It refers to the protection of the integrity of the memory space of both the hypervisor and the guests. Any possibility of a harmful activity going beyond the boundaries of a VM has to be eliminated. To achieve this, each VM operates in its own virtual address space, which is statically mapped to a dedicated region of the shared main memory. A hardware memory protection component is required to preclude that a guest system accesses memory that has not been allocated to it. Such a Memory Management Unit (MMU) is part of the PowerPC 405. All memory references pass the MMU. Next to memory protection, the MMU translates the virtual memory addresses to physical addresses. Proteus requires a MMU and most embedded processors that provide enough capability to allow the hypervisor-based execution of multiple operating systems feature one.

Virtualization adds a level of address translation to memory management. As usual, guest virtual memory addresses are translated to guest physical addresses using the guest operating system page tables. In addition, the hypervisor translates the guest physical addresses to machine physical addresses based on the static allocation of exclusive memory regions. A guest OS manages the assigned virtual memory and can create pages. Proteus virtualizes the MMU and maps these virtual pages to physical pages.

The Translation Look-aside Buffer (TLB) is a cache for the translations of the MMU. In case of a TLB miss, the miss handler fetches the translation from the page table of the process. Since the TLB of the PowerPC is software-managed, Proteus has to virtualize the TLB and maintains a virtual TLB for each VM to store the VM-specific TLB content. [Kerstan, 2011]

3.3.7 Virtualization of Timer and I/O Devices

An accurate timekeeping with sufficient granularity is essential for real-time systems. The PowerPC features for this purpose a hardware timer, which provides the required timing functions. The time base produces a periodic signal, the so-called ticks. The decrementer is a counter, set by software to a value. It decrements at the same rate

as the time base, i.e., by one per tick, and triggers an interrupt when the counter reaches zero. The decrementer is used as a programmable interval timer (PIT) for generating interrupts after the specified time interval has elapsed.

The PIT has to be shared: both the hypervisor and the guest operating systems rely on it for their preemptive scheduling. Before resuming the execution of a VM respectively task, the PIT register is set to the desired maximum execution time until the next scheduling decision. The virtualization of the PIT Register is achieved by a dedicated virtual register for the hypervisor and each VM. In case of a PIT interrupt, the hypervisor analyzes the values of the virtual PIT register of the interrupted VM and of its own virtual PIT register in order to detect whose timer expired and invokes the associated interrupt handler. If both counter reach zero at the same time, the hypervisor has priority. The hypervisor is responsible for updating the virtual PIT registers. [Kerstan, 2011]

Input/output (I/O) devices are assigned statically to VMs. Guest systems obtain a dedicated I/O device, which is accessed by memory mapped I/O: the hypervisor maps the associated memory area statically to the VM's address space and the guest can access the memory area and hence the device directly. There is no emulation overhead and no involvement of the hypervisor after the initialization, a concept introduced by Liu et al. as *VMM-Bypass I/O* [Liu et al., 2006].

3.4 Evaluation

3.4.1 Evaluation Platform: IBM PowerPC 405

Target architecture of our implementation are processors with multiple IBM PowerPC 405 (PPC405) cores [IBM, 2005], designed for low-power embedded systems. It features the required Memory Management Unit and Programmable Interval Timer. Specifications and register-transfer level description are freely available to the research community. Porting the results to other PowerPC processors should be fairly simple due to the API compatibility within the PowerPC family.

CPU. The PPC405 is a 32-bit RISC core, providing up to 400 MHz (in our case 300 MHz). It implements the Power Instruction Set Architecture Version 2.03 [IBM, 2006] and features 32 general purpose registers, a 5-stage pipeline, static branch prediction, an Arithmetic Logic Unit, a Multiply-Accumulate Unit, and an interrupt interface for one critical and one non-critical interrupt signal.

MMU. The MMU provides an independent enabling of instruction and data translation/protection, page level access control, and software control of the page

replacement strategy. The software-managed TLB has 64 entries, plus hardware-managed shadow TLBs (4 entries for instructions, 8 entries for data).

Caches. The PPC405 features an on-chip instruction cache unit and a separate data cache unit (in our case both of size 32kB).

Timers & Debug Interface. As timer facilities, a Programmable Interval Timer (PIT), a Fixed Interval Timer (FIT), and a watchdog timer are provided with a 64-bit time base. For debugging, a JTAG interface offers instruction tracing and multiple instruction address compares, data address compares, and data value compares.

In order to be able to evaluate the software with low effort on different hardware configurations, the evaluation platform is a software simulator for PowerPC multi-cores [IBM Research, 2012]. The *IBM PowerPC Multicore Instruction Set Simulator* emulates PPC405, PPC440, PPC460, and PPC470 processor cores and can optionally include an interrupt controller, main memory (in our case 512 kB of local memory), and peripheral devices (e.g., an UART). It provides an interface for external simulation environments, which we do not use. Many components of the simulated hardware can be configured, e.g., the number of cores (1, 2, or 4) or cache sizes.

The *RISCWatch* debugger program is used to interface the simulator, control the execution, and debug code. The user can step through the code cycle by cycle, examine and manipulate memory locations and registers, and analyze the contents of the caches. RISCWatch features a tracing of instruction and data accesses and saves the entire trace of the executed software in a file.

When the simulator is configured to simulate the PPC405 processor, all architected processor resources are modeled. It operates in cycle approximate mode due to several functional limitations. Fortunately, these functional limitations do not occur during the execution of the software routines that are analyzed for this thesis. For this reason the execution is cycle accurate.

3.4.2 Execution Times

The execution times as presented in the following were determined by counting the processor cycles of the execution of the investigated code part. The basis for this is the instruction trace generated by the simulator. All instructions are executed by the PowerPC 405 in one processor cycle, except from multiplication and division, memory access with cache miss, and branch instructions [IBM, 2005]. For instructions that do not execute in one cycle, the worst-case execution time is always assumed (five cycles for a multiplication, three cycles for a branch with unsuccessful branch prediction). The simulator is not cycle accurate for external memory accesses, which is why we

execute the analyzed software routines with preloaded instruction and data caches, resulting in a duration for each instruction fetch of one processor cycle, including load/store. In practice, this is not always possible, resulting in a larger execution time, depending on the system's hardware.

Virtual Machine Context Switch

If multiple virtual machines share a core, switching between them involves saving of the context of the preempted VM (content of the virtualized registers), selection of the next VM, and resuming of this VM, including restoring of its context. The execution time for this procedure is 1250 ns (375 processor cycles), without scheduling, since the execution time is highly dependent on the specific scheduling algorithm. The algorithm-specific execution time has to be added.

Synchronized Shared Resource Access Routines

The execution time of the semaphore operations `wait()` and `signal()` are plotted in Figure 3.4.2, implemented according to Lamport's Bakery Algorithm for synchronized shared resource access across processor cores. The execution time increases linearly with the number of cores, since the operations perform an iteration over an array of length equal to the number of cores.

However, these execution times denote actually the best case for the operation `wait()`, namely an interrupted execution of the operation, which takes place if resource access is granted immediately. When the resource is not available, `wait()` causes a blocking of the calling VM. The worst-case blocking time is essential for real-time systems. In case of four cores, the worst case occurs if the calling VM is blocked by a VM on each of the three other cores. In this case, the worst-case blocking time for synchronized shared resource access sums up to 1797 processor cycles or about $6\mu s$ [Gilles, 2012].

Interrupt Latency

Virtualization increases the interrupt latency. Any interrupt is first delegated to the hypervisor, analyzed there, and potentially forwarded back to the guest operating system. For example, the additional latency of a programmable timer interrupt (PIT IRQ) is about $6.6\mu s$ [Kerstan, 2011]. The additional latency for a system call interrupt (Syscall IRQ) is about $4\mu s$ [Kerstan, 2011]. To obtain the total interrupt latency, one has to add this delay to the interrupt latency of the guest OS. The additional latency is longer for the timer interrupt, since the virtual interrupt timer

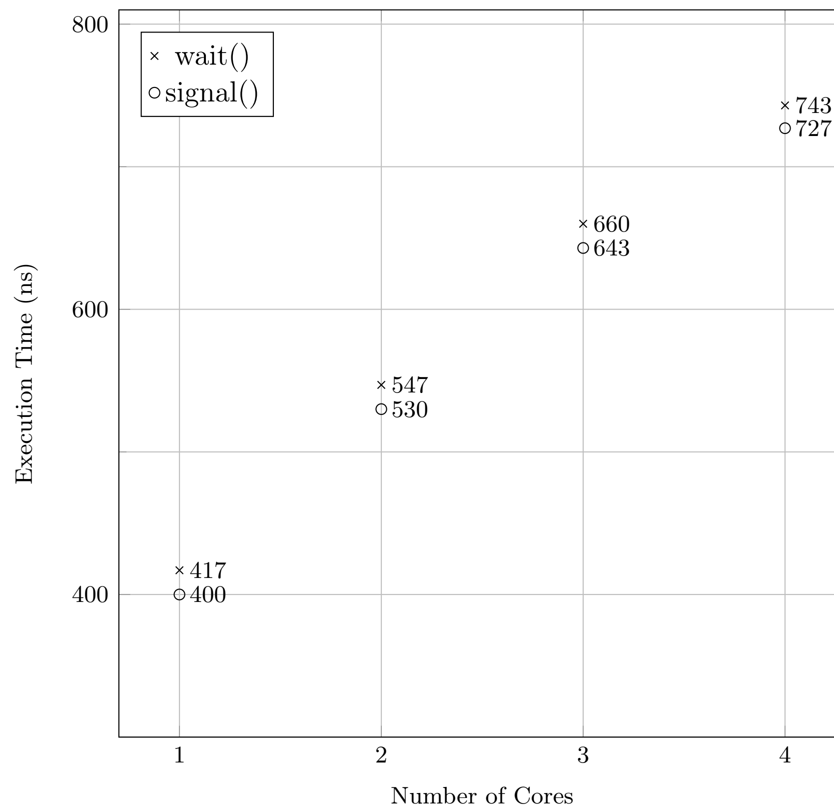


Figure 3.4: Execution time of routines for protected access to a shared resource

has to be updated, as discussed in Section 3.3.7. In case of a system call interrupt, the hypervisor just has to analyze in which virtual processor execution mode it was raised, in order to find out whether it was caused by a hypercall or by a system call. In case of a hypercall, the hypervisor dispatches to the associated hypercall handler. Otherwise, the interrupt is delegated back to the VM and the latencies for this case are given above.

Efficient Virtualization by Paravirtualization

The emulation of privileged instructions is the major cause of virtualization overhead. The virtualized execution of instructions that manipulate or depend on the hardware state is the core functionality of a hypervisor. Since the guest is executed in problem mode, it necessarily includes a context switch to the hypervisor and a processor mode switch. The emulation service is requested via interrupt (Programm IRQ) in case of full virtualization or hypercall in case of paravirtualization.

Paravirtualization can be exploited to achieve a significant speedup for the emulation of privileged instructions. An analysis of the steps of an emulation routine helps to understand why:

1. Reenabling of the data translation and saving of the contents of those registers that are needed to execute the emulation routine.
2. Analysis of the exception in order to identify the correct emulation subroutine and jump to it (dispatching).
3. Actual emulation of the instruction.
4. Restoring of the register contents.

The actual emulation accounts for the smallest fraction of the total execution time and is the same for both full virtualization and paravirtualization. A significant performance gain of paravirtualization is based on the lower overhead for identification of the cause of the exception and dispatching to the correct handler routine. In case of full virtualization, a memory access is required to identify the instruction that raised the interrupt, since the PowerPC stores only the address of the instruction in a register. In addition, it includes the analysis in which virtual processor privilege mode the instruction was executed. In case of paravirtualization, only a register read-out is necessary in order to obtain the hypercall ID, since Proteus' hypercall application binary interface specifies that register 13 is used to pass the hypercall

ID. The WCET of privileged instructions is between 7% and 42% smaller in case of paravirtualization compared to full virtualization. [Baldin, 2009]

An additional performance gain for paravirtualized guests is achieved by Innocuous Register File Mapping (IRFM) [Kerstan, 2011]. For each VM, the hypervisor maintains a virtual register set. Accesses to registers that have to be accessed by privileged instructions trap to the hypervisor, which emulates the instruction on the virtual register set. However, there are registers that can be accessed without having immediate influence on the state or behavior of the VM. By IRFM's mapping of this specific set of registers into the memory space of the VM, they can be accessed by load and store instructions without trapping to the hypervisor. Paravirtualization is required since all calls of privileged instructions to access the registers have to be replaced by calls of load and store. A similar approach has been applied by Xen [Barhamet al., 2003].

Hypercalls

As introduced in Section 3.3.4, a guest operating system can request hypervisor services via the paravirtualization interface. The hypercall `vm_yield`, which voluntarily releases the core, has an execution time of 507 ns (152 processor cycles). By calling `sched_set_param`, the guest OS passes information to the hypervisor's scheduler. The execution time of this hypercall is 793 ns (238 processor cycles). The hypercall `create_comm_tunnel` requests the creation of a shared-memory tunnel for communication between itself and a second VM and is characterized by an execution time of 1027 ns (308 processor cycles). The hypercall `vm_yield` does not return to the VM and the execution time is measured until the start of the hypervisor's `schedule` routine. The other two hypercalls return to the VM and the execution time measurement is stopped when the calling VM resumes its execution.

3.4.3 Memory Footprint

Proteus can be configured offline dependent on the specific requirements of the application, as introduced in Section 3.3.2. In order to reduce the memory footprint, unneeded functional modules can be excluded. Figure 3.5 lists code and data size for the base functionality and the additionally required memory for different features. The hypervisor is written in C and assembly language. The efficiency of a hypervisor is highly dependent on the execution times of the interrupt handling. For this reason, most of the components called by those handlers and the handlers themselves are written in assembly language. All executables are generated with compiler optimiza-

Feature	Memory Footprint [bytes]			
	Assembler	C code	Data	Total
Base	2492	5732	2980	11204
ParaV	252	0	148	400
IRFM	292	476	0	768
PreV	0	256	0	256
TLB V	812	264	656	1732
Driver	0	648	12	660
IVCM	0	500	0	500
Total	3848	7876	3796	15520

Figure 3.5: Impact of individual components on memory footprint

tion level 2 (option -O2 for the GNU C compiler), which focuses on the performance of the generated code and not primarily on the code size. The solely full virtualization supporting base requires a total of about 11 kB. The addition of paravirtualization, Innocuous Register File Mapping, Pre-Virtualization, driver support, or inter-VM communication accounts in each case for less than 1 kB. TLB Virtualization adds less than 2 kB. If all features are enabled, the memory requirement of the hypervisor sums up to about 15 kB.

3.5 Summary

A hypervisor for embedded real-time systems has to meet specific requirements that differ from those of the server domain, as introduced in Section 3.1. This chapter presented the hypervisor *Proteus*, which fulfills these requirements:

Real-time Capability. Proteus provides deterministic behavior for all of its operations and bounded interrupt latencies. Kerstan showed how to derive worst-case execution times for a guest system executed on top of Proteus [Kerstan, 2011]. An appropriate real-time virtual machine scheduling is introduced in Chapter 6.

Safe and Secure Partitioning. Proteus retains control of the hardware resources at all times by executing all guests in the less-privileged problem mode of the processor. As a thin bare-metal hypervisor, the trusted computing base is small, which increases security and certifiability of functional safety. Spatial isolation is realized by statically allocating each VM and the hypervisor itself in unique address spaces, not accessible by other VMs. The integrity of these address spaces is ensured

by the memory access protection of the MMU. Device drivers are not executed in the privileged mode and cannot damage the stability of the entire system.

Secure Inter-VM Communication. VMs can communicate via shared memory. The hypervisor sets up the communication and ensures mutually exclusive access. The spatial isolation guarantees that only authorized VMs have access.

Multicore Support. Proteus supports homogeneous multicore processors. VMs can be statically or dynamically mapped to processor cores. Access to shared resources is synchronized in a multicore-safe manner.

Scalability. The number of VMs is limited only by the system's memory. The number of processor cores is not restricted and the symmetric use of the cores scales with the number of VMs and cores.

Efficiency. The execution time of synchronization primitives, hypercall handlers, emulation routines, VM context switch as well as the additional interrupt latencies are in the range of a few microseconds. Paravirtualization and advanced techniques such as Innocuous Register File Mapping reduce the virtualization overhead.

Small Memory Footprint. The memory footprint is 11 kB for the base functionality and 15 kB for a configuration with all functional features.

Configurability. Proteus' compile-time configurability offers the exclusion of entire modules and module parametrization. It is based on a configuration file, with which the system designer specifies the application-specific requirements.

Support of Both Paravirtualization and Full Virtualization. Proteus supports both full virtualization and paravirtualization without relying on special hardware support, even side by side. Paravirtualized operating systems can use hypercalls to communicate with other guests, pass scheduling information to the hypervisor, or yield the CPU.

Support of OS-less Guests. Bare-metal applications without underlying operating systems can be executed.

Fundamental results could be reused from the predecessor, a hypervisor for single-core PowerPC architectures with the same name *Proteus* [Kerstan, 2011]. In this work, we present a redesign for multicore platforms, including synchronization mechanisms for shared resource access and a multicore scheduling infrastructure that enables both partitioned and global virtual machine scheduling. The Proteus hypervisor serves as an evaluation platform for both the scheduling policy (Chapter 6) and the virtual machine migration (Chapter 7).

Chapter 4

Models

Contents

4.1 Workload Model **70**
4.1.1 Task Model 70
4.1.2 Virtual Machine Model 73
4.2 Resource Model **76**
4.3 Schedulability Analysis **77**
4.4 Suitability of the Model **79**
4.5 Related Work **81**
4.6 Summary **85**

4.1 Workload Model

The workload is composed of a set of virtual machines. A virtual machine, in turn, is characterized as a set of tasks. We abstract from details of the operating system that is executed within the virtual machine, except for the applied task scheduling algorithm. In the following, the details of task model and virtual machine model are introduced.

4.1.1 Task Model

The task model is a combination of the classic periodic real-time task model of Liu and Layland [Liu and Layland, 1973b] and the elastic resource distribution framework of Marau et al. [Marau et al., 2011]. According to the *periodic task model*, as introduced in Section 2.1.2, repeatedly executed computations or data transmissions are modeled as periodic tasks, which are sequential activities to be executed on a single core at any time (suspension, migration to another core and subsequent resumption is possible). Each task τ_i is defined as an infinite sequence of jobs and characterized by a period T_i, denoting the time interval between the activation times of consecutive jobs. The worst-case execution time (WCET) C_i of a task represents an upper bound on the amount of time required to execute the task. We assume implicit deadlines for all tasks (relative deadline $D_i = T_i$), so a real-time job must be completed T units of time after its activation at the start of its period. The execution of a job may be preempted at any time prior to completing execution and resumed later. The utilization $U(\tau_i)$ is defined as the ratio of WCET and period: $U(\tau_i) \stackrel{\text{def}}{=} C_i/T_i$.

A *criticality level* χ is assigned to each task, referring to its importance for the overall system and the severity of failure. Moreover, it might be associated with a specific certification level as defined by a certification authority. Only two generic criticality levels are assumed in this work: HI and LO, with HI denoting higher severity of failure and stronger certification requirements. The periodic task model is applicable to many real-time applications and well-suited for mixed-criticality systems. Certification authorities require to determine WCETs to demonstrate the correctness of the system under pessimistic assumptions and numerous methods and tools to support the determination of WCETs exist [Wilhelm et al., 2008]. Nevertheless, it should be mentioned that the results of this thesis could be transferred to the *sporadic* task model, which defines each task by a WCET, a relative deadline, and a minimum inter-arrival time between subsequent jobs [Mok, 1983].

Marau et al. extended the periodic task model in order to realize an *elastic*

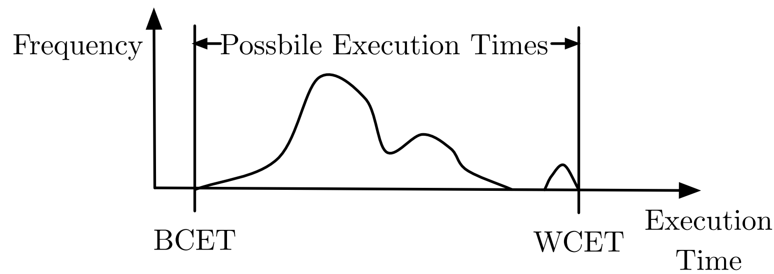


Figure 4.1: Distribution of execution times of a task between best-case and worst-case execution time

scheduling [Marau et al., 2011]. A task [1] τ_i is characterized by a minimum bandwidth $U_{min}(\tau_i) \geq 0$ and a maximum bandwidth $U_{min}(\tau_i)+U_{lax}(\tau_i)$. Therefore, the utilization laxity $U_{lax}(\tau_i) \geq 0$ can be understood as the utilization that the task could use reasonably in addition to the minimum bandwidth. First, all tasks receive their minimum bandwidth U_{min}, since this allocation is required for the correct execution. U_{min} can be derived from the WCET as $U_{min}(\tau_i) = C_i/T_i$. Given a (fixed) resource capacity U_R, the spare bandwidth defines the bandwidth that can be distributed among n tasks after having guaranteed their minimum requirements: $U_{spare} \stackrel{\text{def}}{=} U_R - \sum_{i=1}^{n} U_{min}(\tau_i)$. It is distributed based on a weight $w(\tau_i)$ (e.g., a quality of service parameter). The actual bandwidth allocation $U(\tau_i)$ to a task τ_i is within the range $[U_{min}(\tau_i), U_{min}(\tau_i) + U_{lax}(\tau_i)]$. Consequently, the maximum allowed execution time within a specific period of the task is within the range $[T_i \cdot U_{min}(\tau_i), T_i \cdot (U_{min}(\tau_i) + U_{lax}(\tau_i))]$.

The WCET denotes the upper bound of the execution time, based on the most pessimistic assumptions of the execution of the task. The actual execution time of a job depends on the initial state, the amount and characteristics of the input data (for embedded systems often dependent on the environmental conditions) as well as the state of the hardware platform (e.g., cache miss or hit, pipeline stalls) and is in many cases considerably lower than the WCET [Wilhelm et al., 2008]. Varying execution times are assumed in the context of this work between a (potentially unknown) best-case execution time (BCET) and the known WCET, characterized by an arbitrary and unknown probability distribution, as depicted in Figure 4.1.

Finally, tasks can have multiple *operational modes*. These modes are mutually exclusive, i.e., a single mode is active at each point in time. By means of a mode

[1] Marau et al. refer more generically to *services*, which are either a periodic task or a message stream.

change, the system is able to adapt to changes in the environment [Fohler, 1993]. See Real and Crespo for a survey on multi-mode real-time systems [Real and Crespo, 2004]. Each mode produces a different behavior and implements a different functionality or delivers a different quality of service. In the context of this work, modes are characterized by differing resource demands, in particular regarding processor utilization. They differ regarding utilization laxity U_{lax} and potentially regarding w. Mode-independent task parameters are criticality level, period, relative deadline, and minimum utilization U_{min}. An exception is the disabling of tasks as a special case of a mode change, i.e., it is not executed by the operating system (no resource requirement at all: $U_{min} = 0$), and the corresponding enabling.

A task is characterized by a set of 5-tuples, in which each tuple defines a mode ($M_1, ..., M_m$ indicate different modes of the enabled task, off refers to the disabled mode):

$$\begin{aligned}
\tau_i \stackrel{\text{def}}{=} \{\tau_i^{off} &= (T_i, C_i^{off} = 0, \chi_i, U_{lax}(\tau_i)^{off} = 0, w(\tau_i)^{off} = 0), \\
\tau_i^{M_1} &= (T_i, C_i, \chi_i, U_{lax}(\tau_i)^{M_1}, w(\tau_i)^{M_1}), \\
\tau_i^{M_2} &= (T_i, C_i, \chi_i, U_{lax}(\tau_i)^{M_2}, w(\tau_i)^{M_2}), \\
&\vdots \\
\tau_i^{M_m} &= (T_i, C_i, \chi_i, U_{lax}(\tau_i)^{M_m}, w(\tau_i)^{M_m})\} \qquad \text{, where}
\end{aligned}$$

- $T_i \in \mathbb{N}^+$ is the period ($D_i = T_i$ is the relative deadline),
- $C_i \in \mathbb{N}$ represents an upper bound on the execution time,
- $\chi_i \in \{LO, HI\}$ is the criticality level,
- $U_{lax}(\tau_i) \in [0, 1]$ is the utilization laxity,
- $w(\tau_i) \in [0, 1]$ is the weight.

U_{lax} and w differ for each mode (in addition, $C^{off} = 0$ and $w^{off} = 0$), the other attributes are mode-independent.

Summary. The task model is based on the classic periodic real-time task model. There is a known worst-case execution time bound, but the actual execution times of the jobs can be lower and vary at runtime. The task model is elastic: each task is characterized by a minimum utilization and a utilization laxity that the task could use reasonably in addition. A criticality level refers to the importance of the task for the overall system. Tasks can have several operational modes, which differ regarding behavior and resource demand.

4.1.2 Virtual Machine Model

A virtual machine V_k is modeled as a set $\Omega(V_k)$ of tasks and the task scheduling policy $\sigma(V_k)$ that is applied by the guest operating system. Notation $\tau_i \in \Omega(V_k)$ denotes that task τ_i is executed in V_k. The utilization of a VM is the sum of the utilizations of its tasks, implying that operating system overhead is neglected: $U(V_k) = \sum_{\tau_i \in \Omega(V_k)} U(\tau_i)$. Independent VMs are assumed, with neither shared resources except for the processor (static and exclusive allocation of memory regions), nor data dependencies, nor inter-VM communication.

Many characteristics of the task model apply to the virtual machine model as well. A criticality level χ is assigned to each VM. If a VM's task set is characterized by multiple criticality levels, the highest criticality level determines the criticality of the VM. The assignment of criticality on VM level is no restriction, since our use case deals with the consolidation of already certified systems.

Virtual machines can have multiple mutually exclusive operational modes. Modes differ regarding the set of tasks: which tasks are executed (and not disabled) and in which mode are they executed? Analogous to task modes, each VM mode produces a different behavior or delivers a different quality of service, and is characterized by a differing resource demand (processor utilization). Entire VMs can be disabled and enabled as well.

From the point of view of the hypervisor, we consider VMs to be schedulable entities that can be represented comparably to the task model. Since the task set executed by a VM is characterized by varying execution times, the same is true for the VM itself. The actual execution time within a specific period is in the range of a best-case and a worst-case execution time with an unknown probability distribution. The best case occurs if all jobs of the VM's tasks within the VM period execute for their BCET; the worst case occurs if all jobs execute for their WCET. In the usual case of an arbitrary distribution of the jobs' execution times, the actual execution time of the VM lies in between this range.

For the hypervisor's VM scheduling, each VM V_k will be abstracted as a task that requires a minimum computation bandwidth $U_{min}(V_i)$ to carry out the timely execution of its internal task set $\Omega(V_i)$. However, it may benefit in terms of a higher quality of service from additional bandwidth, if available, up to $U_{lax}(V_i) \geq 0$. The required minimum utilization $U_{min}(V_i)$ and the maximum extra bandwidth $U_{lax}(V_i)$ are dependent on the task set $\Omega(V_i)$ and on the task scheduling policy $\sigma(V_i)$ and derived in the context of the periodic resource model design (Section 4.2).

The *demand bound function* computes for a time interval of length t starting at

an arbitrary point in time t_0 the total computation time demand of a task, i.e., the sum of the execution times of all jobs with arrival time and deadline in $[t_0+t]$. It was introduced by Baruah, Rosier, and Howell [Baruah et al., 1990] and is also known as processor demand criterion [Buttazzo, 2004]. This abstraction is adapted in the context of this work in order to summarize the computation time demand of a VM:

Definition 13. *The* ***Demand Bound Function*** $dbf_\sigma(V_i, t)$ *denotes for any time interval* $[t_0, t_0 + t]$ *(*$t_0 \geq 0, t > 0$*) the maximum cumulative execution time demand of a periodic task set* $\Omega(V_i)$ *under a scheduling algorithm* σ*.*

The temporal characteristics of a guest system are summarized in terms of this function. It is an interface that contains information about the computational resource demand and is later used in order to design and analyze a system that integrates multiple VMs. The correct integration of multiple real-time systems is only possible if such a temporal interface extends the functional interface of a component.

The demand bound functions for *Earliest Deadline First (EDF)* as an optimal dynamic priority assignment and *Rate Monotonic (RM)* as an optimal static priority assignment [Liu and Layland, 1973b] are for example given as:

$$dbf_{EDF}(V_i, t) = \sum_{\tau_k \in V_i} \left\lfloor \frac{t}{T_k} \right\rfloor \cdot C_k \qquad \text{[Baruah et al., 1990] and}$$

$$dbf_{RM}(V_i, t, \tau_k) = \left\lceil \frac{t}{T_k} \right\rceil C_k + \sum_{\tau_l \in HP_{V_i}(\tau_k)} \left\lceil \frac{t}{T_l} \right\rceil \cdot C_l \qquad \text{[Lehoczky et al., 1989],}$$

where $HP_{V_i}(\tau_k)$ denotes the set of tasks in the task set of V_i with a higher priority than τ_k, i.e., $T_l < T_k$. The sum computes the *interference* caused by the computation times of all jobs of higher-priority tasks released before t. In case of EDF, there are exactly $\left\lfloor \frac{t}{T_k} \right\rfloor$ invocations of task τ_k. In case of RM, $\left\lceil \frac{t}{T_l} \right\rceil$ denotes the worst-case number of interferes.

In case of RM, the demand bound function has to be denoted for each task on its own, since the exact schedulability test that is later performed (see Section 4.3) is based on the computation of the worst-case response time for each task. Since we use the demand bound function as a scheduling interface, information that is required for the schedulability test must be included. It is not possible to derive a single function for the entire task set, as observed by Shin and Lee [Shin and Lee, 2008]. Instead, $dbf_{RM}(V_i, t)$ is the set of all task-specific demand bound functions:

$$dbf_{RM}(V_i, t) = \{dbf_{RM}(V_i, t, \tau_k) | \tau_k \in \Omega(V_i)\} \quad .$$

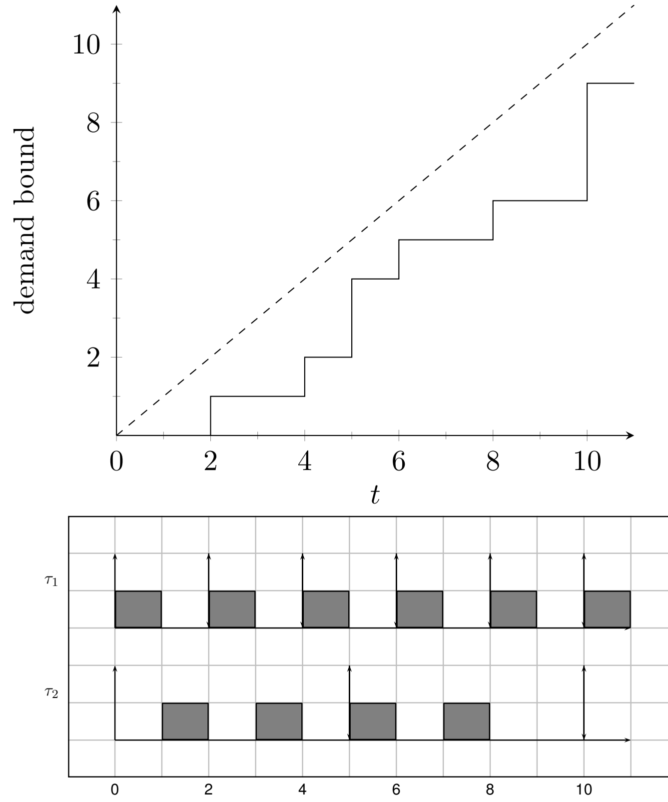


Figure 4.2: Demand bound function for two EDF-scheduled tasks $\tau_1 = (T = 2, C = 1), \tau_2 = (T = 5, C = 2)$ (the dashed line depicts the schedulability bound on a dedicated processor)

Figure 4.2 depicts schedule and the demand bound function for an exemplary task set with two tasks, scheduled by EDF. In case of EDF, a task set is schedulable if the sum of the demand bound functions $\text{dbf}(t)$ of all tasks is smaller than or equal to t for all $t \geq 0$, illustrated by the dashed line with slope 1 through the origin.

Summing up, a virtual machine is characterized by a set of 6-tuples, in which each tuple defines a mode ($M_1, ..., M_m$ indicate different modes of the enabled VM, off refers to the disabled mode):

$$
\begin{aligned}
V_i \stackrel{\text{def}}{=} \{V_i^{off} &= (\Omega(V_i), \sigma(V_i), dbf_\sigma(V_i, t), \chi_i, U_{lax}(V_i)^{off} = 0, w(V_i)^{off} = 0),\\
V_i^{M_1} &= (\Omega(V_i), \sigma(V_i), dbf_\sigma(V_i, t), \chi_i, U_{lax}(V_i)^{M_1}, w(V_i)^{M_1}),\\
V_i^{M_2} &= (\Omega(V_i), \sigma(V_i), dbf_\sigma(V_i, t), \chi_i, U_{lax}(V_i)^{M_2}, w(V_i)^{M_2}),\\
&\vdots\\
V_i^{M_m} &= (\Omega(V_i), \sigma(V_i), dbf_\sigma(V_i, t), \chi_i, U_{lax}(V_i)^{M_m}, w(V_i)^{M_m})\} \qquad \text{, where}
\end{aligned}
$$

- $\Omega(V_i)$ is the task set,

- $\sigma(V_i)$ is the task scheduler,
- $dbf_\sigma(V_i, t)$ is the demand bound function,
- $\chi_i \in \{LO, HI\}$ is the criticality level,
- $U_{lax}(V_i) \in [0, 1]$ is the utilization laxity,
- $w(V_i) \in [0, 1]$ is a weight.

U_{lax} and w differ for each mode (in addition, $C^{off} = 0$ and $w^{off} = 0$), the other attributes are mode-independent. This model is the scheduling interface of the virtual machine, which has to be known to the system designer for the correct integration of the guest system.

Summary. A virtual machine is modeled as a task set and the scheduling policy applied to schedule it. A demand bound function summarizes the computation time demand and is used as an interface for temporal requirements. The virtual machine model derives many characteristics from the task model: it includes modes, a criticality level, varying execution times, and an elastic utilization defined by a utilization minimum and a utilization laxity.

4.2 Resource Model

Target platforms are homogeneous multicore systems of m identical cores with equal computational power $P = \{P_1, P_2, ..., P_m\}$, also know as *identical parallel machines* [Funk et al., 2001]. This implies that each task has the same execution time and utilization on each processor core. A *virtual processor* is a representation of a share of the physical processor core to the VM. A dedicated virtual processor is created for each VM. It provides a lower computational bandwidth than the physical processor core to allow a mapping of multiple virtual processors onto a single physical core. If so, a continuous progress of multiple VMs cannot be achieved in practice, but approximated. (It might provide the same bandwidth at best, in case of the execution of a VM on an exclusive processor core.)

The *periodic resource model* $\Gamma(\Pi,\Theta)$ [Almeida and Pedreiras, 2004, Shin and Lee, 2003, Saewong et al., 2002, Lipari and Bini, 2003] provides a formal abstraction of the computational power supplied by a virtual processor. According to this model, a resource can at most execute for a budget of Θ time units (periodic allocation time)

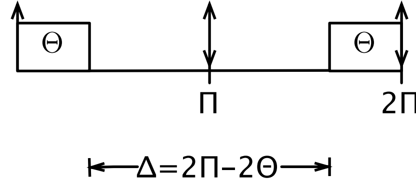


Figure 4.3: Service delay of a periodic resource $\Gamma = (\Pi, \Theta)$

every period Π ($0 < \Theta \leq \Pi$). The resource capacity, i.e., bandwidth of the periodic resource, is defined as Θ/Π. A continuous supply is modeled by $\Gamma(\Pi, \Pi)$.

Note that this model, which characterizes a partitioned resource by a periodic behavior, is actually based on the classic Liu and Layland periodic task model [Liu and Layland, 1973b]: the virtual processor executes a workload with a periodic timing behavior (set of periodic tasks) and in order to satisfy this periodic timing requirement derives itself a periodic behavior [Shin and Lee, 2003].

The minimum computation time allocation that a periodic resource provides is specified in terms of the *supply bound function sbf* [Shin and Lee, 2008]:

Definition 14. *The* ***Supply Bound Function*** $sbf_\Gamma(t)$ *denotes for any time interval* $[t_0, t_0 + t]$ *(*$t_0 \geq 0, t > 0$*) the minimum cumulative computation time allocation of a periodic resource* Γ*:*

$$\begin{aligned} sbf_\Gamma(t) &= \left\lfloor \frac{t - (\Pi - \Theta)}{\Pi} \right\rfloor \cdot \Theta + \epsilon_s, \text{ where} \\ \epsilon_s &= \max\left(t - 2(\Pi - \Theta) - \Pi \left\lfloor \frac{t - (\Pi - \Theta)}{\Pi} \right\rfloor, 0\right). \end{aligned} \tag{4.1}$$

The supply is calculated as the sum of complete periods of the periodic resource within t and the minimum fraction of the last period that overlaps with t. This minimum occurs if the supply is delayed to the very end of the period.

The *service delay* $\Delta(\Gamma)$ specifies the maximum period of time the associated VM may have to wait before receiving computational service by the periodic resource [Lipari and Bini, 2003], as illustrated in Figure 4.3:

$$\Delta(\Gamma) = 2 \cdot (\Pi - \Theta) \tag{4.2}$$

4.3 Schedulability Analysis

The hypervisor-based execution of multiple real-time guest systems has to guarantee that the processor cores are shared in a manner that allows all guests to meet their

real-time constraints. How can we check based on the introduced models for workload and resource whether all tasks of all guest systems can be completed according to their specified constraints?

The maximum cumulative computation time demand of a guest system V_i applying task scheduling policy σ is modeled by the demand bound function $dbf_\sigma(V_i, t)$. The minimum cumulative computation time allocation of the processor share that is provided to a guest is modeled as a periodic resource Γ_i and given by the supply bound function $sbf_{\Gamma_i}(t)$. The comparison of bounded demand of a VM and bounded supply by the associated virtual processor realizes the schedulability analysis [Shin and Lee, 2008]:

$$\forall t : dbf_\sigma(V_i, t) \leq sbf_{\Gamma_i}(t) \tag{4.3}$$

The demand bound is given and it is undesired to modify task set or task scheduling to change it. The periodic resource, however, and consequently the supply bound function $sbf_{\Gamma_i}(t)$ can be designed to enforce schedulability of the task set by forcing it to be at least as high as the demand bound function $dbf(V_i, t)$ for all t.

Theorem 4.3.1. *Assume a set of n virtual machines $V = \{V_1, ..., V_n\}$ with the computation time demand given in terms of demand bound functions $dbf_{\sigma_i}(V_i, t)$ to be executed on a shared processor core with the VM-specific computation time allocation given in terms of supply bound functions $sbf_{\Gamma_i}(t)$. The set V is schedulable, if, and only if,*

$$\forall V_i : \forall t : dbf_\sigma(V_i, t) \leq sbf_{\Gamma_i}(t)$$

and

$$\forall t : \sum_{i=1}^{n} sbf_{\Gamma_i}(t) \leq t$$

Shin and Lee have proven Equation 4.3 for EDF and RM and that it is sufficient to test it for the hyperperiod $\forall t : 0 < t \leq LCM_{V_i}$ in case of EDF (where LCM_{V_i} is the least common multiple of the task periods of $\Omega(V_i)$) [Shin and Lee, 2008]. In case of RM, they identified the following condition as necessary and sufficient:

$$\forall \tau_j \in V_i : \exists t_k \in [0, T_j] : dbf_{RM}(V_i, t_k, j) \leq sbf_{\Gamma_i}(t_k)$$

Figure 4.4 illustrates the schedulability analysis. Depicted are the demand bound function of a virtual machine with two tasks scheduled by EDF and the supply bound function of a periodic resource. Since the demand bound is less than the supply bound for the entire hyperperiod, the VM is schedulable.

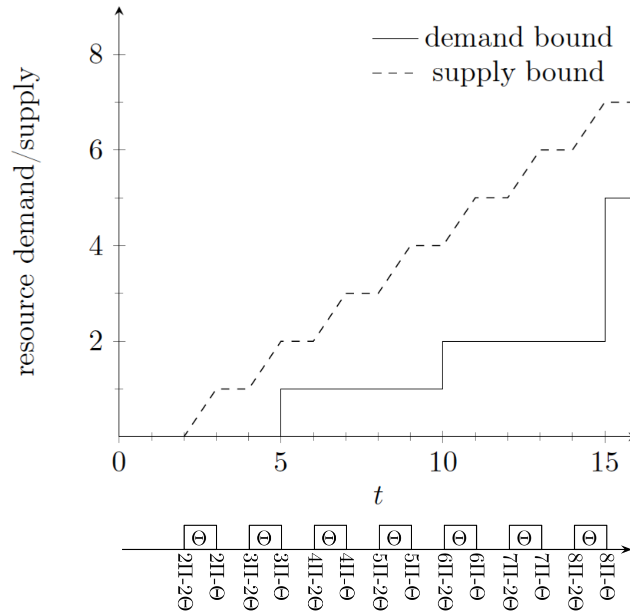


Figure 4.4: Schedulability analysis by comparing demand bound function and supply bound function. Example: two EDF-scheduled tasks $\tau_1 = (T = 5, C = 1), \tau_2 = (T = 15, C = 2)$ and periodic resource $\Gamma = (\Pi = 2, \Theta = 1)$

4.4 Suitability of the Model

Referring back to the motivational application example in Section 1.3, the target systems were defined by the following characteristics:

C1 coexistence of independently developed guest systems of operating system and application tasks,

C2 coexistence of guests of different criticality levels, especially safety-critical (e.g., driver assistance systems) and non-critical (e.g., infotainment systems),

C3 coexistence of guests of different resource requirement characteristics, especially hard real-time and QoS-driven,

C4 existence of guests with real-time requirements, which benefit from additional resource allocations (e.g., computer vision systems),

C5 guests with multiple operational modes, incl. deactivation,

C6 guest with varying execution time demand.

The introduced workload model meets the requirements of such systems. A requirement derived from **C1** is the possibility to analyze the schedulability of a guest

system independently of other guests. As introduced in Section 4.3, the schedulability analysis of a guest is based solely on its own demand bound function and the supply bound function of the associated virtual processor. The integration of a system within a virtual machine next to other systems on top of the hypervisor does not require any insight into the system's internals, especially its task set or applied scheduling algorithm. Required is only the scheduling interface.

Corresponding to **C2**, a criticality level is assigned to each guest and a criticality-aware resource management becomes possible. So far, the model distinguishes only between safety-critical and non-critical guests and might be extended by a dedicated level for each risk class / certification level of the considered industry, e.g., the four Safety Integrity Levels if the standard IEC 61508 or derivatives apply [IEC, 2010] (e.g., railway, process industries), the four Automotive Safety Integrity Levels plus the risk class QM for automotive systems [ISO, 2011], or the five Design Assurance Levels for the avionics domain [RTCA/DO, 2012].

The flexibility of the workload model with a guaranteed minimum bandwidths and a potentially added bandwidth defined by the utilization laxity enables the specification of different resource requirement characteristics as demanded by **C3** and **C4**. Table 4.1 gives a descriptive example with three different guests. A safety-critical control system (V_1) typically has a constant computation time demand, modeled by a positive U_{min}. It is determined based on the worst-case conditions. An additional allocation is useless, which is why U_{lax} is set to zero. A computer vision system (V_2) can benefit from additional computation capacity and process more frames per second, modeled by a positive U_{lax}. If safety-critical (e.g., collision warning system, lane-departure warning system), it demands a certain guaranteed bandwidth $U_{min} > 0$. Finally, V_3 represents a background functionality with a U_{min} of zero and a positive U_{lax}, which makes sense if this system has only a computation demand when the other virtual machines have a very low demand. An example is the seat control, which is only enabled if the car is parked.

The inclusion of varying execution times and the specification of operation modes meet the requirements **C5** and **C6**. The execution time of vision systems varies, for example, since the number of objects to detect varies. Moreover, illumination and weather conditions heavily influence the demand. This could be tackled as well with mode changes. Mode changes are suitable in addition for example for an infotainment system, whose demand depends on the activity of the passengers, or systems that are only enabled in a specific situation, such as the rear view camera system or the control of an adjustable driver seat, which can be disabled when the

Table 4.1: Suitability of the model: exemplary virtual machine set

VM	V_1	V_2	V_3
Description	safety-critical control system	safety-critical vision system	non-critical control system
χ	HI	HI	LO
U_{min}	>0	>0	0
U_{lax}	0	>0	>0

car is not reversing, respectively when the car is not stopped.

4.5 Related Work

Mixed-Criticality Task Model

Vestal introduced a model for tasks with different criticalities [Vestal, 2007]. Main idea is that different criticality levels require different levels of assurance for a task's execution time bound: the higher the criticality level, the less tolerable a deadline miss, for which reason the more conservative (i.e., larger) the bound. In practice, the different required levels of assurance result in the use of different methods and tools for WCET analysis, from measurements during normal operational scenarios at low criticality levels to instruction cycle counting at high levels. He proposes to model a task not by a single WCET, but by a set of alternative WCETs, one per criticality level in the system and determined at different levels of assurance.

Since then, many researchers used Vestal's formalization of the mixed-criticality scheduling problem and provided scheduling algorithms [Baruah et al., 2010a, Li and Baruah, 2010, Mollison et al., 2010, Guan et al., 2011, Huang et al., 2012] and response-time analysis [Baruah and Fohler, 2011]. Su and Zhu proposed an elastic mixed-criticality task model with variable periods [Su and Zhu, 2013]. Anderson et al. presented the first work on server-based mixed-criticality multicore scheduling [Anderson et al., 2009]. Petters et al. addressed temporal isolation of subsystems for mixed-criticality systems [Petters et al., 2009].

Our work is not based on this model, since according to the use case of consolidating certified systems, the system designer has only one WCET per scheduling entity, namely the one that was used in the certification process. It is unrealistic to belatedly determine the WCETs corresponding to other criticality levels.

De Niz et al. introduced an alternative mixed-criticality model for tasks that may overrun [de Niz et al., 2009]. They defined the *criticality inversion problem*: a

high-criticality task that was overrunning its nominal WCET is stopped to schedule a low-criticality task with a higher scheduling priority, making the high-criticality task miss its deadline. Two execution times are assigned to each task: the WCET under non-overloaded conditions and an overload execution budget. In case of an overload scenario, high-criticality tasks are protected by transferring utilization from lower-criticality tasks to higher-criticality tasks. This model is well-suited for the protection of critical tasks in overload situations, however, not flexible enough for the adaptive resource management that is the research goal of this thesis.

Elastic Task Models

Kuo and Mok provided a framework for tasks with varying timing requirements [Kuo and Mok, 1991]. Their model defines an adjustable periodic process as a set of pairs of WCET and period. They address the schedulability of such processes by a preemptive fixed priority scheduler and present an algorithm for the selection of a configuration that picks one pair for each process.

Buttazzo et al. observed that the classic periodic real-time task model is hardly applicable to multimedia or adaptive control systems due to its lack of flexibility [Buttazzo et al., 1999,Buttazzo et al., 2002]. Their proposal replaces the fixed task period by varying periods, each associated with a different quality of service. A task itself can intentionally change its period and the other tasks can adapt to sustain schedulability. An elastic coefficient specifies the flexibility of the task to vary its period within a defined range. A change is accepted only if there exists a feasible schedule.

Su and Zhu proposed an elastic task model with variable periods that is based on Vestal's mixed-criticality model [Su and Zhu, 2013]. Assuming two criticality levels, high-criticality tasks have a single period. Low-criticality tasks have a maximum period that is related to a minimum quality of service, however, this period can be reduced at runtime to provide a higher service level if high-criticality tasks did not need their worst-case utilizations.

In contrast to these models, we vary the utilization of tasks and VMs not by flexible periods, but by flexible execution time allocations.

Multi-Mode Models

Mode change protocols for real-time systems have been classified in a survey by Real and Crespo [Real and Crespo, 2004]. Some approaches define modes by varying periods, for example [Buttazzo et al., 1999] or [Lee et al., 1996]. In the context of this work, modes are characterized by differing desired utilization, but equal period.

Phan, Lee, and Sokolsky introduced a model for multiple multi-mode applications under a hierarchical scheduling policy [Phan et al., 2010]. In contrast to this thesis, their work builds on stream-based task models that specify the resource requirements in terms of service functions. It models each application as an automaton with a finite set of modes that differ regarding the set of active tasks and/or the scheduling policy. One could consider our workload model as a finite automaton with a transition from all modes to all other modes. Their work allows for defining transition relations including signals that trigger the transition. The multi-mode resource interface is a finite state machine and the states abstract the resource requirement of one or more modes. Each state comes with a minimum service function that is required to guarantee schedulability, comparable to our inclusion of the demand bound function.

Oberthür developed the Flexible Resource Manager for self-optimizing real-time systems [Oberthür et al., 2010]. It addresses non-virtualized architectures with a single operating system. A set of profiles representing implementation alternatives with specified transitions is assigned by the developer to each application. An operating system extension is in charge of switching between these profiles at runtime. Their work enables assigning temporarily unused resources to other applications based on profile switches, maintains schedulability, and was very influential for this thesis.

Resource Models

The periodic resource model has been intensively researched. Abstraction of a system in terms of a periodic resource is addressed in [Davis and Burns, 2005] and [Saewong et al., 2002]. [Shin and Lee, 2003] and [Easwaran et al., 2006] studied the minimal dimensioning of periodic resource interfaces. [Almeida and Pedreiras, 2004] used a more realistic task model that incorporates release jitter and synchronization among tasks, for example in the context of accessing shared resources. [Shin et al., 2008a] included as well resource sharing across subsystems and analyze the trade-off between resource locking time and CPU allocation. Lipari and Bini presented a methodology for finding a class of possible parameters for period and budget that guarantee schedulability [Lipari and Bini, 2003]. It uses a sufficient but not necessary response time analysis based on an availability function. Their work includes the minimization of the context switch overhead.

Saewong et al. presented an exact schedulability analysis for RM or deadline-monotonic scheduling of periodic resources and derived utilization bounds [Saewong et al., 2002]. It is based on response time analysis and considers deferrable and sporadic servers for the implementation of the periodic resources. It was extended

by the worst-case response time analysis in [Almeida and Pedreiras, 2004]. [Davis and Burns, 2005] extended as well Saewong et al.'s work by improving the calculation of the worst-case response time and providing exact analysis for periodic servers. In [Harbour and Palencia, 2003], the authors presented a response time analysis for tasks scheduled under EDF within a component, with the component scheduled by a fixed-priority scheduler. Their work includes mutually exclusive synchronization to shared resources.

Matic and Henzinger extended the periodic resource model for interacting tasks, both within a component and across component borders [Matic and Henzinger, 2005]. Data dependencies are specified as task precedence graphs. The authors analyze the trade-off between a low end-to-end latency and tighter resource bounds, resulting in an increased composability. Davis and Burns extended the periodic resource model by mutually exclusive resource sharing within (guest-local) and between components [Davis and Burns, 2006]. Their Hierarchical Stack Resource Policy combines server and task ceiling protocols to bound priority inversion and interference on low priority components caused by overruns. The sharing of any other resource than the processor core is not addressed in this thesis. For a future extension, however, the work of Davis and Burns is highly relevant.

Shin, Easwaran, and Lee extended the periodic resource model to a multiprocessor model [Shin et al., 2008b]. A resource (Π, Θ, m) provides a budget of Θ time units every period Π by at most m processors. They address the dimensioning of a component interface with minimal processor utilization in case of component scheduling based on EDF. With this new model, they analyze cluster-based multiprocessor scheduling. Tasks are statically assigned to processor clusters and globally scheduled within this cluster. Clusters are scheduled at runtime on the multiprocessor. Leontyev and Anderson proposed a similar multiprocessor hierarchical scheduling, which considers next to hard real-time requirements as well soft deadlines [Leontyev and Anderson, 2009]. Bini, Buttazzo, and Bertogna introduced the Multi Supply Function abstraction, which models a parallel machine as a set of virtual processors [Bini et al., 2009]. Each virtual processor is specified in terms of a supply function. Their approach can be applied for arbitrary reservations. A multiprocessor model is not needed in the context of this thesis, since after the static partitioning of VMs to cores, a uniprocessor hierarchical scheduling is applied.

Easwaran, Anand, and Lee introduced the Explicit Deadline Periodic (EDP) resource model, a generalization of the periodic resource model [Easwaran et al., 2007]. As in case of the periodic resource model, Θ resource units are provided every period

Π, but in addition within Δ time units after the start of a period (relative deadline of allocation). A periodic resource model $\Gamma = (\Pi, \Theta)$ is therefore equal to the EDP model $(\Pi, \Theta, \Delta = \Pi)$. Their model was designed for the sporadic task model with explicit relative deadlines ($D \leq T$), whereas the periodic resource model is based on the periodic task model with implicit deadlines ($D = T$). This enables task scheduling according to the deadline monotonic scheduling policy [Leung and Whitehead, 1982], a generalization of RM for tasks with explicit deadlines: tasks obtain a static priority inversely proportional to their relative deadline. In this thesis, the component scheduling is based on RM and the periodic resource model is therefore sufficiently powerful.

As an alternative to the periodic resource model, Feng, Mok, and Chen proposed two resource partition models [Mok et al., 2001]. First, the static resource partition model formalizes a resource that is in a time-multiplexed manner only available at certain time intervals specified by a list, which periodically repeats itself. The availability factor α denotes the sum of the time units of availability within the period divided by the length of the period. Second, the bounded-delay resource partition is specified by the availability factor α and the maximum delay Δ, which specifies the maximum time the task group may have to wait before resource availability. The authors provide a schedulability analysis for a task set that is executed by such a resource partition for both fixed priorities and EDF, but did not address component abstraction. Feng and Mok [Feng and Mok, 2002] and Shin and Lee [Shin and Lee, 2004] extended this work by analysis techniques, component abstraction, and component composition. The model is especially useful for time-slice based scheduling, which is not applied in this thesis.

Lorente et al. added interaction between tasks to a hierarchical scheduling model for component-based real-time systems [Lorente et al., 2006]. A component consists of a set of threads and a local scheduler. A thread is implemented as a sequence of tasks and method calls. Tasks interact through Remote Procedure Calls (RPC), which can be either synchronous or asynchronous. They derive a schedulability analysis for this architecture based on minimum inter-arrival times for consecutive RPCs. This thesis considers only independent VMs.

4.6 Summary

The workload model is based on the classic Liu and Layland periodic real-time task model. A virtual machine is modeled as a set of such tasks and the scheduling policy applied by the guest operating system to schedule it. The actual computation time

demand of tasks and, consequently, of virtual machines might vary from period to period up to a known upper bound. Tasks and virtual machines can be modeled by a utilization laxity that the entity could use effectively in addition to a utilization minimum. Tasks and virtual machines can have several operational modes, which differ regarding behavior and resource demand. A demand bound function denotes the maximum cumulative computation time demand of a virtual machine and is used as a temporal interface in order to design and analyze a system that integrates multiple virtual machines. This workload model meets the requirements of systems that integrate guests of different criticality levels and resource requirement characteristics, especially, if there are guests with hard real-time requirements that benefit from additional resource allocations.

The resource model maps multiple virtual processors to a homogeneous multicore systems. Each virtual processor is a representation of a share of the physical processor and assigned exclusively to a virtual machine. The periodic resource model provides a formalization of the computation time supply of a virtual processor. The supply bound function specifies the minimum cumulative computation time allocation within an interval. The service delay denotes the maximum duration in which a virtual processor does not provide any computation time. Schedulability analysis is performed based on the comparison of demand bound functions and supply bound functions.

Chapter 5

Partitioning

Contents

5.1 Problem Statement 89

5.2 Related Work 90

5.3 Branch-and-Bound Partitioning 93

5.3.1 Pruning & Server Transformation 94

5.3.2 Optimization Goals 99

5.3.3 The Algorithm 100

5.3.4 Example 103

5.4 Evaluation 103

5.5 Summary 108

There are two main approaches in the multiprocessor scheduling domain, as introduced in detail in Section 2.3.1. *Partitioned scheduling* allocates each task statically to a processor core, without migration of tasks among cores. Each job of a task is executed on the same core. In contrast, *global scheduling* permits migration. A global ready queue is used to dynamically map the tasks' jobs to the cores (task-level migration) or it might even be possible that a single job is migrated and executed on different cores (job-level migration) [Davis and Burns, 2010]. Aside from the higher schedulability bounds of global strategies (irrelevant in our case, as we will see in Section 5.3.1), partitioned scheduling has several advantages. It is characterized by a higher offline analyzability, enables to apply well-known uniprocessor scheduling results, and does not introduce a migration overhead [Davis and Burns, 2010].

It is an important observation that the hypervisor-based integration of independently developed and validated systems precludes a global task scheduling. The consolidation of entire software stacks including operating system results in hierarchical scheduling and the hosted guest operating systems schedule their tasks according to their specific task scheduling policies. This is irreconcilable with a scheduling based on a global task ready queue. Task migration is neither desirable (one does not want to split up verified or even certified systems) nor in general technically possible across operating system borders.

In contrast, a dynamic assignment of the entire VMs to the processor cores is possible. On the shared-memory target platform, virtual machine migration across cores is no technical issue. Nevertheless, this work allocates each VM statically to a processor core. A static mapping eases certification significantly, due to the lower run-time complexity, the higher predictability, and the wider experience of system designer and certification authority with uniprocessor scheduling. Run-time scheduling can be performed efficiently in such systems and the overhead of a more complex VM scheduler is avoided. Cache performance is increased in case of a hierarchical cache architecture, e.g., a tree-like cache hierarchy with all cores sharing an L3 cache, two cores sharing an L2 cache and dedicated L1 cache [Calandrino et al., 2007]. Besides (and presumably for the listed reasons), a static solution is the option taken by the AUTOSAR consortium [Navet et al., 2010].

Please note, virtual machine migration is not performed across cores of the same multicore processor. However, migration to another ECU, which is connected via a network, is possible and actually performed as a fault tolerance technique as introduced in Chapter 7.

With these assumptions, the scheduling problem for system virtualization on mul-

ticore platforms consists of two sub-problems:

(i) partitioning: allocation of virtual machines to processor cores,

(ii) uniprocessor hierarchical scheduling on each core.

For sub-problem (ii), Chapter 6 introduces a server-based hierarchical scheduling. This chapter focuses on sub-problem (i). After defining the design problem of mapping VMs with real-time requirements onto a multicore platform, a partitioning algorithm is presented that splits a set of virtual machines into subsets, with each subset being schedulable on a single processor core.

Computation time servers realize the computation time supply to the virtual machines. By transforming their periods to harmonic ones, the core's utilization bound is increased and the number of required cores decreased. An introduced optimization metric realizes a partitioning that considers multiple criticality levels and distributes critical VMs among the cores.

5.1 Problem Statement

The partitioning problem refers to the decision which virtual machine to execute on which of the m processor cores $P_1, ..., P_m$. Since homogeneous multicore processors are considered, the problem is equivalent to subdividing the set of n VMs into m subsets with 0 to n elements.

$$\Xi = \{\Xi_1, \Xi_2, ..., \Xi_m\}, \text{ with } \forall i (1 \leq i \leq m) : 0 \leq |\Xi_i| \leq n, \sum_i |\Xi_i| = n$$

Such a mapping Ξ of virtual machines (equivalent to virtual processors) to physical processor cores is correct if and only if the computation requirements of all virtual processors are met; and by consequence the schedulability of the associated VMs is guaranteed. This implies that the partitioning problem depends on the applied VM scheduling algorithm (sub-problem (ii)), since a set of VMs might be schedulable by a specific scheduling algorithm, but not by another one. In the context of this thesis, the VM scheduling is based on servers, as introduced in Section 2.4.3. Each VM is executed by a dedicated server, characterized by a periodically replenished execution time budget, which implements a virtual processor by providing a guaranteed but limited computation bandwidth.

Next to assuring schedulability of the VM set assigned to a processor core, the partitioning of the VMs focuses on two goals. Minimizing the overall required computation bandwidth is the first goal, since it determines the number of processor

cores required to host the set of VMs. In addition, a distribution of critical VMs among the processor cores is targeted. If VMs of differing criticality share a core, it is possible to protect the critical guest when it overruns its execution time budget (overload situation) by stealing computation time from non-critical guests and there are in general more possibilities to apply an adaptive VM scheduling, since the non-critical VMs can benefit from unused reservations of the critical VMs.

5.2 Related Work

Task Partitioning

Recently, Singh et al. published a survey about the related problem of mapping tasks to multi/many-core systems and categorized into design-time and run-time methodologies, different optimization goals as well as homogeneous and heterogeneous multicore systems [Singh et al., 2013]. Run-time mapping has the advantage of being able to adapt to both dynamic workloads (addition/termination of applications at runtime, varying parameters) and dynamic platforms (varying resource availability caused for example by enabling/disabling cores or hardware failures). Drawback is the need for a small execution time of the mapping algorithm in order to continue the regular operation of the tasks as soon as possible. This thesis addresses design-time mapping: information about the virtual machines is available at design time and the execution time of the algorithm is not exceptionally restricted.

Typical optimization goals are performance and timing-related aspects, especially the guarantee of real-time requirements, but also a minimization of latency or a maximization of throughput. Other goals are an optimization of the energy consumption or the reliability, for example by exploiting redundancy or producing mappings with lower peak temperatures, which has a positive effect on a chip's reliability. The goal of our work is primarily the guarantee of real-time requirements, and among mappings that provide this guarantee the minimization of the number of required cores respectively the maximization of the criticality distribution. Our work addresses only homogeneous processors: each core offers the same computing power, which implies that each virtual machine and task has the same execution speed and utilization on each processor core.

The partitioning of a periodic task set upon homogeneous multiprocessor platforms has been extensively studied, both theoretically and empirically, as [Carpenter et al., 2004] or [Davis and Burns, 2010] list. Since finding an optimal assignment of tasks to processors is a problem that is NP-hard in the strong sense [Garey and John-

son, 1979], task set partitioning methodologies typically apply Bin-Packing Heuristics [Carpenter et al., 2004, Davis and Burns, 2010], Simulated Annealing [Orsila et al., 2007], or Genetic Algorithms [Choi et al., 2012]. Dhall and Liu applied in initial work on the topic the heuristic bin-packing algorithms First Fit and Best Fit [Dhall and Liu, 1978]. In contrast to these heuristic approaches, this thesis proposes an algorithm that guarantees to find an optimal solution and that is nevertheless practical due to the low number of VMs.

Lopez et al. analyzed the utilization bound for partitioned EDF scheduling of independent tasks on homogeneous multiprocessor systems and examined different allocation algorithms [Lopez et al., 2000]. Shin et al. proposed cluster-based scheduling to improve utilization bounds [Shin et al., 2008b]. Each task is statically assigned to a processor cluster, tasks in each cluster are globally EDF-scheduled among themselves, and clusters in turn are scheduled by EDF. This leads to a hierarchical scheduling. In contrast, our work assigns the scheduled entities statically to cores.

Mixed-Criticality Task Partitioning

Kelly at al. proposed bin-packing algorithms for the partitioning of fixed-priority mixed-criticality real-time task sets [Kelly et al., 2011]. They experimentally compared offline task ordering according to decreasing utilization and decreasing criticality (with decreasing utilization as the ordering criteria among tasks of the same criticality) and observed that the latter performed better in terms of schedulability of random task sets. They also investigated the relative importance of task allocation and priority assignment and conclude that the priority assignment is more important. Giannopoulou et al. proposed a simulated annealing-based technique that combines mixed-criticality task partitioning and static mapping of memory blocks in order to account for the effects of shared memory access interference [Giannopoulou et al., 2014]. These works use a common mixed-criticality model, characterized by an assignment of multiple WCETs, one per criticality level in the system, which is however not used in the context of this thesis (see Section 4.5).

Branch-and-Bound Partitioning

Closest to this work and very influential, Buttazzo et al. proposed a branch-and-bound algorithm for partitioning a task set with precedence constraints, in order to minimize the required overall computational bandwidth [Buttazzo et al., 2011]. Parallel real-time applications are partitioned into a set of sequential flows so that

the overall computational bandwidth or the number of processors is minimized. Not motivated by hypervisor-based virtualization, but by portability on different hardware architectures, the authors use a virtual platform, which can be implemented by any resource reservation mechanism (incl. periodic resources). They use Mok et al.'s bounded-delay time partition model [Mok et al., 2001] for the virtual processors (see Section 4.5) and address how to select optimal parameters. Since they observed that the run-time overhead of the branch-and-bound algorithm is too high for more than 15-20 tasks, they developed as well polynomial-time heuristic algorithms. Peng and Shin presented a branch-and-bound algorithm in order to partition a set of communicating tasks in a distributed system [Peng and Shin, 1997]. Our algorithm follows the same paradigm, but for different objectives.

Partitioning of Virtual Entities

Lin et al. briefly touched on the mapping of server-based virtual cores to physical cores [Lin et al., 2010], but focused on energy-efficient designs. They proposed a task mapping based on dynamic programming as a design time solution, plus, a simple runtime solution (mapping to the core with largest or least remaining utilization). Bobroff et al. introduced a VM mapping approach for non-real-time systems, which uses a first-fit bin packing heuristic [Bobroff et al., 2007].

Period Transformation

Lopez et al. observed that ordering tasks according to decreasing utilization prior to the partitioning proves helpful [Lopez et al., 2003], a technique applied in our work as well. Burchard et al. examined the impact of task period relationships on the utilization for RM-scheduled systems and proposed heuristics to assign tasks that fulfill these relationships to the same processor [Burchard et al., 1995]. Similarly, Lauzac et al. determined ratios of the minimum and maximum period of a task set that lead to a high utilization and derived an admission control and a partitioning scheme for RM-scheduled systems [Lauzac et al., 1998]. Instead of looking for favorable period relationships, we exploit the freedom of server design to *create* favorable period relationships.

Easwaran et al. proposed algorithms for abstracting periodic resource models for components that apply RM or EDF [Easwaran et al., 2006]. For each component, a set of periodic resource models differing regarding period is generated. Out of these sets, one periodic resource model is selected for each component so that all components of the system have the same period. This is motivated by a minimization

of the collective computation time requirements (least resource demand). As this leads to all components having the same priority under both RM and EDF, arbitrary priorities are assigned. In contrast, Shin and Lee let each component itself choose its period [Shin and Lee, 2003], without restrictions. Our work could be considered to be in the middle of these two extremes: instead of unrestricted individual periods, we select periods with a specific relation in order to use the processor more efficiently, as Easwaran et al. do. Harmonic periods are however less restrictive than equal periods. A single period can in many cases result in a much smaller period for components than required by their reactivity requirements, and consequently, in a higher context switching overhead.

5.3 Branch-and-Bound Partitioning

The partitioning problem is equivalent to bin-packing, as for example Baruah [Baruah, 2004] has shown for the task partitioning problem by transformation from 3-Partition. The VMs are the objects to pack with size determined by their utilization factors. The bins are processor cores with a computation capacity value that is dependent on the applied VM scheduler. The bin-packing problem is known to be intractable (NP-hard in the strong sense) [Garey and Johnson, 1979] and the research focused on approximation algorithms (see [Coffman et al., 1996] for a survey).

Common task set partitioning schemes apply Bin-Packing Heuristics or Integer-Linear-Programming (ILP) approaches in order to provide an efficient algorithm [Carpenter et al., 2004, Davis and Burns, 2010]. In the context of this work, however, the number of objects is comparatively small and the partitioning algorithm is to be run offline and does not have to be executed on the embedded processor. Therefore, instead of applying an approximation algorithm, the here presented algorithm performs a systematic enumeration of all candidate solutions following the branch-and-bound paradigm [Land and Doig, 1960], a general algorithm for finding optimal solutions of discrete optimization problems.

The systematic enumeration is depicted in Figure 5.1 as a state-space tree. The problem is solved VM by VM, that is to say that first a partitioning for just V_1 is computed, subsequently, a partitioning for V_1 and V_2, and so on. Each node represents a partial solution (or a final solution if it includes all VMs). Branches represent all possible next steps, each leading to a new partial or final solution. The recursion of a branch stops when the solution includes all VMs (in this case this solution is compared to the so far known best solution) or when the partitioning mapped a set of VMs to a core that is not schedulable. This second case is called *pruning*: a

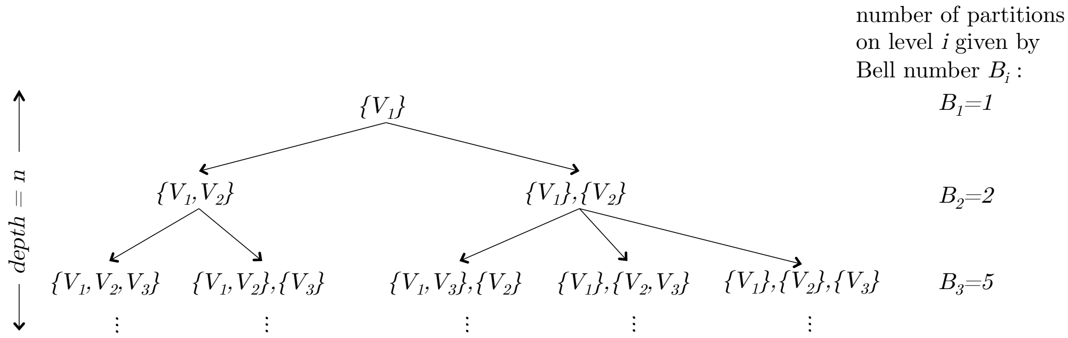


Figure 5.1: Tree-based visualization of the systematic enumeration of candidate solutions — Example: on level 2, V_2 might either be added to the same partition as V_1 or a second partition is created.

non-valid partial solution cannot be turned into a valid one (an unschedulable VM set cannot become schedulable by adding a VM), for which reason such a branch can be safely discarded from the search. The depth of the search tree is equal to the number of VMs n and the number of partitions on level $i \geq 1$ is given by the Bell number B_i (1, 2, 5, 15, 52, 203, 877, ...). We argue that these numbers of partitions are large enough to justify an automated solution, and small enough to not settle for a heuristic solution.

5.3.1 Pruning & Server Transformation

It is not sufficient that the partitioning algorithm produces a mapping that assigns a schedulable set of virtual machines to each core (i.e., there exists at least one algorithm that can produce a feasible schedule, see Section 2.1.2). In fact, practicability requires that the resulting partitions are schedulable by a given virtual machine scheduling algorithm. The partitioning algorithm has therefore to be specific to the applied VM scheduling algorithm and guarantee that exactly this scheduling algorithm produces a feasible schedule. As introduced in the next chapter, a VM scheduling according to the rate monotonic (RM) policy is applied. Consequently, a branch of the tree can be pruned as it cannot lead to a valid solution if and only if the following equation is not fulfilled:

$$\sum_{i=1}^{n} U_{min}(V_i) \leq U_{lub} \leq U_R$$

Using a rate monotonic server scheduler with its static priority assignment has the significant advantage of a low runtime overhead and a high analyzability at

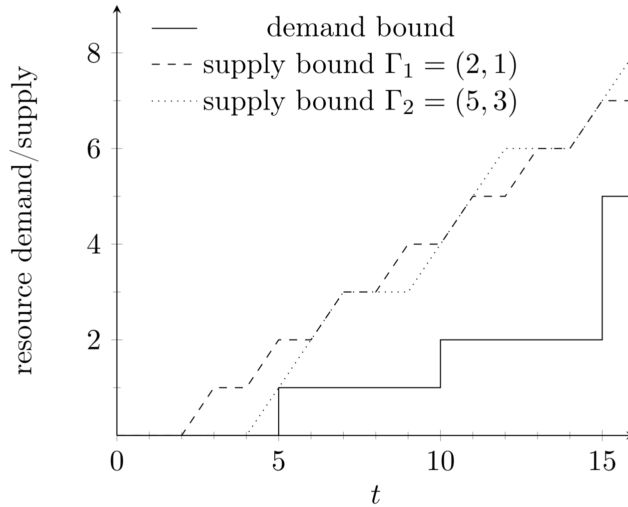


Figure 5.2: Existence of multiple periodic resources. Example: two EDF-scheduled tasks $\tau_1 = (T = 5, C = 1), \tau_2 = (T = 15, C = 2)$ and two periodic resources $\Gamma_1 = (\Pi = 2, \Theta = 1)$, $\Gamma_2 = (\Pi = 5, \Theta = 3)$, both guarantee schedulability

design time, which supports certification. The major drawback is the low utilization bound, the minimum of the utilization factors over all server sets that fully utilize the processor core: $U_{lub} = n(2^{1/n} - 1)$ [Liu and Layland, 1973b]. Note that the least upper bound defines a sufficient and not a necessary condition. All of the server sets that pass the test are in fact schedulable, but not all server sets that fail the test are actually unschedulable: a set of servers with a utilization greater than U_{lub} and less than or equal to one might be schedulable, which can be analyzed with a more complex schedulability test (see [Fidge, 1998] for a survey on schedulability tests). Lehoczky, Sha and Ding obtained by simulations an average utilization bound of 0.88 for random task sets [Lehoczky et al., 1989].

This low utilization bound, however, can be tackled by the freedom of design regarding the periodic resources. As introduced in Section 4.3, there is not a unique periodic resource dimensioning that guarantees schedulability for a given workload. As an example, the supply bound functions of two possible periodic resources that are both appropriate to execute a given workload are depicted in Figure 5.2.

We exploit this design freedom by choosing harmonic periods. If the period of each server is an exact multiple of the periods of every other server with a shorter period, rate monotonic can fully utilize the core [Kuo and Mok, 1991]: $U_{lub} = U_R = 1$. Harmonic periods are in fact feasible, since server design approaches such as [Shin and Lee, 2008] and [Almeida and Pedreiras, 2004] allow the server period to be chosen within a range of possible values without impact on the schedulability of the internal

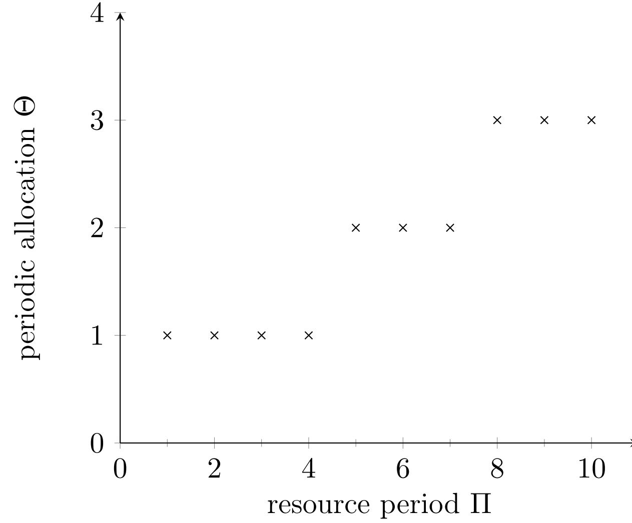


Figure 5.3: Schedulable region of a periodic resource. Example: two EDF-scheduled tasks $\tau_1 = (T = 50, C = 7), \tau_2 = (T = 75, C = 9)$. Integral resource allocation minimums for $1 \leq \Pi \leq 10$

task set, in particular without violating the largest possible VM service delay. Figure 5.3 depicts an exemplary solution space for the selection of period and capacity of a periodic resource. It shows the minimum periodic allocation Θ in integer values for periods up to ten (integer values since a scheduler is limited by the granularity of the timer and can execute an entity only for a multiple of this minimum, which is assumed to be 1).

Shin and Lee addressed the problem of deriving resource period Π and capacity Θ in order to guarantee the schedulability for a given workload [Shin and Lee, 2003]. They define the optimal *periodic capacity bound* $PCB_{V_i}(\Pi, A)$ as a number such that the workload V_i is schedulable by the algorithm A if $PCB_{V_i}(\Pi, A) \leq \Theta/\Pi$.

The PCBs for EDF and RM are defined as [Shin and Lee, 2003]:

$$PCB_W(\Pi, A) = \frac{\Theta_A^+}{\Pi} \quad , where \tag{5.1}$$

$$\Theta_{EDF}^+ = \max_{0<t\leq 2\cdot L} \left(\frac{\sqrt{(t-2\Pi)^2 + 8\Pi dbf_W(t)} - (t-2\Pi)}{4} \right) \tag{5.2}$$

$$L = lcm(\{T_i | \tau_i \in W\}), \text{ least common multiple}$$

$$\Theta_{RM}^+ = \max_{\forall \tau_i \in W} \left(\frac{-(T_i - 2\Pi) + \sqrt{(T_i - 2\Pi)^2 + 8\Pi I_i}}{4} \right) \tag{5.3}$$

$$I_i = C_i + \sum_{\forall \tau_k \in \{\tau_k | \tau_k \in W \wedge T_k < T_i\}} \left\lceil \frac{T_i}{T_k} \cdot C_k \right\rceil$$

Note that the + indicates that Θ^+ is the smallest possible Θ and any larger Θ guarantees the schedulability as well. Shin and Lee assume Π as given and the parameters of Γ_i follow immediately as $\Gamma_i(\Pi_i, \Theta_i = \Pi_i \cdot PCB_W(\Pi_i, A))$.

The question of how to derive Π_i for a given VM V_i remains. We assume that the largest possible service delay Δ_i^{max} is known for a VM. It is defined by the largest affordable blackout phase, also known as dead interval [Almeida et al., 2002], the largest acceptable interval without resource supply. If we choose a period Π_i and compute with Equation 5.2 or respectively Equation 5.3 Θ_i^+, then we can compute with Equation 4.2 the service delay Δ for this resource dimensioning and check whether the service delay of the periodic resource is less than or equal to Δ_i^{max}. Only if this is the case, the periodic resource $(\Pi_i, \Theta_i = \Theta_i^+)$ can be accepted to implement the virtual processor for VM V_i.

A small period causes a high number of VM context switches. A large period results in a large service delay and hence in a low reactivity. In general, a large period is desired, because of the high costs of a VM context switch. Initial resource parameters were computed for all VMs based on the maximum possible periods, derived from the reactivity requirements of the guest system, i.e., the largest possible VM service delay Δ_i^{max}. These resource parameters are denoted as Π_i^{opt} and Θ_i^{opt}.

Algorithm 1 accomplishes the transformation of the resource periods to harmonic ones. First, the set of periodic resources $\Gamma = \{\Gamma_1, \Gamma_2, ..., \Gamma_n\}$ is sorted according to increasing Π_i^{opt}. The smallest period is taken as a pivot element (line 3). Resource per resource, a new period is assigned as a multiple of the so far selected maximum period Π^{max}. Since the resources were sorted according to increasing Π_i^{opt}, the assigned period in each iteration of the loop is at least as large as the one that was set in the previous iteration. For this reason, we can set Π^{max} to the current Π_i in each iteration (line 6). The corresponding resource capacity Θ_i is set to the smallest possible value that guarantees schedulability (depending on the applied task scheduler).

In the following, two important properties of this algorithm are proven.

Lemma 5.3.1. *Algorithm 1 produces for each resource Γ_i a period that is less than or equal to Π_i^{opt}.*

Proof. By contradiction. Assume that the algorithm computed a resource period $\Pi_i > \Pi_i^{opt}$. By consequence, $\lfloor \Pi_i^{opt} / \Pi^{max} \rfloor \cdot \Pi^{max} > \Pi_i^{opt}$, which requires that $\lfloor \Pi_i^{opt} / \Pi^{max} \rfloor > \Pi_i^{opt} / \Pi^{max}$. This is a contradiction to the definition of the floor

Algorithm 1 Harmonization

1: **function** HARMONIZE(Γ)
2: $\Gamma \leftarrow Sort(\Gamma, increasing\ \Pi^{opt})$
3: $\Pi^{max} \leftarrow \Pi_1^{opt}$ ▷ maximum selected period
4: **for all** $\Gamma_i \in \Gamma$ **do**
5: $\Pi_i \leftarrow \lfloor \Pi_i^{opt} / \Pi^{max} \rfloor \cdot \Pi^{max}$
6: $\Pi^{max} \leftarrow \Pi_i$
7: $\Theta_i \leftarrow \Pi_i \cdot PCB_W(\Pi_i, A)$ ▷ according to Eq. 5.1

function, which maps a real number to the largest integer not greater than the input value ($\lfloor x \rfloor = \max\{k \in \mathbb{Z} \mid k \leq x\}$). □

This lemma describes an important characteristic of the algorithm, since Π_i^{opt} was derived from the largest possible VM service delay Δ_i^{max}. The algorithm cannot generate larger resource periods and, therefore, guarantees that the reactivity requirements of the VM are met.

Lemma 5.3.2. *Algorithm 1 produces harmonic periods for all n periodic resources: the periods of all resources pairwise divide each other.*

Proof. By induction and contradiction. Assume that the algorithm computed a resource period that does not divide a larger period: $\exists \Pi_i, \Pi_j > \Pi_i$ with $\Pi_i \nmid \Pi_j$. This implies that there is no integer d with $d \cdot \Pi_i = \Pi_j$.

Base case ($n = 2$): $\Pi_1 = \Pi_1^{opt}$, since Π_1^{opt} is the maximum selected period at the beginning (Line 3), leading to $\Pi_1 = \lfloor \Pi_1^{opt} / \Pi_1^{opt} \rfloor \cdot \Pi_1^{opt} = \Pi_1^{opt}$ (Line 5). The period Π_2 follows as: $\Pi_2 = \lfloor \Pi_2^{opt} / \Pi_1^{opt} \rfloor \cdot \Pi_1^{opt}$. By consequence, $d = \lfloor \Pi_2^{opt} / \Pi_1^{opt} \rfloor \geq 1$, which is a contradiction to the assumption that there is no such d.

Inductive step ($n \rightsquigarrow n+1$): The periods $\{\Pi_1, \Pi_2, ..., \Pi_n\}$ pairwise divide each other, with $\Pi^{max} = \Pi_n$ being the largest so far selected period. According to line 5, by replacing Π^{max} it follows: $\Pi_{n+1} = \lfloor \Pi_{n+1}^{opt} / \Pi_n \rfloor \cdot \Pi_n$. Based on Lemma 5.3.1, we know that Π_n is less than or equal to Π_n^{opt}, which in turn is less than or equal to Π_{n+1}^{opt} (sorting of Γ). $\Pi_n \leq \Pi_{n+1}^{opt}$ leads to the term $\lfloor \Pi_{n+1}^{opt} / \Pi_n \rfloor$ being an integer equal or greater than one. This implies that Π_n divides Π_{n+1}, so there is a d_2 with $\Pi_{n+1} = d_2 \cdot \Pi_n$. We know that the periods $\{\Pi_1, \Pi_2, ..., \Pi_n\}$ pairwise divide each other, so for each period Π_i, there is a positive integer d_i with $\Pi_n = d_i \cdot \Pi_i$. This, finally, implies that each period Π_i divides Π_{n+1}, with $d = d_i \cdot d_2$, which, again, is a contradiction to the assumption that there is no such d. □

The transformation of the periodic resources to harmonic periods results in a schedulability bound U_{lub} of one. If the sum of the utilizations of all VMs mapped

to a core exceeds one, the branch is pruned (in accordance with Equation 6.1).

5.3.2 Optimization Goals

The algorithm performs a systematic enumeration of all candidate solutions following the branch-and-bound paradigm. Candidates are compared according to two optimization goals. Minimizing the number of cores required to host the set of VMs is the basic optimization goal. In addition, a distribution of critical VMs ($\chi = HI$) among the cores is targeted. In the best case, each critical VM is mapped to a dedicated core, which is potentially shared with non-critical VMs ($\chi = LO$), but not with other critical VMs.

This second optimization goal is motivated by two different aspects: resource utilization and safety. First, if VMs of differing criticality share a core, there are in many cases more possibilities to apply an adaptive scheduling. Critical VMs are typically characterized by more pessimistic resource reservations that are underrun significantly and often at runtime, whereas non-critical VMs are often quality-of-service driven and can benefit from additional computation time out of the unused share of the critical VMs. Chapter 6 introduces an adaptive bandwidth management that exploits the co-existence of critical and non-critical VMs on the same core.

Second, if the critical VMs are distributed, it becomes possible to protect a critical VM in case of an unforeseen run-time overload at the expense of non-critical VMs. The *Criticality Inversion Problem*, defined by de Niz et al. [de Niz et al., 2009] (and here transferred to VM scheduling), occurs if a critical VM overruns its execution time budget and is stopped to allow a non-critical VM to run, resulting in a deadline miss for a task of the critical VM. The authors observed that the WCET is difficult to calculate for many mixed-criticality systems, for example in case of an obstacle avoidance algorithm whose execution time depends on the number of objects to detect. Instead of a criticality-unaware scheduling, according to which a non-critical VM can have a higher scheduling priority than a critical VM, and by definition of criticality as severity of failure, it is more appropriate to continue the execution of the critical VM. This can be done for highly utilized cores by stealing execution time from non-critical VMs. This might lead to a violation of timing requirements of the non-critical VMs, but if only the requirements of either a critical VM or a non-critical VM can be satisfied, the former should be induced.

One might argue that a stronger isolation between critical and non-critical VMs can be achieved if all critical VMs are concentrated on dedicated cores. In this case as well, however, the hypervisor has to implement and guarantee isolation between

all VMs and we assume this as a prerequisite, realized for example by the hypervisor Proteus and the scheduling approach of Chapter 6. In case of an overload situation, it is not possible to protect the execution of a critical VM if it shares the core only with other critical VMs, since we cannot steal execution time from critical VMs. Moreover, the execution of different criticality levels on dedicated cores results in at least as many cores as criticality levels. Automotive industry's differentiation in five safety integrity levels for example could lead to a minimum of five cores, which in addition are possibly under-utilized due to the very differing resource requirements of the software of the different levels.

The second goal is defined by the metric $CriticalityDistribution$:

Definition. The *CriticalityDistribution* Z denotes for a partitioning Ξ the distribution of the $n_{crit} \leq n$ critical VMs among the m cores:

$$Z(\Xi) = \frac{\sum_{i=1}^{m} \zeta(\Xi_i)}{n_{crit}} \quad \text{, with} \tag{5.4}$$

$$\zeta(\Xi_i) = \begin{cases} 1, & \text{if } \exists V_j \in \Xi_i \colon \chi(V_j) = HI, \\ 0, & \text{otherwise.} \end{cases}$$

For example, assumed that $n_{crit} = 4$ and $m = 4$, $Z = 1$ if there is one critical VM mapped to all cores and $Z < 1$ if there is at least one core without an assigned critical VM. Thus, for a given set of critical VMs, Z is maximized when the VMs allocation spans over the largest possible number of cores. We say that a VM is *heavy*, if certification requires that this VM is exclusively mapped to a dedicated core and this constraint is kept by the algorithm as well.

If the number of critical VMs does not exceed the number of cores, all critical VMs are mapped to different cores. The partitioning algorithm either minimizes the number of cores, maximizes the criticality distribution (while minimizing the number of cores among partitions of same criticality distribution), or maximizes the criticality distribution for a given maximum number of cores. If minimizing the number of cores is more important, but a high criticality distribution is a secondary goal, one might first run the algorithm with the goal to minimize the number of cores, and then run it again with the obtained number of cores as an input in order to maximize the criticality distribution among solutions with this number of cores.

5.3.3 The Algorithm

The pseudocode listing of the actual partitioning algorithm is shown in Algorithm 2. The variables m_{opt}, Z_{opt}, and Ξ_{opt} (line 2) store the number of processor cores, the

criticality distribution value, and the partition of the so far known optimal solution and are updated whenever a better solution is found. Before generating the search tree, the set of VMs V is sorted according to decreasing utilization (line 4). This is motivated by the introduced pruning condition: if at some node, the sum of the utilizations of the VMs assigned to a specific core is greater than 1, the computational capacity of this core is exceeded and the whole subtree can be pruned. Such a subtree pruning tends to occur earlier, if the VMs are ordered according to decreasing utilization, as Lopez et al. observed for tasks partitioning [Lopez et al., 2003].

The search function SEARCHPARTITION is recursive [Wirth, 1976]. Each call of the function adds a VM (first parameter) to the so far obtained partitioning (second parameter). The parameters of the first call are the first VM and an empty set (line 5), since no partitioning decision was taken at this point. The function calls itself with the next VM (line 30), but only if the stopping condition of the recursion that all VMs were considered ($i = |V|$, line 14) is not fulfilled. This stopping condition ensures that the depth of the tree of n as introduced in Section 5.3 is not exceeded.

First, a new set is created, which represents an additional processor core (line 7). It is empty at this point, since no VM was mapped to this core so far. Subsequently, the for-loop looks at each processor core k, more precisely the set Ξ_k of VMs mapped to it, and adds the considered VM. The periods of the resulting set are harmonized (line 10). Line 11 implements the pruning condition: the algorithm proceeds only if the set of VMs mapped to the considered processor core is schedulable and otherwise removes the last added VM and ends this iteration of the loop without a recursive call (line 31).

If the schedulability condition (line 11) was passed and if all VMs were considered and added to a processor core (line 14), the obtained solution is compared to the so far known optimal solution. If the optimization goal is the maximization of the criticality distribution, a higher value of this metric (line 17) or an equal value with a lower number of processor cores (line 22) means that a better solution was found. Otherwise, the optimization goal is the minimization of the number of processor cores (lines 26-28), since, for the sake of readability, the optimization goal “maximize the criticality distribution for a given maximum number of cores” is not included in this pseudocode listing. For the same reason, the enforcement of the mapping of all heavy VMs to a dedicated core is not included.

Algorithm 2 Partitioning Algorithm

1: ▷ attributes of optimal solution
2: $m_{opt} \leftarrow \infty,\ \ Z_{opt} \leftarrow 0,\ \ \Xi_{opt} \leftarrow \emptyset$

3: **function** BranchAndBoundPartitioning(V)
4:     $V \leftarrow Sort(V, decreasingUtilization)$
5:     SearchPartition(V_1, $\emptyset$, V)

6: **function** SearchPartition(V_i, Ξ, V)
7:     $\Xi_{new} \leftarrow \emptyset$
8:     **for all** $\Xi_k \in (\Xi \cup \Xi_{new})$ **do**
9:         $\Xi_k \leftarrow \Xi_k \cup V_i$
10:         $\Xi_k \leftarrow harmonize(\Xi_k)$
11:         **if** $utilization(\Xi_k) \leq 1$ **then**
12:             **if** $\Xi_k = \Xi_{new}$ **then**
13:                 $\Xi \leftarrow \Xi \cup \Xi_{new}$
14:             **if** $i = |V|$ **then**
15:                 $m \leftarrow |\Xi_k|$
16:                 **if** $goal = criticalityDistribution$ **then**
17:                     **if** $\mathrm{Z}(\Xi) > Z_{opt}$ **then**
18:                         $m_{opt} \leftarrow m$
19:                         $Z_{opt} \leftarrow Z(\Xi)$
20:                         $\Xi_{opt} \leftarrow \Xi$
21:                     **else**
22:                         **if** $(Z(\Xi) = Z_{opt})$ & $(m < m_{opt})$ **then**
23:                             $m_{opt} \leftarrow m$
24:                             $\Xi_{opt} \leftarrow \Xi$
25:                 **else**
26:                     **if** $m < m_{opt}$ **then**
27:                         $m_{opt} \leftarrow m$
28:                         $\Xi_{opt} \leftarrow \Xi$
29:             **else**
30:                 SearchPartition(V_{i+1}, Ξ, V)
31:         $\Xi_k \leftarrow \Xi_k \setminus V_i$

Table 5.1: Example for partitioning: set of virtual machines

	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}
χ	LO	HI	LO	LO	HI	HI	HI	LO	LO	HI
U	0.6	0.5	0.5	0.3	0.25	0.2	0.2	0.2	0.2	0.15

5.3.4 Example

The different outcomes dependent on the optimization goal of the algorithm are illustrated with the exemplary VM set of Table 5.1, assumed that the transformation to harmonic periods was already performed. Figure 5.4 depicts the resulting VM-to-core mapping for three different goals (critical VMs are marked with an asterisk *). Subfigure (a) depicts the outcome if the number of cores is minimized. The VM set is not schedulable on less than four cores, since the sum of the VM utilizations is 3.1. For this solution, the average utilization per core is 0.775 and the criticality distribution Z is $3/5 = 0.6$. Subfigure (b) depicts the outcome for the maximization of the criticality distribution, however with a maximum number of $m_{max} = 4$ cores allowed. The partitioning is therefore still characterized by the minimum number of cores. The criticality distribution Z improves to $4/5 = 0.8$. From a criticality point of view, this mapping is more suitable, since the options to apply an adaptive scheduling and/or protect critical VMs in case of an overload are very limited for core P_3 in the first solution. Subfigure (c) depicts an unrestricted maximization of the criticality distribution, resulting in as many cores as there are critical VMs, in this case one additional core. The optimal criticality distribution $Z = 1$ is achieved, however at the cost of exceeding the minimum number of cores, which leads to a decrease of the average utilization per core to 0.62.

5.4 Evaluation

In the following, the results of the partitioning algorithm for three different optimization goals are compared:

- minimize number of processor cores (abbreviation: *NP*)
- maximize criticality distribution (*CD*)
- exclusive: all critical VMs are heavy and cannot share the core (*EX*)

For the latter two goals, minimizing the number of processor cores is a secondary optimization goal: solutions are considered as better if they are characterized by the same criticality distribution value respectively as well a mapping of all heavy VMs to dedicated cores, but a lower number of processor cores. In order to evaluate

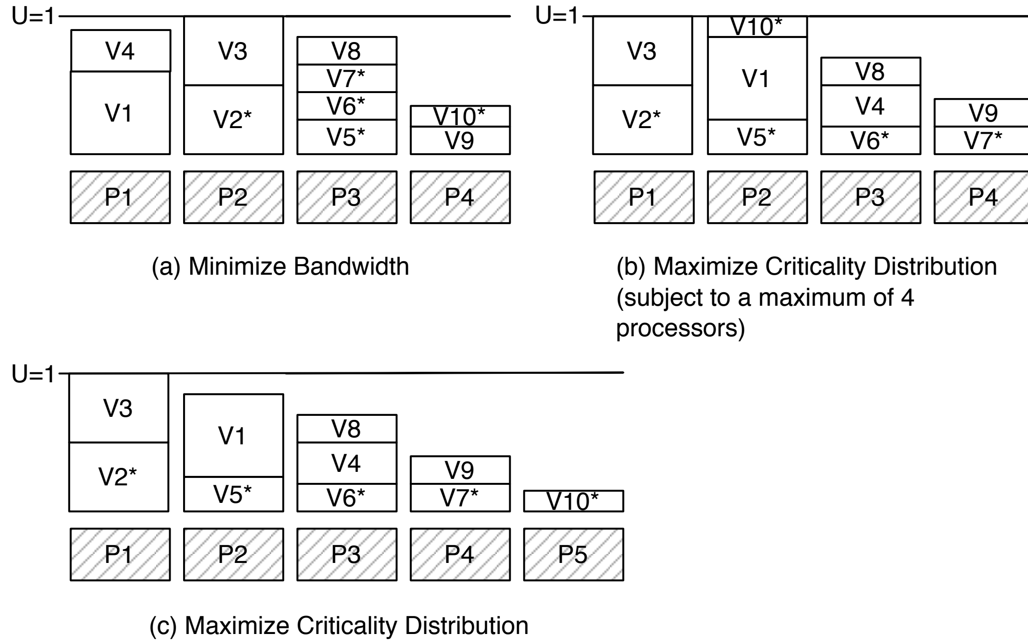


Figure 5.4: Mappings for different optimization goals

the effectiveness of the harmonization of the periods of the periodic resources, the algorithm was executed for each of these goals with and without transformation to harmonic periods (denoted by attaching a H to the introduced abbreviations).

For the random generation of workloads according to uniform distributions, we used Brandenburg's toolkit SchedCAT [Brandenburg, 2014]. It is based on Emberson et al.'s technique for workload generation, which implements an unbiased synthesis of workload attributes for any values for the number of tasks n and sum of their utilizations U [Emberson et al., 2010]. It is possible to vary each attribute independently: a parameter of interest is varied, all other parameters are held constant. The unbiased workload generation ensures fairness, since the attribute selection has a significant influence on the results of empirical evaluations, in our case especially the period selection.

1000 sets of VMs were generated for each size n between 2 and 10, so a total of 9000, with the following parameters:

- a server period Π uniformly distributed within $[10\mu s, 1000\mu s]$,
- a bandwidth Θ of each VM uniformly distributed within $[0.1, 0.2, ..., 0.9]$ in the first experiment and within $[0.1, 0.2, ..., 0.5]$ in the second experiment,
- a criticality $\chi \in \{LO, HI\}$ (with the same probability).

Figure 5.4 plots the obtained number of cores as a function of the number of VMs. For the bandwidth of each VM within $[0.1, ..., 0.9]$ (Figure 5.4 (a)), the transformation of the VM sets to harmonic periods reduces the required computation bandwidth on average by about 15% in the case of minimizing the number of cores, 10% in case of maximizing the criticality distribution, and 5% in the case of the exclusive execution of all critical VMs on a dedicated core. In the case of a maximum of 0.5 for the utilization of any VM (Figure 5.4 (b)), harmonization results in an average reduction of the required computation bandwidth of about 18% (NP and NPH), 7% (CD and CDH), and 6% (EX and EXH).

The harmonization is effective for all three optimization goals. However, the additional partitioning constraints of CD and EX lead to a smaller relative improvement. This results from the lower number of possible solutions (in case of EX, critical VMs cannot share the core; in case of CD, critical VMs can share a core only with non-critical VMs) and the therefore expected lower number of VMs that share a core. Especially the exclusive execution of critical VMs limits the possibility to increase the utilization per core by transformation to harmonic periods, since a transformation cannot be performed for all cores that execute only one critical VM.

If the optimization goal is either the maximization of the criticality distribution or the exclusive execution of critical VMs, the fewer possibilities for VMs to share a core result in a significantly higher average number of cores. Compared to NPH, EXH results in a 33% higher average number of cores for VM bandwidths between 0.1 and 0.9, and even 83% higher for VM bandwidths between 0.1 and 0.5. The difference is larger for the partitioning of smaller VMs, since there are more possibilities to share cores, from which NPH benefits more than EXH, which is restricted regarding core sharing. CDH results in as many cores as there are critical VMs. The average number of cores is 11% higher for VM bandwidths between 0.1 and 0.9 and 36% higher for VM bandwidths between 0.1 and 0.5.

On the other side, the advantage of these partitioning goals is a criticality distribution of one, whereas NP produces an average criticality distribution in the range of about 0.8 to 0.9 and in several cases significantly lower values, even down to the minimum of $1/n$. Therefore, Figure 5.4 depicts next to the average values of the criticality distribution as well the range of possible values from $1/n$ to 1. In case of a maximum of 0.5 for the utilization of a VM, the average criticality distribution drops to values in the range of 0.65 to 0.8, since it is more likely that VMs share a core and, thus, as well more likely that critical VMs share a core.

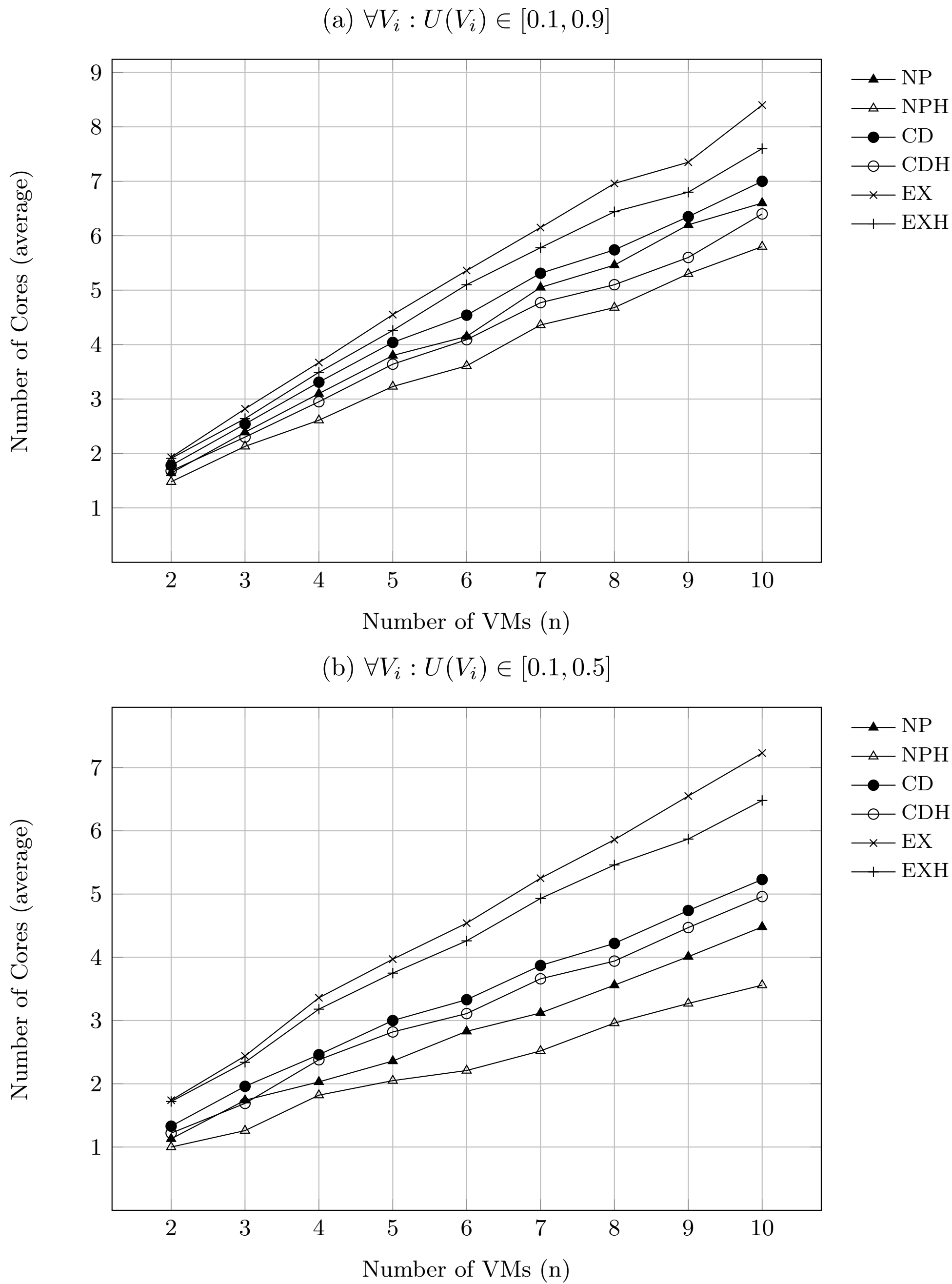


Figure 5.5: Average number of processor cores for different partitioning goals: minimize number of processor cores (NP), minimize number of processor cores incl. transformation to harmonic server periods (NPH); maximize criticality distribution (CD), incl. transformation to harmonic server periods (CDH); minimize number of processor cores and assign cores exclusively to critical VMs (EX), incl. transformation to harmonic server periods (EXH); (a) VM Bandwidths between 0.1 and 0.9, (b) VM Bandwidths between 0.1 and 0.5

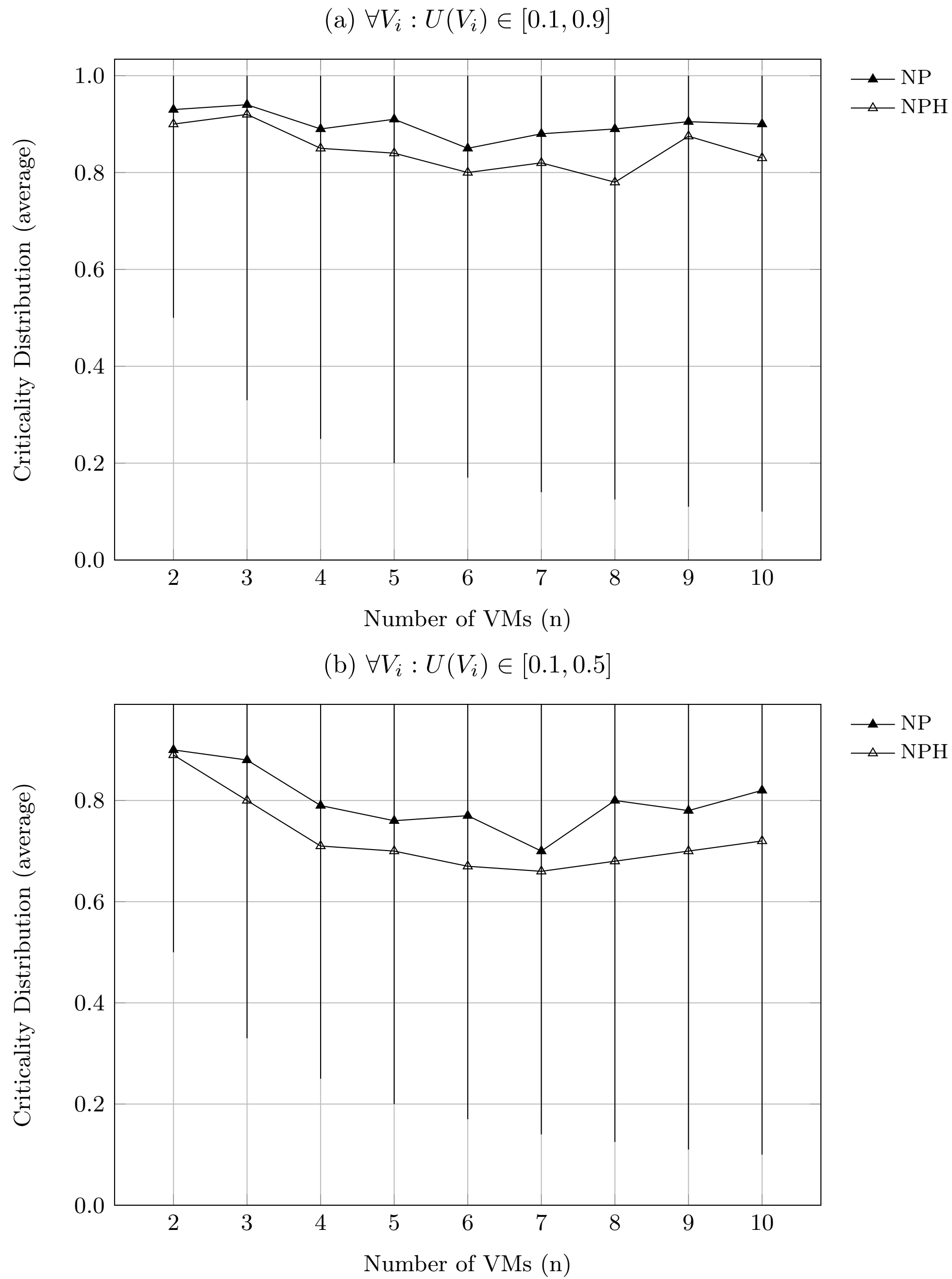


Figure 5.6: Average criticality distribution for the partitioning goal minimize number of processor cores, with (NPH) and without (PH) transformation to harmonic server periods; (a) VM bandwidths between 0.1 and 0.9, (b) VM bandwidths between 0.1 and 0.5

For practical applications, a combination of the different optimization goals might be reasonable. It could be the case that not all critical VMs but some have to be executed on a dedicated core. Moreover, one could first run the algorithm with the optimization goal minimization of the number of cores and in a subsequent run find among all solutions with this minimum the one that maximizes the criticality distribution. Or, if the minimum number of cores is three and this means that a quad-core processors had to be used, one could maximize the criticality distribution for all solutions with three or four cores.

5.5 Summary

The manual partitioning of virtual machines with real-time requirements onto a multicore platform does not guarantee optimality and does not scale with regard to the upcoming higher number of both virtual machines and processor cores. This thesis proposes an algorithmic solution. As a prerequisite, the partitioning problem of mapping real-time virtual machines to a homogeneous multiprocessor architecture was defined. Models were adapted to abstract and specify the computation time demand of a virtual machine and the computation time supply of a shared core, in order to analytically evaluate whether it is guaranteed that the execution time demand of a virtual machine is satisfied. The virtual processor dimensioning keeps the reactivity requirements of the virtual machine and exploits the freedom of design to create harmonic periods for all virtual processors that are assigned to the same core in order to increase the schedulable utilization.

The application of a branch-and-bound algorithm was proposed, since the realistic small number of virtual machines enables the systematic generation and comparison of all candidate solutions. In contrast to a manual solving, the algorithm provides analytical correctness, which can support system certification. The algorithm's optimization goal can be configured according to two basic optimization metrics: required number of processor cores and criticality distribution. The latter realizes a partitioning that considers criticality levels. It increases the chances to be able to protect critical virtual machines in case of an overload situation and increases the potential to benefit from an adaptive scheduling. The partitioning algorithm either minimizes the number of cores, maximizes the criticality distribution, or maximizes the criticality distribution for a given maximum number of cores. If certification requires that virtual machines are mapped exclusively to a dedicated core, this constraint is enforced by the algorithm. The different outcomes were illustrated exemplarily and evaluated with synthetic workloads.

Chapter 6

Adaptive Partitioned Hierarchical Scheduling

Contents

6.1 Problem Statement 110

6.2 Related Work 112

6.3 Scheduling Architecture 114

6.3.1 Server-based Virtual Machine Scheduling 115

6.3.2 Fixed Priority Virtual Machine Scheduling 117

6.4 Adaptive Bandwidth Distribution 118

6.4.1 Distributing Structural Slack 119

6.4.2 The Algorithm and its Computational Complexity 121

6.4.3 Protection under Overload Conditions 123

6.5 Correctness of Bandwidth Distribution 126

6.5.1 Steady State: Temporal Isolation and Minimum Bandwidth Guarantee 127

6.5.2 Correctness during Mode Transitions 129

6.5.3 Correctness of Redistribution of Dynamic Slack 133

6.5.4 Handling of Multiple Mode Change Requests 140

6.6 The Case for Paravirtualization 140

6.7 Integration into Hypervisor and Operating System . . . 143

6.8 Evaluation 144

6.8.1 Scheduling Simulator 144

6.8.2 Execution Times 146

6.8.3 Overhead versus Benefit: Threshold for Slack Redistribution 148

6.8.4 Memory Footprint 149

6.8.5 Paravirtualization Effort 150

6.8.6 Comparative Evaluation 150

6.9 Summary 158

A partitioned multicore scheduling is proposed in this thesis, due to its advantages over a global scheduling strategy as discussed in the last chapter, e.g., lower run-time complexity, higher predictability, more efficient run-time scheduling, and improved cache performance. For a partitioned scheduling, the following two sub-problems have to be solved:

(i) partitioning: allocation of virtual machines to processor cores,

(ii) uniprocessor hierarchical scheduling on each core.

The previous chapter presented a solution for sub-problem (i). The partitioning algorithm produces a mapping of virtual machines to processor cores that guarantees schedulability of all virtual machines, minimizes the number of processor cores and distributes the critical guests among the cores. This second goal is motivated by the benefits for an adaptive scheduling technique, since the non-critical virtual machines can benefit from unused reservations of the critical virtual machines. This chapter presents such a technique for sub-problem (ii). The virtual machine scheduling is based on computation time servers, which are replenished at fixed period but with varying budget in order to implement an adaptive bandwidth management. The use of the periodic resource model, rate monotonic scheduling, and a bandwidth distribution mechanism that ensures a guaranteed minimum combines analyzability at design time with adaptability at runtime.

After specifying the issues that need to be addressed (Section 6.1), we discuss related work (Section 6.2), introduce the scheduling infrastructure (Section 6.3) and the dynamic bandwidth distribution policy (Section 6.4), analyze the policy in order to show its correctness (Section 6.5), make a case for paravirtualization (Section 6.6), present the integration into system software (Section 6.7), and finally evaluate the proposal (Section 6.8).

6.1 Problem Statement

Existing virtualization solutions for embedded systems apply a static computation bandwidth allocation [Gu and Zhao, 2012]. Typical solutions are the allocation of each VM to a dedicated processor core, the static precedence of a single real-time VM per core (with the non-real-time VMs scheduled in background), or a cyclic schedule with fixed execution time slices. The application of static resource allocation to systems that are characterized by varying computation time demand inherently leads to resource fragmentation. Reserved but unused bandwidth cannot be reclaimed to

improve the performance of other VMs, but is wasted as idle time. An adaptive management of the computation bandwidth is of great potential for hypervisor-based systems. The two major requirements can be derived from the characteristics of the hosted guest systems (as introduced with the motivational example of Chapter 1 and the workload model in Section 4.1): temporal isolation and adaptability.

Temporal Isolation. Safety requires that the VMs do not interfere with each other, especially since independently developed system of different criticality levels are consolidated. Key requirement for the integration of real-time systems and particularly for certification is therefore temporal isolation / freedom from temporal interference, defined in the context of this work by the following conditions:

- The timing requirements of the integrated guest systems can be validated independently.
- The hypervisor's VM scheduling provides all guest systems sufficient computation time to meet their real-time constraints.
- Overruns within a VM provoke under no circumstances that other VMs violate their real-time constraints (failure containment).
- The execution of a guest system is never interrupted for a longer time than demanded by its reactivity requirements.

Adaptability. The scheduling shall respond to varying computation time demand with an appropriate redistribution that reduces waste of CPU cycles (avoidable idle time) and increases the allocation to VMs that benefit from additional computation time. Computation time variations are caused by mode changes (incl. enabling/disabling) and when a guest system's actual execution time is considerably smaller than the reserved worst-case demand.

With the given definition of temporal isolation, these two requirements are not conflicting. It does not demand an entirely uninfluenced execution of the guest systems, a degree of temporal isolation that is irreconcilable with any adaptive scheduling. The targeted level of temporal isolation is the absence of corrupting impact between guest systems, derived from the standard for functional safety of road vehicles ISO 26262 [ISO, 2011].

6.2 Related Work

Adaptive Resource Management for Tasks

Feedback-control algorithms for adaptive reservations [Abeni et al., 2005, Santos et al., 2012] measure the performance of the served tasks and adjust the budgets according to a certain control law. Khalilzad et al. introduced a hierarchical single-core scheduling framework that modifies the budgets of periodic servers after a deadline miss (overload situation) based on the amount of idle time [Khalilzad et al., 2012]. Block et al. presented an adaptive multiprocessor scheduling framework, which adjusts processor shares in order to maximize the quality of service (QoS) of soft real-time tasks [Block et al., 2008]. In contrast, the redistribution of bandwidth in the context of this work is triggered by the scheduled components themselves, and not based on the observation of their performance.

Bini et al. introduced an adaptive multicore resource management for mobile phones [Bini et al., 2011], which selects a service level for each application by solving an integer linear programming problem. Maggio et al. proposed a game-theoretic approach for the resource management among competing QoS-aware applications, which decouples service level assignment and resource allocation [Maggio et al., 2013]. Applications do not have to inform the resource manager about the available service levels, but about the start and stop time of each job.

Zabos et al. presented the integration of a spare reclamation algorithm into a middleware layer [Zabos et al., 2009], which is placed on top of a real-time OS, and not underneath as a hypervisor. Dynamic reclamation algorithms such as GRUB [Lipari and Baruah, 2000] or BASH [Caccamo et al., 2005] take advantage of spare bandwidth when tasks do not need their WCET and distribute it in a greedy or weighted manner, as proposed in this work.

Nogueira and Pinho proposed a dynamic scheduler for the coexistence of isolated and non-isolated servers [Nogueira and Pinho, 2007]. An isolated server obtains a guaranteed budget, whereas budget can be stolen from a non-isolated server. In order to avoid the increased computational complexity of a fair distribution, the entire slack is assigned to the currently executing server. Bernat and Burns proposed as well a budget-stealing server-based scheduling [Bernat and Burns, 2002]. Each server handles a single soft task and in overload situations can steal budget from the other servers. Temporal isolation is lost and a server of low priority might receive less bandwidth than requested.

IRIS is a resource reservation algorithm that handles overload situations by spare

bandwidth allocation among hard, soft, and non-real-time tasks [Marzario et al., 2004]. As it is the case for our approach, minimum budgets are guaranteed and the remaining bandwidth is distributed in a fair manner among the servers. Conversely to the proposal in this thesis, the scheduling is based on an extension of the Constant Bandwidth Server (CBS) and EDF. In the context of the ACTORS EU project, an adaptive reservation-based multicore CPU management for soft real-time systems was developed, as well based on CBS and EDF [Arzen et al., 2011]. Similar to our work, a partitioned hierarchical scheduler is proposed, however one for the OS (implemented in Linux), handling groups of threads.

Su and Zhu proposed an elastic mixed-criticality task model and an EDF-based uniprocessor scheduling [Su and Zhu, 2013]. Slack is passed at runtime to low-criticality tasks based on variable periods, not on variable execution time allocations as in our work. Anderson et al. presented the first work on server-based mixed-criticality multicore scheduling [Anderson et al., 2009]. On each core, budget is specified for each criticality level and consumed in parallel corresponding to the respective level. Mollison et al. introduced the notion of higher-criticality tasks as slack generators for lower-criticality tasks [Mollison et al., 2010]. Herman et al. presented the first implementation of an OS's mixed-criticality multicore scheduler and discussed design tradeoffs [Herman et al., 2012]. Their framework reclaims capacity lost due to WCET pessimism. All these works are based on the mixed-criticality task model with one WCET per criticality level, which is not used in this work as discussed in Section 4.5

Virtual Machine Scheduling

All cited approaches are concerned with the operating system's scheduling of tasks. In the following, related work regarding the hypervisor's scheduling of VMs is discussed, starting with non-adaptive approaches. Bruns et al. evaluated virtualization to consolidate subsystems of mobile devices on a single processor [Bruns et al., 2010]. The software stack consists of an L4/Fiasco microkernel with a priority-based round-robin scheduling and a paravirtualized Linux. Crespo et al. designed XtratuM, a hypervisor for the avionics domain with a fixed cyclic scheduling [Peiró et al., 2010]. A redesign for multicore processors was recently published [Carrascosa et al., 2013]. Sha [Sha, 2004] and Kerstan [Kerstan, 2011] proposed as well fixed cyclic scheduling for single-core processors.

Yang et al. proposed a compositional scheduling framework for virtualization based on the L4/Fiasco microkernel [Yang et al., 2011]. As our proposal, it is based

on servers scheduled by RM, but without slack distribution. Masrur et al. implemented a fixed-priority variant of Xen's Simple EDF scheduler [Masrur et al., 2010]. Moreover, they proposed to schedule VMs by partitioned RM and showed how to design a schedule for guest systems that schedule their tasks according to the deadline monotonic policy [Masrur et al., 2011]. Cucinotta et al. examined hard reservations and an EDF-based soft real-time scheduling policy to provide temporal isolation among I/O-intensive and CPU-intensive VMs [Cucinotta et al., 2011b]. Their implementation is based on the Linux kernel module KVM.

Lee et al. presented an adaptive compositional scheduling framework for the Xen hypervisor [Lee et al., 2011]. They realize resource models with periodic servers and introduce enhancements to the server design in order to increase the resource utilization. Their work-conserving periodic server lets a lower-priority non-idle server benefit when a high-priority server idles. Their capacity reclaiming periodic server allows idle time of a server to be used by any other server. In contrast to this work, their slack distribution does not consider fairness (slack is always passed to a single VM) and they target application domains with powerful hardware and timing requirements in the range of milliseconds (scheduling quantum of 1ms), whereas our work targets low-performance and memory-constrained embedded hardware with timing requirements in the sub-millisecond range. In addition, our approach includes the redistribution in case of mode changes.

6.3 Scheduling Architecture

An adaptive hierarchical real-time scheduling technique for hypervisor-based virtualization is introduced in the following. It is based on execution time servers, a fixed priority assignment according to the Rate Monotonic policy, and an efficient algorithm for the online computation of bandwidth. The hosted operating systems can apply any scheduling algorithm for task scheduling, as long as it allows to abstract the computation time requirements of the task set in terms of a demand-bound function, as introduced in Section 4.1.2. The scheduling architecture is depicted in Figure 6.1. In this example, four VMs are executed on two processor cores. The guest operating systems apply Rate Monotonic (RM), Earliest Deadline First (EDF), and Cyclic Executive as scheduling policies.

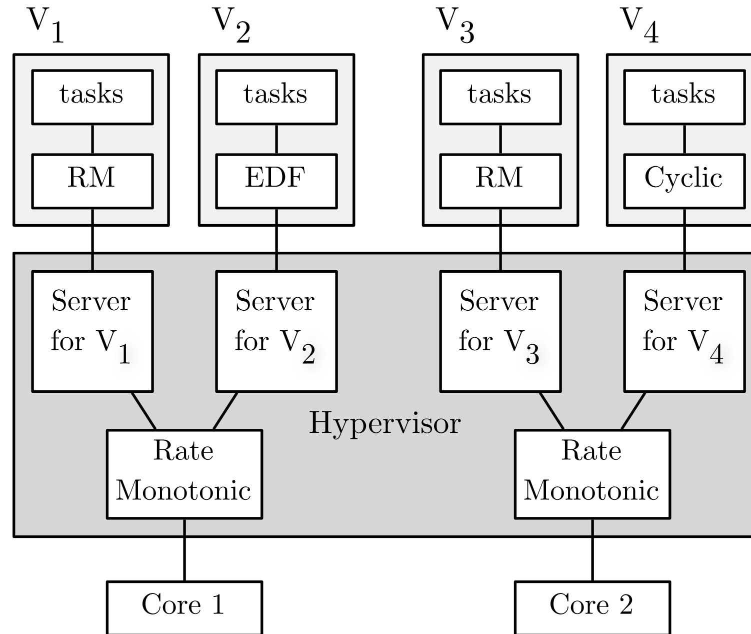


Figure 6.1: Server-based partitioned hierarchical scheduling

6.3.1 Server-based Virtual Machine Scheduling

Servers, as introduced in Section 2.1.2, are originally a scheduling concept for hybrid task sets of periodic and aperiodic tasks. Beyond, servers can be used to realize virtual processors. The characteristics of a server, provision of computation time, limited by a capacity, turn servers into an excellent abstraction of virtual processors. Deng, Liu, and Sun (University of Illinois) presented a pioneering server-based hierarchical real-time scheduling approach for the two levels *application* and *task*, with multiple tasks belonging to each application and scheduled by application-specific schedulers [Deng et al., 1996, Deng and Liu, 1997, Deng et al., 1997]. Server-based hierarchical scheduling has been applied primarily to multiple scheduling levels within the operating system [Goyal et al., 1996, Kuo and Li, 1998, Lipari et al., 2000, Wang and Lin, 2000, Regehr and Stankovic, 2001, Lipari and Baruah, 2001, de Niz et al., 2001, Saewong et al., 2002, Lipari and Bini, 2003, Davis and Burns, 2005, Davis and Burns, 2006, Pulido et al., 2006, Zhang and Burns, 2007, Behnam et al., 2008, Asberg et al., 2009, Inam et al., 2011].

Independent of each other, Cucinotta et al. [Cucinotta et al., 2011a], Xi et al. [Xi et al., 2011], Yang et al. [Yang et al., 2011], and Groesbrink [Groesbrink, 2010] were to the best of our knowledge the first to implement server-based hierarchical scheduling on the two levels hypervisor and operating system.

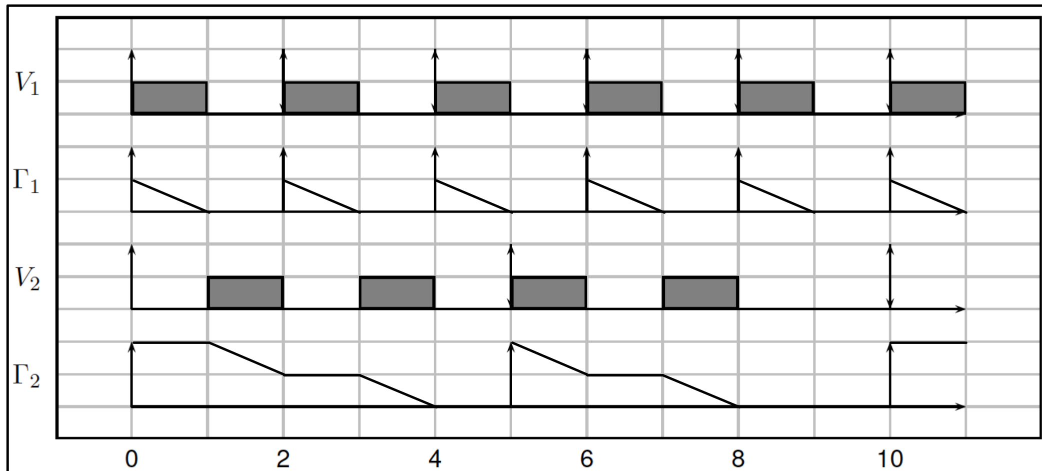


Figure 6.2: Server-based scheduling of two virtual machines V_1 and V_2: the virtual processors $\Gamma_1(2, 1)$ and $\Gamma_2(5, 2)$ are implemented by periodic servers

In the context of this work, each virtual processor $\Gamma_i(\Pi_i, \Theta_i)$ is implemented as a periodic server, characterized by a period and an execution time budget. At the beginning of each period, the budget is replenished by the hypervisor. If scheduled and therefore active, a server's budget is used to satisfy the computation time demand of the associated virtual machine. The budget is consumed at the rate of one per time unit and once exhausted, the server is not ready for execution until the next period. There is no cumulation of budget from period to period. Figure 6.2 depicts an exemplary schedule with two virtual machines and their server budgets.

Such a server enforces a guaranteed, but bounded computation time for a VM in a specified time span, even in the presence of overloads internal to any VM. Just as important, it provides already the mechanism to apply an adaptive scheduling. By varying the replenishment budgets at runtime, it is possible to increase or decrease the computational bandwidth allocation. By discarding the remaining budget within a server's period, reserved but unneeded bandwidth can be withdrawn.

Figure 6.3 shows the state transition diagram for such a periodic server, relative to the following four states:

Ready. A server that is ready to execute, however, currently not executed, is in this state. A server is ready to execute if it has budget left for its current period.

Running. In this state, the associated VM is executing on the processor core.

Idle. In the idle state, the associated VM is executing on the processor core, but does run only the idle task, since it finished the workload for the current period and

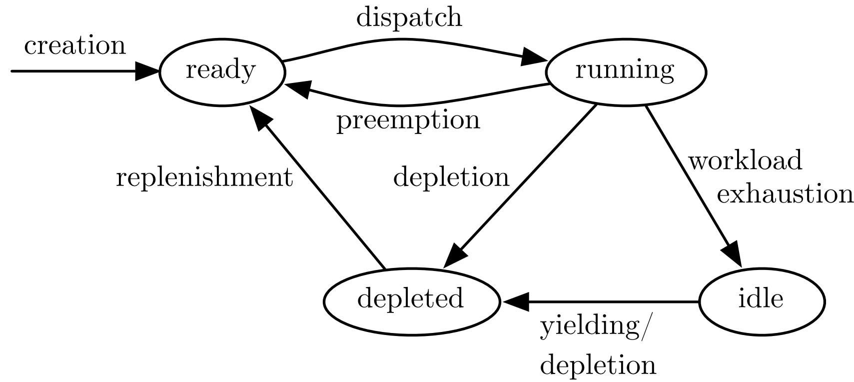


Figure 6.3: State transition diagram for the periodic server

waits for the beginning of the next period.

Depleted. Once the budget of a server is exhausted, the server enters this state. The associated VM cannot be executed before the budget is replenished.

After creation, a server enters the ready state. A server moves to the running state once it was selected by the hypervisor's scheduler as the highest-priority server among the ready servers and dispatched. It may be preempted when a higher priority server becomes ready. In this case, it enters the ready state and preserves its budget. When the running VM finished its workload for the current period and starts the execution of the idle task, it moves from the running state to the idle state. It may either idle its budget away and enter the depleted state, or yield in order to move to this state immediately (budget is set to zero). Not shown in Figure 6.3, it may as well be preempted during the execution of the idle task and moved to the ready state. When the budget of the server in running state is exhausted, it is moved to the depleted state. Servers in this state wait for the replenishment of their budget at their next period. As well not shown in the diagram: it is possible that a server in the ready state, running state, or idle state is replenished. In these cases, the server stays in its state.

6.3.2 Fixed Priority Virtual Machine Scheduling

In the context of this work, a partitioned virtual machine scheduling is applied, as justified at the beginning of Chapter 5. On each core, the hypervisor schedules the corresponding servers of the assigned guest systems by static priorities according to the Rate Monotonic (RM) policy: the higher the request rate of a server (i.e., the smaller the period Π), the higher its priority (priority assignment as a monotonic

function of the servers' rates) [Liu and Layland, 1973b]. The first period of all servers starts at the exact same time. The hypervisor manages one ready queue per core and enforces that at each point in time the highest-priority server among those that have budget (and are therefore ready to execute) is executed. Starvation of lower-priority servers is nevertheless precluded, since the budget limits the execution time of the higher-priority servers. For the same reason, a criticality-disregarding priority assignment may be made.

Since rate monotonic is based on a fixed-priority assignment, it is characterized by a high analyzability and predictability, even in overload situations. In case of a dynamic-priority scheduling, it is largely unpredictable to tell which scheduled task/VM will miss its deadline, since it is dependent on the dynamic priorities at the time at which the overload occurs. Industry and certification authorities have a strong preference for static scheduling algorithms such as RM [Leung and Zhao, 2005]. The major drawback of using rate monotonic as server scheduler is its low utilization bound. This, however, can be tackled by selecting harmonic periods, as presented in the last chapter. The schedulability of n servers is guaranteed if the sum of the minimum utilizations is less than or equal to one:

$$\sum_{i=1}^{n} U_{min}(V_i) \leq 1 \tag{6.1}$$

The overhead of VM context switches is not negligible and therefore added to the execution time demand of the VMs. With the applied fixed-priority assignment, each VM preempts at most one VM (VMs do not perform self-blocking and resume later). The overhead is included by adding the time for two context switches, one at the start and one at the completion of a VM's execution. If a VM is preempted, the context switch overhead is accounted for through the execution time of the preempting VM. This approach is pessimistic but safe, since it assumes that every instance of an executing VM causes preemption, which is not necessarily the case.

6.4 Adaptive Bandwidth Distribution

Virtual machine scheduling with fixed bandwidth allocations is inefficient for dynamic systems and wastes CPU time. An adaptive scheduling that dynamically allocates processor bandwidth to VMs and not statically at design time has a great potential to increase the processor utilization and reduce delays. This section introduces such an adaptive VM scheduling, which makes spare computation time caused by mode changes and idling of a guest system available to other VMs. The bandwidth

adaptation is realized by a dynamic setting of the servers' replenishment budgets. Their harmonic periods are always kept fixed, though, which simplifies analysis and maintenance of schedulability. Moreover, since VMs are statically assigned to the processor cores, bandwidth redistribution has to be handled separately for each core.

A bandwidth redistribution is considered in two situations:

I. *Distribution of Structural Slack*:
Events with a significant and lasting impact on the resource utilization trigger a redistribution of the updated spare bandwidth U_{spare}. Those events are an enabling or disabling of a task or entire VM, or a mode change to a mode with differing resource demand (modes differ regarding U_{lax}). Both the hypervisor or the guest system itself can trigger a mode change.

II. *Distribution of Dynamic Slack*:
Task execution times vary at runtime. When the actual execution time of a task is considerably smaller than the WCET, the difference is termed dynamic slack. Especially in the case of critical tasks, the very pessimistic WCET is often not reached, but has to be reserved for each period. When a guest OS does not demand the allocated share, it can yield and the hypervisor reassigns the reserved but no longer required bandwidth, instead of wasting bandwidth by idling.

The occurrence of these situations potentially triggers a bandwidth redistribution. But since a redistribution incurs a certain overhead, the hypervisor evaluates whether a potential adaptation is reasonable. The two situations differ regarding duration. Structural changes have a long term impact and for most systems tend to be in effect for orders of magnitude longer than the redistribution overhead. The dynamic slack distribution is a short term measure, potentially occurring in each server period. For this reason, the distribution is enforced in a different manner, including an expiration mechanism, which is explained in detail in Section 6.5.3. Moreover, the redistribution of dynamic slack is performed only if the slack compensates the costs, which can be determined offline for each specific hardware platform and used at runtime as a threshold (see Section 6.8.3).

6.4.1 Distributing Structural Slack

The adaptive resource management is implemented by a dynamic modification of the bandwidth allocations of the servers. The server period Π_k is fixed, whereas the

capacity Θ_k is set dynamically in an adaptive manner. The actual bandwidth distribution among the VMs is carried out in two steps. First, the minimum bandwidth requirement U_{min} is allocated to each VM. Second, the spare bandwidth U_{spare} is distributed with the objective to satisfy U_{lax} of the active VMs as much as possible. Consequently, a minimum bandwidth is guaranteed for each VM and additional bandwidth might be assigned. Since VMs are statically assigned to the processor cores and scheduled in a partitioned manner, spare bandwidth has to be handled separately for each core. The spare bandwidth of a core that hosts n VMs at point in time t is equal to:

$$U_{spare}(t) = U_{lub} - \sum_{V_i \in V'} U_{min}(V_i) = 1 - \sum_{V_i \in V'} U_{min}(V_i) \tag{6.2}$$

U_{lub} is equal to one because of the harmonic relation between the server periods. U_{spare} is dependent on the point in time t, since a disabled VM has a U_{min} of zero. The calculation of U_{spare} is therefore subject to the set of currently enabled VMs, denoted as V'.

The distribution policy considers two VM characteristics for determining the VM-specific spare bandwidth shares, namely criticality level and weight, in this order. The criticality level χ is the dominant factor and the bandwidth is assigned in a greedy manner in order of decreasing criticality. The highest criticality level obtains as much bandwidth as possible, limited by either the distributable amount U_{spare} or the maximum bandwidth requirement of its VMs. Typically, the higher the criticality of a VM, the more likely a large U_{min} and a low U_{spare}, since critical systems are rarely quality of service driven. If there is spare bandwidth left, the next lower criticality level is served and so on. The weights influence the bandwidth assignment among VMs of the same criticality level, since a greedy strategy lacks fairness. The weights are normalized considering the current set of VMs V' among which U_{spare} will be distributed:

$$\hat{w}(V_i) = \frac{w(V_i)}{\sum_{V_j \in V'} w(V_j)} \tag{6.3}$$

Assuming the bandwidth U_{spare} is to be distributed among VMs of similar criticality level, the VM-specific shares U_{add} are set to:

$$U_{add}(V_i) = \begin{cases} U_{lax} & \text{if } \hat{w}(V_i) \cdot U_{spare}(t) > U_{lax} \\ \hat{w}(V_i) \cdot U_{spare}(t) & \text{otherwise} \end{cases} \tag{6.4}$$

This results in a total bandwidth assignment to a specific VM of:

$$U(V_i) = U_{min}(V_i) + U_{add}(V_i) \tag{6.5}$$

This value determines the new server bandwidth and the replenished budget follows as $\Theta_i = U(V_i) \cdot \Pi_i$.

The formula $\hat{w}(V_i) \cdot U_{spare}$ may result in a value of U_{add} greater than U_{lax} for some VMs. In this case, to avoid a total bandwidth assignment that exceeds the VM's maximum, their U_{add} is truncated to U_{lax}, and the remaining bandwidth is distributed among the other VMs in the same proportion of their weights. The additional bandwidth distribution may result in more VMs reaching their U_{lax} limit, causing again truncations. This process may continue until there is no remaining bandwidth, resulting in the iterative algorithms for elastic bandwidth management proposed in [Buttazzo et al., 2002] and [Marau et al., 2011]. These iterative algorithms are characterized by a significant overhead and execution time variations, which are problematic for real-time systems.

The next section proposes a new non-iterative algorithm with significant benefits regarding overhead. The algorithm takes advantage of the fact that, if any VM will reach its U_{lax} limit by using Equation 6.4, then the first to reach that limit will be the VM with the lowest value of $U_{lax}(V_i)/\hat{w}(V_i)$. By setting the U_{add} values for all VMs, starting with the VM with lowest $U_{lax}(V_i)/\hat{w}(V_i)$ and ending with the largest, it is guaranteed that the first m VMs (m may be 0) will reach their U_{lax} limit, while all subsequent VMs will not reach this limit. A single iteration over all VMs is sufficient to achieve the same bandwidth distribution as the previously proposed algorithms with several iterations.

The use of U_{lub} when computing U_{spare} guarantees that schedulability of the VMs is maintained. An important aspect to consider is when to apply the capacity changes that result from the bandwidth redistribution. The enforcement policy and its analysis is presented in Section 6.5.

6.4.2 The Algorithm and its Computational Complexity

Algorithm 3 presents the pseudocode of the proposed bandwidth distribution. As just motivated, the VM set is sorted according to increasing $U_{lax}(V_i)/\hat{w}(V_i)$. First, U_{spare} is calculated based on Equation 6.2. In the following, the algorithm iterates over all criticality levels in descending order to implement the greedy strategy regarding criticality levels. If no U_{spare} is left after the distribution to the next higher criticality level or there was none at all, the algorithm terminates (line 4).

Only VMs of the considered criticality level, which are enabled (not depicted), and could benefit from additional bandwidth are considered (line 6) and added to the set $V[\chi]$. Next, the sum of the weights ($\hat{w}^{\Sigma}$) and the sum of the utilization laxities

(U_{lax}^{Σ}) is computed for this set (lines 8-9). If the available U_{spare} exceeds U_{lax}^{Σ}, all VMs can be satisfied immediately (lines 10-12). Otherwise, the algorithm iterates over all VMs and assigns utilization shares based on the normalized weights (lines 14-15). U_{add} is bounded to U_{lax} (lines 16-17). If it has to be truncated, the sum of the weights and of the utilization laxities has to be updated (lines 18-19).

Algorithm 3 Bandwidth Distribution

Require: V (sorted regarding increasing $U_{lax}/\hat{w}$)

1: $U_{spare} \leftarrow$ COMPUTE_U_SPARE(V)
2: **for all** χ (descending order) **do**
3:     **if** $U_{spare} = 0$ **then**
4:         exit()
5:     $\hat{w}^{\Sigma}[\chi] \leftarrow 0$, $U_{lax}^{\Sigma}[\chi] \leftarrow 0$
6:     $V[\chi] \leftarrow \{V_i | V_i \in V \wedge \chi(V_i) = \chi \wedge U_{lax}(V_i) > 0\}$
7:     **for all** $V_i \in V[\chi]$ **do**
8:         $\hat{w}^{\Sigma}[\chi] \leftarrow \hat{w}^{\Sigma}[\chi] + \hat{w}(V_i)$
9:         $U_{lax}^{\Sigma}[\chi] \leftarrow U_{lax}^{\Sigma}[\chi] + U_{lax}(V_i)$
10:     **if** $U_{spare} \geq U_{lax}^{\Sigma}[\chi]$ **then**
11:         **for all** $V_i \in V[\chi]$ **do** $U_{add}(V_i) \leftarrow U_{lax}(V_i)$
12:         $U_{spare} \leftarrow U_{spare} - U_{lax}^{\Sigma}[\chi]$
13:     **else**
14:         **for all** $V_i \in V[\chi]$ **do**
15:             $U_{add}(V_i) \leftarrow U_{spare} \cdot \hat{w}(V_i)/\hat{w}^{\Sigma}[\chi]$
16:             **if** $U_{add}(V_i) > U_{lax}(V_i)$ **then**
17:                 $U_{add}(V_i) = U_{lax}(V_i)$
18:                 $\hat{w}^{\Sigma}[\chi] \leftarrow \hat{w}^{\Sigma}[\chi] - \hat{w}(V_i)$
19:                 $U_{spare} \leftarrow U_{spare} - U_{add}(V_i)$
20:         exit()

The basic bandwidth distribution for n virtual machines based on Equation 6.4 has a computational complexity of $\mathcal{O}(n)$. But since the algorithm requires the input VM set to be sorted, the computational complexity becomes $\mathcal{O}(n \cdot \log n)$. Nevertheless, note that an initial sorting can be done offline. The sorting needs to be repeated whenever the utilization laxity U_{lax} of one or multiple VMs changes due to a mode change (weights are set by the system designer and do not change at runtime).

In the best case, which is expected to be the most frequent case (it takes for example place when the mode change is triggered by the guest system itself), only

the parameters of a single VM differ to the previous execution of the algorithm. Consequently, the algorithm just has to correct the previous order, i.e., insert this VM into the sorted order. This can be done in a single iteration over all VMs, resulting in a computational complexity of $\mathcal{O}(n)$. In the worst case, the hypervisor wants to perform a mode change for all VMs at the exact same time, which requires a re-sorting of the entire VM set with a computational complexity of $\mathcal{O}(n \cdot \log n)$. However, note that the number of VMs assigned to the same core is determining, not the total number of VMs executed on the multicore processor.

6.4.3 Protection under Overload Conditions

This chapter introduced so far a scheduling framework that is adaptive regarding mode changes and execution time variations. By adapting the bandwidth according to the current load of the guest systems, the CPU utilization is increased. A bandwidth redistribution is performed as well in a very specific load situation, namely when an unforeseen overload of a critical guest results in an overload of the entire system. Such an overload might take place if the determined worst-case demand is wrong, which is not only theoretically possible due to the complexity of worst-case analysis for modern embedded software and processor architectures.

Overload handling is motivated by the *Criticality Inversion Problem* [de Niz et al., 2009], as already discussed in Section 5.3.2. This problem occurs if a critical VM requires more computation time than provided by its worst-case reservation and is stopped to allow a non-critical VM to run, resulting in a deadline miss for a task of the critical VM. Note that the VM scheduler as introduced so far produces this behavior: the worst-case reservations are enforced by the hypervisor, even in case of a run-time overload of a critical guest. The VM scheduling does not consider criticality levels, but schedules according to priorities based on the periods.

By definition of criticality as severity of failure, it is more important to prevent a deadline miss for a critical VM by continuing its execution, instead of protecting non-critical VMs. However, this is true only if computation time of a non-critical VM can be stolen; there is no preference among VMs of same criticality. If only the requirements of either a critical VM or a non-critical VM can be satisfied, a criticality inversion precluding scheduling has to fulfill the requirements of the former.

The considered overload situation is defined as follows: a critical guest system ran out of budget (both U_{min} and U_{add}) in the current period of its virtual processor, but did not finish the execution of critical tasks. This can only happen if the determined worst-case execution time is incorrect, i.e., too small, resulting in a value for U_{min}

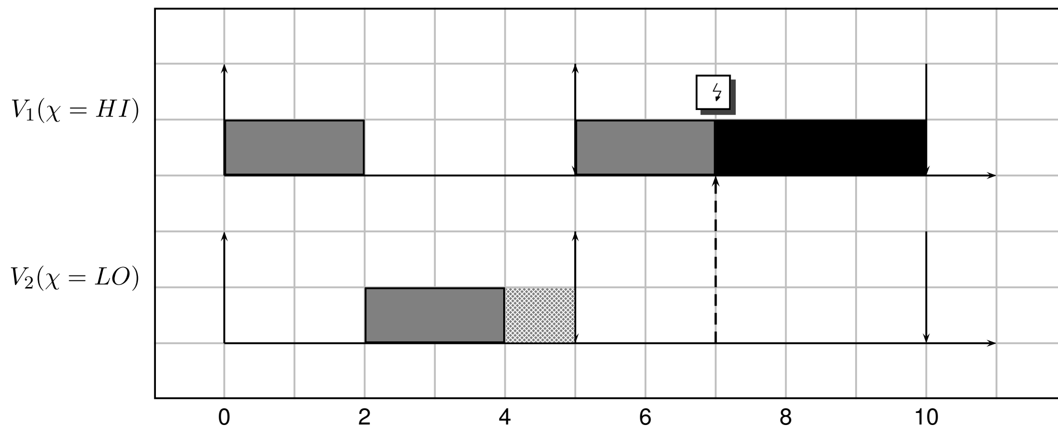
that is too small. An overload is only given if the workload that specifies the minimum bandwidth is not finished and not related to an unsatisfied utilization laxity U_{lax}.

An overload does not trigger a redistribution of the spare bandwidth as introduced in the previous section. Instead of considering all VMs and obtaining a higher bandwidth for the overloaded guest in its next period, an immediate allocation of additional computation time is attempted. We differentiate between two cases, namely whether the guest system that would be dispatched next by the hypervisor's scheduler is critical itself or non-critical. These two cases are explained in the following with the example of Figure 6.4. Assume two VMs, V_1 is executed for two time units every five time units, V_2 is executed for three time units with the same period of five. From $t = 0$ until $t = 5$, the normal non-overloaded scheduling is depicted. The hatched rectangle in V_2's schedule illustrates that this execution time is based on the additional bandwidth U_{add}, whereas the grey rectangles are based on U_{min}.

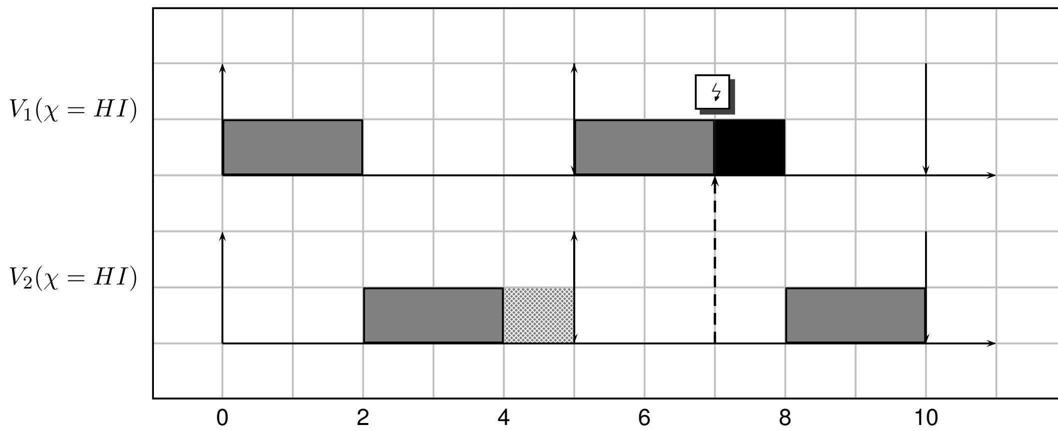
Assume now an overload of V_1 at point in time seven: it received already its minimum bandwidth, could however not finish the execution of its critical tasks. According to the normal schedule, the hypervisor would perform a VM context switch to V_2 and we assume that V_2 would be executed for a time slice of length l. That is to say, the hypervisor's scheduler would become active again in l time units, because either V_2 ran out of its budget or because it is preempted by a VM with higher scheduling priority.

In case of the next-to-dispatch VM V_2 being a non-critical guest (Figure 6.4(a)), the overloaded critical guest V_1 is executed for this time slice of length l instead of V_2. The hypervisor withdraws the amount l of V_2's budget and continues the execution of V_1. The black rectangle illustrates the computation time that was additionally assigned to V_1 as a reaction to the overload. The overloaded VM might use the entire time span for which V_2 would normally be executed. If this is not necessary, the overloaded VM yields and the non-critical VM is executed for the remaining fraction (not depicted in Figure 6.4). Note, since it does not occur in this simple example with only two VMs (V_2 is not preempted): budget reallocation as a reaction to an overload considers always individual time slices of the schedule, not the entire budget of a virtual processor.

In case of a critical guest being up next (Figure 6.4(b)), this drastic measure is not possible, since we have to avoid that an overload of a guest corrupts the correct execution of another critical guest. However, it is possible to execute a critical guest only for its bandwidth minimum U_{min}. Instead of assigning an additional bandwidth U_{add} to enable it to provide better results, it is more appropriate to protect the basic



(a)



(b)

Figure 6.4: Example of overload reaction: overload of V_1 at $t = 7$; (a) next-to-dispatch VM V_2 is non-critical; (b) next-to-dispatch VM V_2 is critical

functionality of the overloaded guest by passing the additional bandwidth. Therefore, the overloaded guest V_1 is executed instead of V_2, but only for V_2's execution time that is based on U_{add} (depicted by the hatched rectangle). V_1 obtains an additional execution time of l time units, if l is smaller than or equal to $\Pi_2 \cdot U_{add}(V_2)$, which is the execution time that results from U_{add} and can therefore be withdrawn. Otherwise, V_1 gets only $\Pi_2 \cdot U_{add}(V_2)$ additional computation time units and V_2 is executed for the remaining $l - \Pi_2 \cdot U_{add}(V_2)$ time units.

If the allocation of this additional computation time slice did not solve the overload condition, the next scheduled time slice is handled in the same way. The system designer might set a limit in terms of a time span for the continuing of this overload reaction, in order to avoid the system to be in this mode for indeterminate duration. Expected, however, is that the guest operating system signals a successfully corrected or unwinnable (deadline miss occurred already) situation. It is not possible to handle multiple overloads simultaneously.

The idea of this overload reaction is to help the overloaded guest immediately by transferring the entire or partial next schedule slice. Criticality inversion cannot occur anymore, since the overloaded critical guest is executed instead of the non-critical guest. The reallocation of bandwidth from another critical guest is safe because only the additional bandwidth U_{add} is touched. Schedulability is based on the minimum bandwidth U_{min}, which is still guaranteed. It is as well safe to transfer execution time of the giving guest before it is executed, since it just has to be guaranteed that the minimum bandwidth was provided at the end of the period. There is no benefit from providing it early.

6.5 Correctness of Bandwidth Distribution

In this section, we show that the proposed adaptive bandwidth management is safe, i.e., is characterized by the service guarantee defined as follows:

Theorem 6.5.1. ***Service Guarantee.*** *Let Θ_i^k denote the computation time allocation for VM V_i at time $(k-1) \cdot T_i$ (k^{th} instance). Given a set of periodic servers adaptively maintained by the presented bandwidth management policy: all servers receive in each instance a computation time allocation of at least $U_{min}(V_i) \cdot \Pi_i$:*

$$\forall V_i : \forall k : \Theta_i^k \geq U_{min}(V_i) \cdot \Pi_i \tag{6.6}$$

Note, the theorem is about the allocation at the beginning of the instance. A guest might voluntarily yield and reduce by itself the computation time for a specific instance when this is safe (because the deadlines were already met).

The next section shows that this theorem is true for the steady state. The following two sections present policies for mode change enforcement and dynamic slack passing that keep correctness as well during the transition phase.

6.5.1 Steady State: Temporal Isolation and Minimum Bandwidth Guarantee

In Section 6.1, temporal isolation was introduced as a key requirement for hypervisor-based consolidation of real-time systems and their certification. The presented computation bandwidth management approach fulfills this level of temporal isolation:

- *The timing requirements of the integrated guest systems can be validated independently.*
  As noted in Section 4.2, the design of the periodic server is based only on the characteristics of the associated guest system. The selection of the periods according to harmonic relations considers exclusively server dimensionings for which schedulability can be derived directly from the temporal interface of the guest system.

- *The hypervisor's VM scheduling provides all guest systems sufficient computation time in order to meet their real-time constraints.*
  Schedulability is enforced by the appropriate server design (the server's supply bound function is equal or greater than the guest's demand bound function for all t, see Section 4.2) and the partitioning algorithm (see Chapter 5), which guarantees to produce a schedulable mapping, i.e., with n VMs assigned to a specific core:
  $$\sum_{i=1}^{n} U_{min}(V_i) \leq 1.$$
  The hypervisor's scheduler ensures by the correct maintenance of all servers that each guest system receives its required computation bandwidth share. The specified minimum bandwidth allocation is guaranteed for all VMs. Distributed dynamically is only the spare bandwidth that remains after subtracting the U_{min} of all VMs.

  The online bandwidth allocation for a VM V_i is realized as an addition of a dynamic part ($U_{add}(V_i)$) to a static part ($U_{min}(V_i)$). Algorithm 3 computes under all circumstances for all VMs a value for $U_{add}(V_i)$ that satisfies $0 \leq U_{add}(V_i) \leq U_{lax}(V_i)$. The upper bound is a direct implication of Equation 6.4. The lower bound is true since the multiplication of two non-negative factors

(U_{spare} and normalized weight $\hat{w}(V_i)/\hat{w}^{\Sigma}[\chi]$) results in a non-negative number. From an implementation perspective, $U_{min}(V_i)$ and $U_{add}(V_i)$ are realized as different parameters in the VM control block and only the latter is modified at runtime. It is therefore precluded that the allocated bandwidth falls below $U_{min}(V_i)$, implying that an execution time budget of at least $\Theta_i = U_{min}(V_i) \cdot \Pi_i$ is allocated every instance, as defined by Theorem 6.5.1. This minimum budget is not touched by the dynamic bandwidth management. The adaptive bandwidth distribution might just add budget.

Note, a guest might voluntarily yield and reduce by itself the computation time for a specific instance, which is analyzed in Section 6.5.3.

The combination of correct determination of U_{min}, a partitioning that ensures $\sum_{i=1}^{n} U_{min}(V_i) \leq 1$, and a bandwidth distribution algorithm that produces for each VM a $U(V_i) \geq U_{min}$ with $\sum_{i=1}^{n} U(V_i) \leq 1$ guarantees that all VMs receive sufficient computation time to meet their real-time requirements.

- *Overruns within a VM provoke under no circumstances that other VMs violate their timing requirements, since a VM is never executed if the associated server ran out of budget.*
  Based on the appropriate setting of the programmable interval timer, the depletion of a server's budget leads immediately to an interruption of the guest system and the execution of the hypervisor's scheduler. Exception is the prevention of criticality inversion: in this case the overrun of a critical VM provokes that a non-critical VM does not receive sufficient budget, explicitly desired.

- *The execution of a guest system is never interrupted for a longer time than its reactivity requirements demand.*
  The selection of the server dimensioning as part of the partitioning algorithm considers exclusively solutions that do not violate the largest affordable blackout phase, as introduced in Section 5.3.1.

The combination of the provided degree of temporal isolation and the guaranteed minimum bandwidth allocations ensures the correct execution of the guests in terms of their timing constraints, independent from the execution of other VMs on the same core. Therefore, these guaranteed minimum bandwidths and the described temporal isolation are the basis for both the schedulability analysis and a potential certification. An actual certification requires the proper determination (or safe overestimation) of the minimum bandwidths in due consideration of the inter-core interferences through shared memory and bus interference. Very pessimistic minimum

bandwidths can be expected if these interferences are considered, but the presented approach has the benefit of reclaiming unused capacity at runtime, thus making an efficient use of the processor even in such a situation.

So far, correctness was shown for the steady state. The next section covers correctness of the mode transition phase and defines a policy that leads to safe mode transitions, guaranteeing the minimum bandwidths in this phase as well.

6.5.2 Correctness during Mode Transitions

A resource management for critical systems must be safe and must guarantee to fulfill the computation time requirements of the hosted guest systems at all times, including the mode transition phase. This section shows how mode changes are enforced in order to allocate a bandwidth of at least U_{min} to all VMs as well during the instances that are affected by the mode change.

Performing each mode change immediately is not an option, as it might cause timing problems. Consider an example with two VMs of same criticality:

- V_1 with $U_{min}(V_1) = 1/3$, $U_{lax}(V_1)^{M1} = 0$, $U_{lax}(V_1)^{M2} = 2/3$ and a mode-independent weight of $\hat{w} = 1/3$
- V_2 with $U_{min}(V_2) = 1/6$, $U_{lax}(V_2) = 5/6$, $\hat{w} = 2/3$

Assumed that V_1 is first in mode $M1$ in which it cannot benefit from additional computation time ($U_{lax}(V_1)^{M1} = 0$), the spare bandwidth of 3/6 is given entirely to V_2, resulting in a budget of 1 for V_1 and 4 for V_2 at point of time zero (see Figure 6.5). The hatched rectangles illustrates that this execution time is based on the additional bandwidth U_{add}, whereas the grey rectangles are based on U_{min}. V_1 requests a mode change from $M1$ to $M2$ at point in time 4. In this mode, it receives one third of U_{spare}, leading to a budget of two time slices per period and reducing the allocation to V_2 to three time slices per period. However, V_2 received a budget of four time slices for its current instance, based on the previous situation. A forced reduction of the budget during an instance by the hypervisor is not an option, as it might cause consistency problems for the guest system. A guest might have to perform an internal mode change in order to function correctly with the reduced bandwidth allocation. Performing V_1's mode change immediately would result in a total allocation of more computation time than available on the processor.

In the following, the different cases of adaption are analyzed in detail one after the other. For each case, the transition policy is introduced and it is shown that

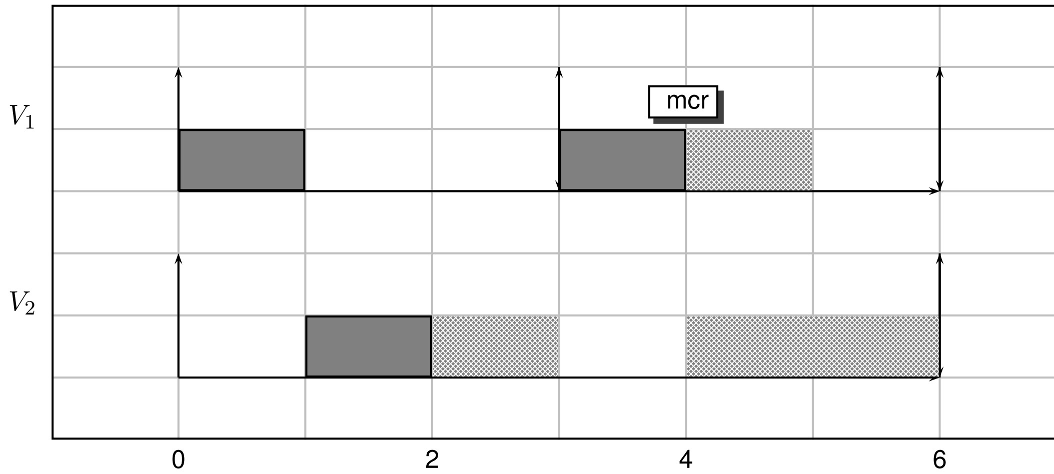


Figure 6.5: Mode change request of V_1 at $t = 4$: immediate mode change leads to overallocation

each VM receives the minimum allocation as well during the transition phase.

Case 1: VM is enabled

In order to avoid the necessity of online acceptance tests, the partitioning algorithm produces only partitions with guaranteed schedulability for the case that all VMs are enabled:

$$\sum_{i=1}^{n} U_{min}(V_i) \leq 1.$$

It is therefore not possible that the enabling of a VM leads to a violation of the service guarantee.

Transition Policy: Without loss of generality, assume that V_1 was disabled before and is now enabled. This results in a new value for U_{spare}, since $U_{min}(V_1)$ has to be subtracted. Consequently, U_{add} is newly computed according to Algorithm 3 for all VMs. The enabled VM is activated at the end of the last finishing instance of all currently enabled VMs, as depicted by an example in Figure 6.6. At this point in time, all other instances end as well, based on the harmonic relationship: all smaller periods divide the longest period, so a new hyperperiod starts. From this point on, all VMs receive the new allocation, but there is no need to enforce the new (smaller) allocation for any VM before this point. This transition policy is safe, since there is no situation in which some VMs receive the old allocation and others receive already the new allocation. In addition, it can be applied to enable multiple VMs at the same time.

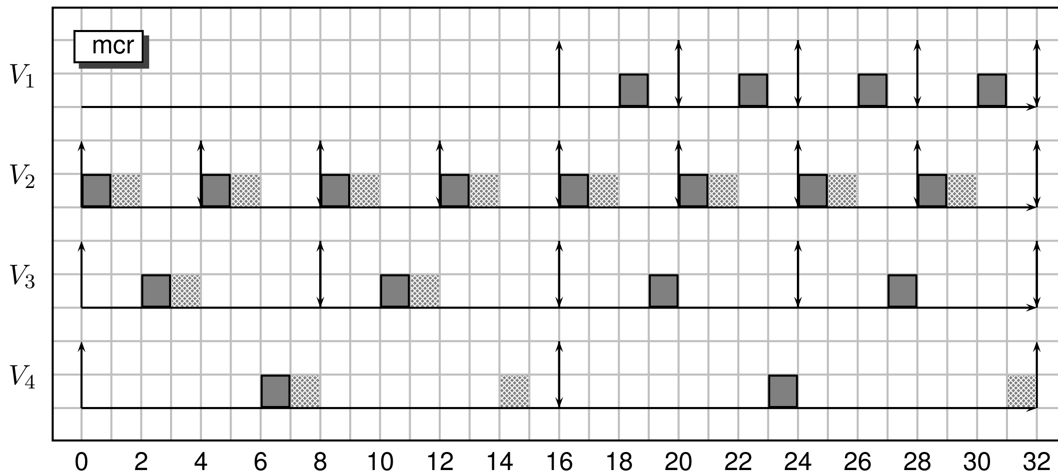


Figure 6.6: Case 1: request to enable V_1 at $t = 1$, activation at $t = 16$

Case 2: VM is disabled

Transition Policy: When a VM is disabled, its impact lasts for the entire length of its current instance. Figure 6.7 depicts an example with the request to disable V_2 at point in time $t = 14$. The released bandwidth can be used to provide the other VMs additional bandwidth, however starting not before the end of V_2's instance. Again, there is no situation in which some VMs receive the old allocation and others receive already the new allocation and therefore this transition policy is safe.

Case 3: VM Mode Change

This case deals with the situation that a VM changes to a mode with a differing U_{lax}. The set of enabled VMs does not change.

Transition Policy: A mode change of a VM invokes the bandwidth distribution algorithm, with potentially new U_{add} values. Since the spare bandwidth is distributed in a greedy manner in order of decreasing criticality, a mode change of a specific VM cannot result in new U_{add} values for all VMs of higher criticality, but just for VMs of same and lower criticality. After the computation of the new distribution, the new allocations are activated comparably to the introduced policies for the enabling and disabling of a VM. In case of a change to a mode with a higher utilization laxity U_{lax}, the higher allocation is activated at the end of the last finishing instance of all currently enabled VMs of same or lower criticality. From this point on, all VMs receive the new allocation. An example is given in Figure 6.8.

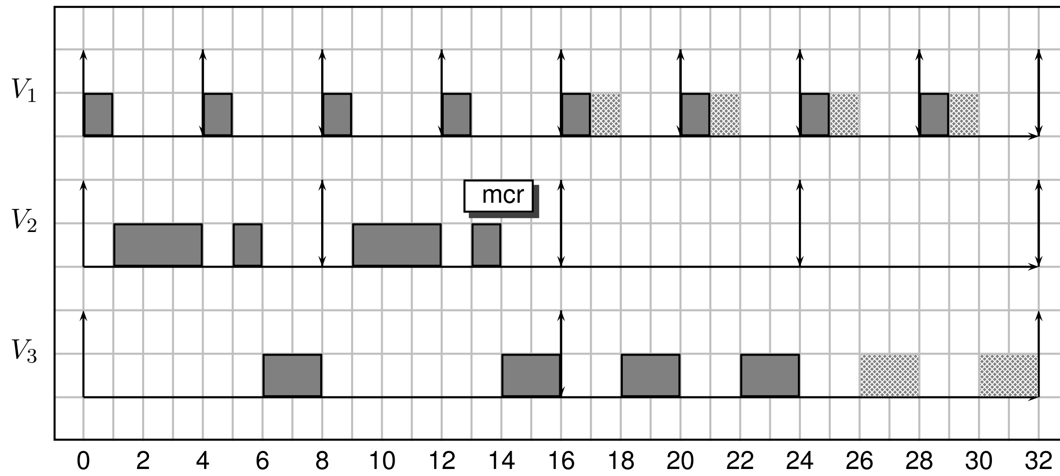


Figure 6.7: Case 2: request to disable V_2 at $t = 14$, activation at $t = 16$ (end of V_2's instance)

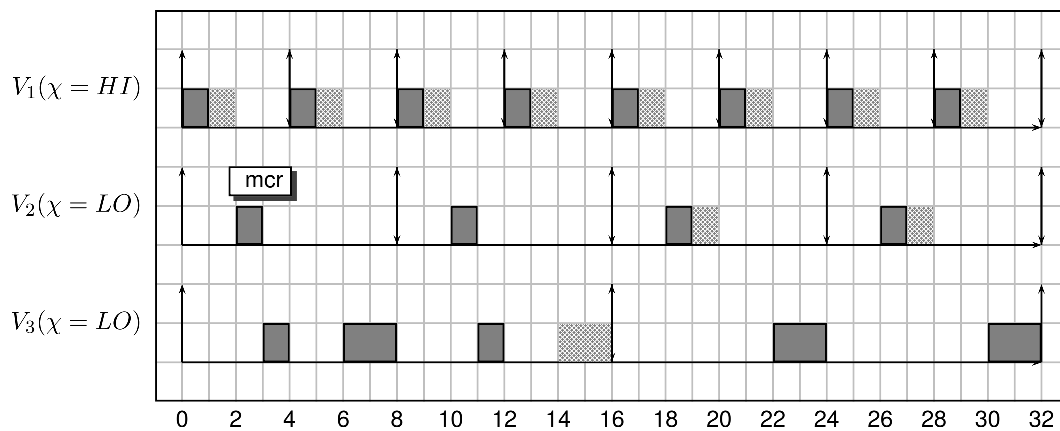


Figure 6.8: Case 3: mode change request of V_2 at $t = 3$ (from a mode with $U_{lax}(V_2) = 0$ to a mode with $U_{lax}(V_2) = 0.125$; assumed is a higher weight of V_2 compared to V_3): the allocation to V_1 of higher criticality is unchanged, V_2's larger allocation is activated at the end of V_3's instance (V_3's allocation is reduced at the same point in time)

In case of a change to a mode with a lower utilization laxity, the new allocation is activated at the end of the current instance. VMs of same or lower criticality receive their potentially larger allocation with start of their next instance, but not before the end of the current instance of the mode-changing VM. In both cases, this transition policy is safe, since there is no situation in which some VMs receive the old allocation and others receive already the new allocation.

6.5.3 Correctness of Redistribution of Dynamic Slack

The just discussed mode changes are handled in a safe manner by phasing out the old allocations, so that there is no situation in which old allocations and new allocations are active at the same time. We have seen before that both the old distribution and the new distribution are safe, guaranteed by the distribution algorithm. This is an appropriate solution for mode changes, which happen with an inter-arrival time of multiple periods. Moreover, this policy avoids a forced reduction of the budget during an active instance, which is likely to cause problems for the guest system.

However, we cannot apply the same policy to the distribution of dynamic slack. First, we do not have to delay. We delayed the activation of the smaller budget in the case of a redistribution on mode change to the beginning of the next period, since a forced reduction of the budget during an instance might cause inconsistencies. But this is not true in the case of a yield, since the reduction takes place explicitly and intentionally by the affected guest. Second, the passing of dynamic slack is a punctual change that has to be reverted with the start of the next instance of the giving VM. Therefore, it does not make any sense to delay the redistribution to the end of the current instance of the giving VM. Dynamic slack is characterized by a limited validity duration, as explained with the example of Figure 6.9.

Assume that V_1 is first in a mode in which it cannot benefit from additional bandwidth ($U_{lax}^{M1}(V_1) = 0$) and receives an allocation according to its $U_{min}(V_1) = 1/2$. V_2 receives an allocation according to $U_{min}(V_2) = 1/6$ plus $U_{add}(V_2) = 2/6$. At point in time $t = 7$, V_1 requests a mode change, we assume to a mode in which it can benefit from additional bandwidth ($U_{lax}^{M2}(V_1) = 1/2$). In addition, we assume that the weight of V_1 is zero, so V_1 does not receive any structural slack. This mode change is implemented at the end of V_2's current instance, as introduced previously. At point in time $t = 8$, V_2 yields and passes two time units of dynamic slack to V_1. V_1 cannot use it in this moment, but might keep it for the next instance (after the already scheduled mode change). In this case, however, V_2 is not scheduled again before $t = 17$, since it has a lower priority than V_1. For its period from $t = 12$ to

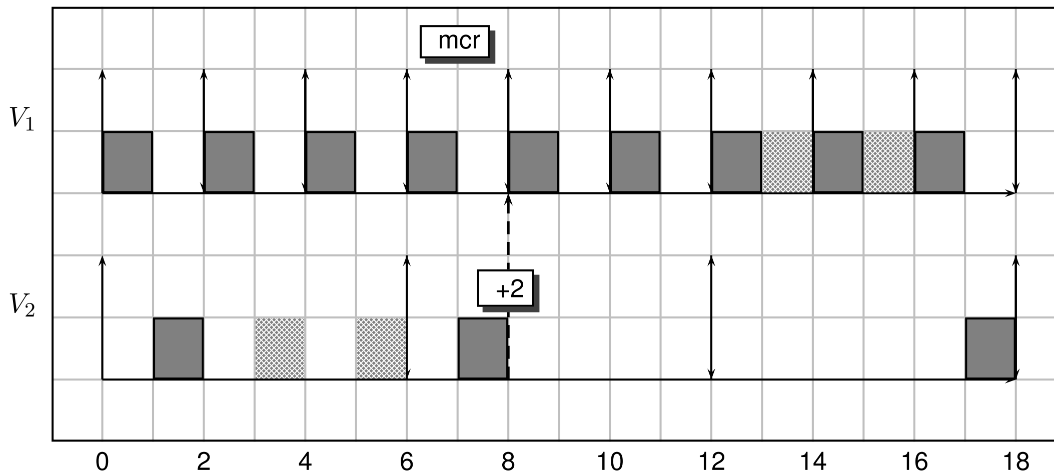
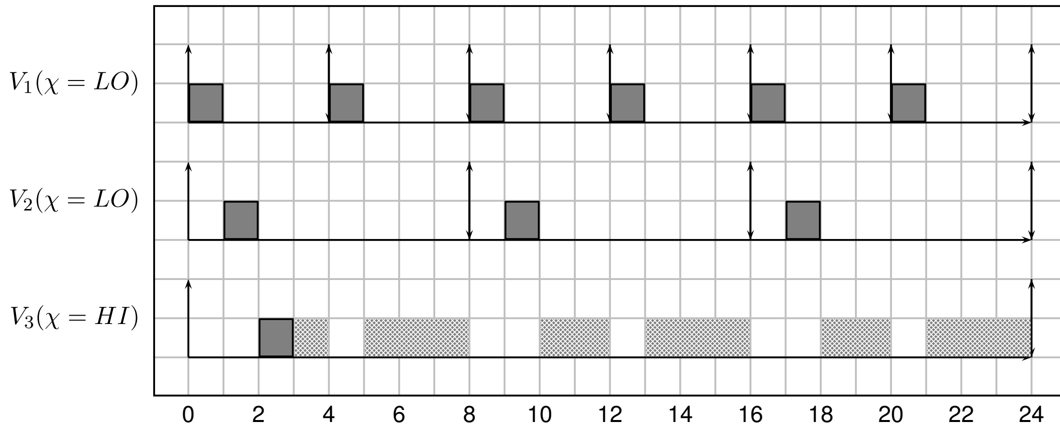


Figure 6.9: Limited validity duration of dynamic slack: timing violation due to use of dynamic slack later than the end of the period of the giving virtual machine

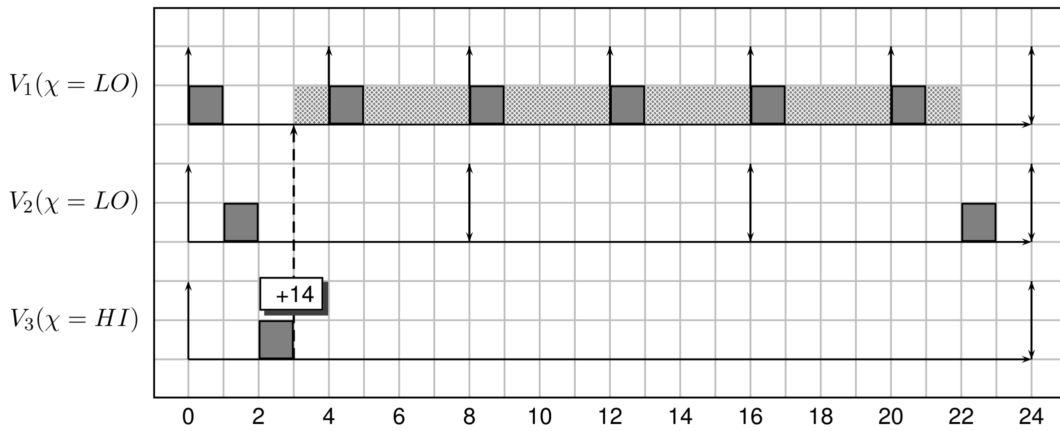
$t = 18$, V_2 does therefore not receive the correct allocation of 3/6, but only 1/6. The time interval $12 \leq t \leq 18$ is overallocated: V_1 received 5/6 (it received dynamic slack and did not use it before) and V_2 received 3/6.

To prevent this kind of overallocation, received dynamic slack can only be used until the end of the instance of the giving VM. But the use of dynamic slack has to be restricted even more, as the example of Figure 6.10 makes clear. Subfigure 6.10 (a) depicts the schedule without yield. V_3 is the only critical VM and receives therefore the entire U_{spare}, leading to a large additional bandwidth of $U_{add}(V_3) = 14/24$. Moreover, we assume that V_2 cannot and that V_1 can benefit from additional bandwidth and would actually fully utilize the processor if possible ($U_{lax}(V_2) = 0, U_{lax}(V_1) = 6/8$). In Subfigure 6.10 (b), V_3 yields at point in time $t = 3$, generating a slack of 14, which is passed completely to V_1. By consequence, V_2 is not executed at all within its period from $t = 8$ to $t = 16$ and misses its deadlines.

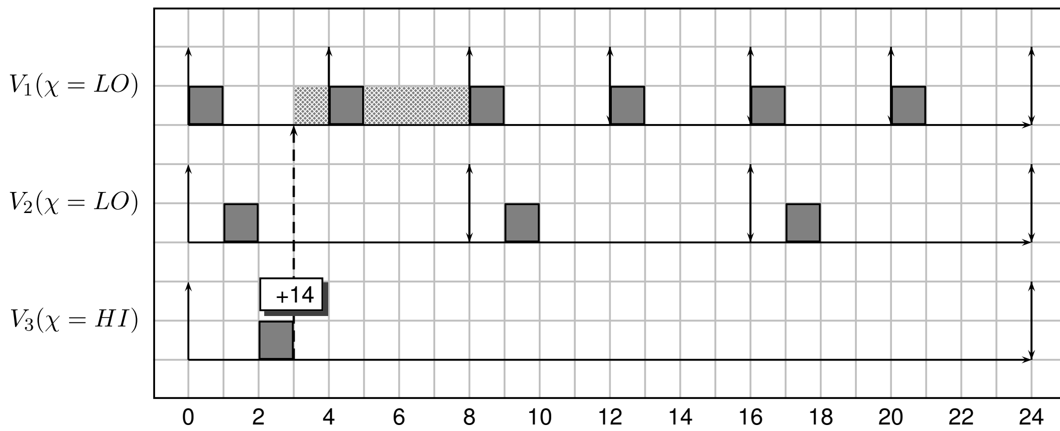
For this erratic situation to occur, slack has to be passed from a low-priority VM to a high-priority VM, with the VM of intermediate priority suffering. While the 14 execution time slices were not critical for the VM of intermediate priority V_2 as long as executed with a lower priority by V_3, they prevent V_2 from being executed when executed with a higher priority by V_1. The interference of V_2 by V_1 causes the timing violation, which becomes clear when applying Audsley et al.'s response time analysis [Audsley et al., 1991]. According to this method, the longest response time R_i of a periodic schedulable entity is given by the sum of its own computation



(a) Without yield.



(b) With yield and slack passing (unsafe).



(c) With yield and slack passing and slack expiry time (safe).

Figure 6.10: Limited validity duration of dynamic slack: timing violation of VM of intermediate priority due to use of dynamic slack later than the end of the instance of the receiving VM (example: $U_{min}(V_1) = 1/4$, $U_{lax}(V_1) = 3/4$, $U_{min}(V_2) = 1/8$, $U_{lax}(V_2) = 0$, $U_{min}(V_3) = 1/24$, $U_{lax}(V_3) = 23/24$)

time (in our case Θ_i) and the interference I_i caused by preempting entities of higher priority. For schedulability, this response time must be not greater than the period Π_i (assumed is a task ordering by decreasing priority: $i < j \Leftrightarrow \Pi_i < \Pi_j$):

$$R_i = \Theta_i + I_i = \Theta_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{\Pi_j} \right\rceil \Theta_j \text{ [Audsley et al., 1991]}$$

The passing of dynamic slack from V_3 to V_1 *over* V_2*'s head* increased the interference time I_2 and caused the timing violation. We obtain the following response time for V_2's second period:

$$R_2 = \Theta_2 + I_2 = 1 + 14 \nleq 8 = \Pi_2$$

To preclude such timing violations, dynamic slack obtains an expiry time, which is communicated to the receiving VM with the notice that dynamic slack was passed. In the following, V_{giving} refers to the slack releasing VM (d_{giving} to its absolute deadline), $V_{receiving}$ refers to the slack receiving VM, and $V_{intermediate}$ refers to a VM with a priority in-between these two VMs. The expiry time is defined as (with p being the priority of a VM: the priority is based on Π_i according to RM, with the VM index as a tie breaker, so that two VMs do not have the same priority):

$$\begin{aligned} & t_{expiry} = min\{d_{giving}, d_{intermediate}\} \\ & \text{with} \quad d_{intermediate} = min\{d_i | p_{receiving} < p_i < p_{giving}, \} \end{aligned} \tag{6.7}$$

Note that t_{expiry}= d_{giving} if there is no intermediate task between the receiving and the giving VM.

Dynamic slack is distributed in the weighted manner as introduced in Section 6.4.1, but differs from the distribution of structural slack in this expiration mechanism. A guest system can receive slack from multiple other guests with differing expiry times. Both hypervisor and operating system have to keep track of this.

In the following, we show that this dynamic slack distribution is safe. Figure 6.11 shows the different classes of VMs according to their priority relative to the slack-giving VM and slack-receiving VM for the case that the receiving VM is of higher priority. All VMs fall into one of these five categories. We are going to show that the passing of slack is safe according to Theorem 6.5.1 for all of them. This categorization assumes that the entire slack is passed to a single receiving VM. If fractions of slack are passed to multiple receiving VMs, or if a VM receives slack from multiple giving VMs, the situation can be analyzed as multiple individual slack transfers from one VM to another VM.

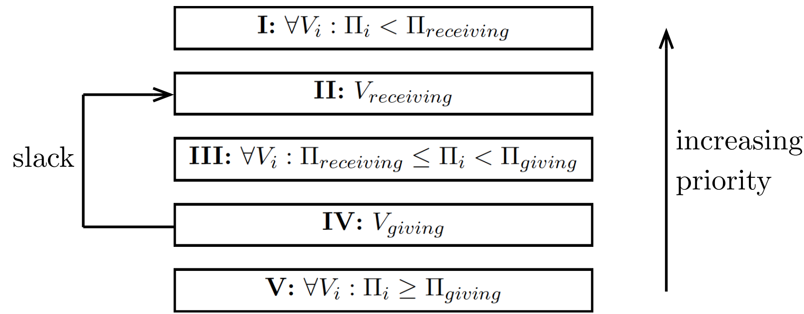


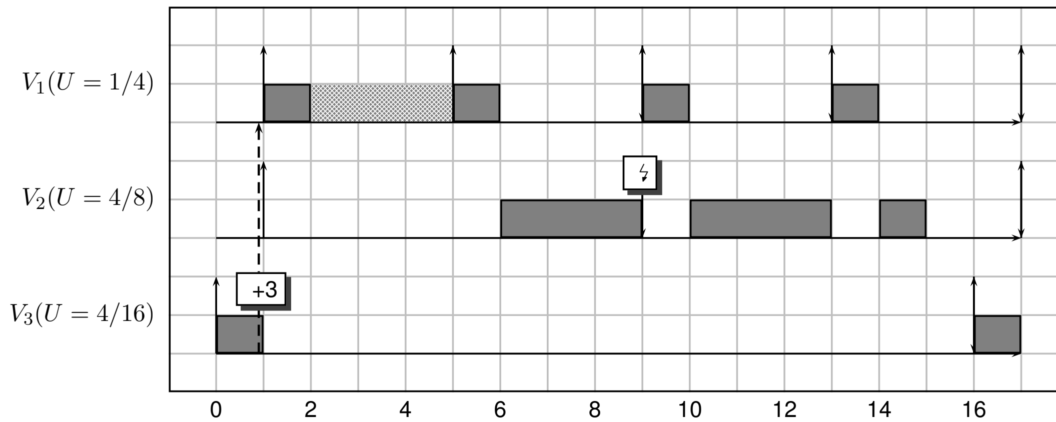
Figure 6.11: Categorization of all VMs based on priority relative to giving virtual machine and receiving virtual machine for the case that V_{giving} is of lower priority

Class I: The VMs of higher priority than the receiving VM are safe, since there execution is uninfluenced from the slack passing: the interference time is unchanged.

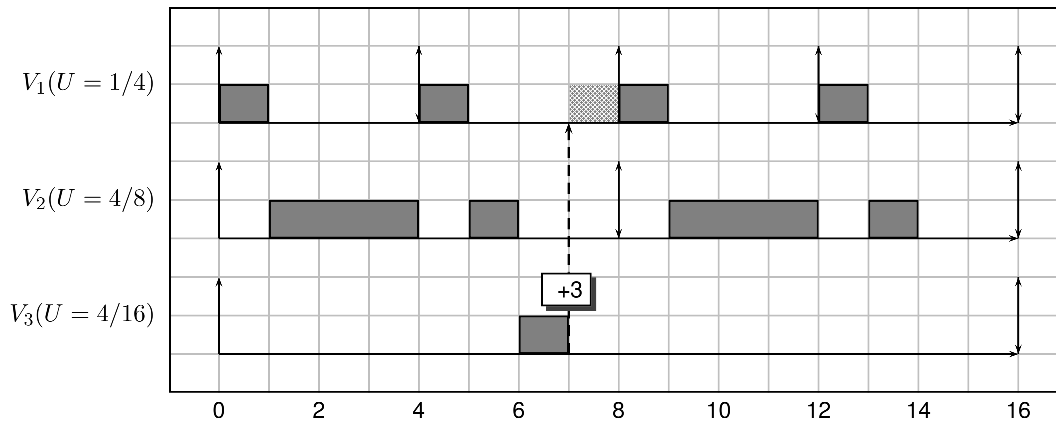
Class II: The receiving VM itself is safe, since its interference time is unchanged and it gets more execution time than planned and required to meet its deadlines.

Class III: The VMs with a lower priority than the receiving VM, but a higher priority than the giving VM (intermediate priority) are safe due to the expiry time according to Equation 6.7, which is not greater than the smallest time until the end of the current instance of all VMs of intermediate priority. Note that multiple instances are executed within the considered instance of the giving VM, since VMs of intermediate priority have a smaller period than the giving VM and all periods are harmonic. The first instance within the instance of the giving VM is safe, since it completed already, otherwise the lower-priority giving VM would not be executed and could not yield. It is crucial that the hypervisor starts the first instance of all VMs at the exact same time $t = 0$, as denoted in Section 6.3.2, as it is otherwise not guaranteed that the first instance completed already, as illustrated in Figure 6.12. The subsequent instances are safe, since they are not influenced by the slack passing due to the limited validity of the slack: the slack expired with the first end of the current instance of all VMs of intermediate priority. This is depicted in Figure 6.10 (c): the slack expires at $t = 8$ and can therefore not block the second or third instance of V_2. This is safe, however, less slack can be used by the receiving VM.

Class IV: The giving VM is safe: it yielded by its own choice since it completed its execution in the current instance. The next instance is not influenced.



(a) Unsynchronized: deadline miss of V_2.



(b) Synchronized.

Figure 6.12: Need to activate virtual machines in a synchronized manner (fractions denote Θ and Π)

Class V: The VMs of lower priority than the giving VM are safe, since their interference time is unchanged. It does not make a difference to them whether they are interfered by the giving VM or the receiving VM (both of higher priority).

Next, we analyze the safeness according to Theorem 6.5.1 for slack passing in the other direction, i.e., the receiving VM is of lower priority than the giving VM. Figure 6.13 depicts the different classes of VMs according to their priority relative to the slack-giving VM and slack-receiving VM and we show that the passing of slack is safe for all of them.

Class I: The VMs of higher priority than the giving VM are safe, since there execution is uninfluenced from the slack passing: the interference time is un-

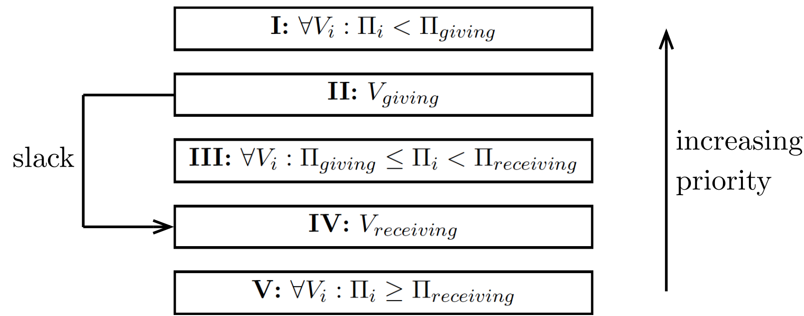


Figure 6.13: Categorization of all virtual machines based on priority relative to giving virtual machine and receiving virtual machine for the case that V_{giving} is of higher priority

changed.

Class II: The giving VM is safe: it yielded by its own choice since it completed its execution in the current instance. The next instance is not influenced.

Class III: The VMs with a lower priority than the giving VM, but a higher priority than the receiving VM are safe, since their interference time is reduced.

Class IV: The receiving VM itself is safe, since its interference time is reduced due to the smaller execution time of the giving VM and it gets more execution time than planned and required to meet its deadlines.

Class V: The VMs of lower priority than the receiving VM are safe, since their interference time is unchanged. It does not make a difference to them whether they are interfered by the giving VM or the receiving VM (both of higher priority).

By consequence, it was shown that the slack passing policy is safe for all VMs in both cases, slack passing to a VM of higher priority and to a VM of lower priority. The remaining case slack passing to a VM of same priority does not exist, since the VM index is used as a tiebreaker for VMs of equal period.

Important prerequisite, the yield of a VM keeps the general schedulability of the VM set. Baruah and Burns defined a schedulability test as *sustainable* if any system deemed schedulable remains schedulable when the parameters of one or more individual job(s) are changed, among others, by decreased execution requirements [Baruah and Burns, 2006, Burns and Baruah, 2008]. In addition, they showed that the utilization-based schedulability analysis for fixed-priority scheduling by Liu and Layland [Liu and Layland, 1973a] and especially as well the one considering harmonic

relations by Kuo and Mok [Kuo and Mok, 1991] are in fact sustainable. This means that a set of servers that are scheduled by RM and passed the schedulability test, cannot become unschedulable by a yield of a VM (i.e., a reduction of its execution requirements).

6.5.4 Handling of Multiple Mode Change Requests

As mode changes are not performed immediately upon mode change request, a mode change request might occur while a mode change is still pending. This section introduces how multiple mode change requests are handled. The request to perform mode changes for multiple VMs typically occurs when the controlled system enters a different state, e.g., a different operational mode or stand-by mode. This is no problem, since the bandwidth distribution algorithm can handle resource requirement changes of multiple VMs.

If a guest requests a mode change before the hypervisor enforced the new allocation resulting from a pending mode change of the same guest, the first mode change request is discarded and a new allocation based on the second mode change is enforced. If mode changes to a mode with a greater U_{lax} or to enable the VM are requested for two or more different VMs, they are all performed at the same point in time, i.e., the end of the last finishing instance of all currently active instances. If mode changes to a mode with a smaller U_{lax} or to disable the VM are requested for two or more different VMs, they are performed together only if the associated VMs have the same period. Otherwise, the mode changes are performed individually at the end of the current instance of each associated VM.

If different kinds of mode changes are requested for two or more different VMs, then the resource requesting ones are performed together as introduced, the resource releasing ones are performed individually. The distribution algorithm is executed at each mode change. The distribution of dynamic slack is independent from the distribution of structural slack.

6.6 The Case for Paravirtualization

The capability to host virtualization-unaware operating systems classifies hypervisors, as discussed in Chapter 2. *Full* (also known as *transparent* or *pure*) *virtualization* allows hosting unmodified guest operating systems. The same binary can be executed on a bare machine or by the hypervisor. Contrary, if an OS has to be ported to the hypervisor's application programming interface (API) in order to be

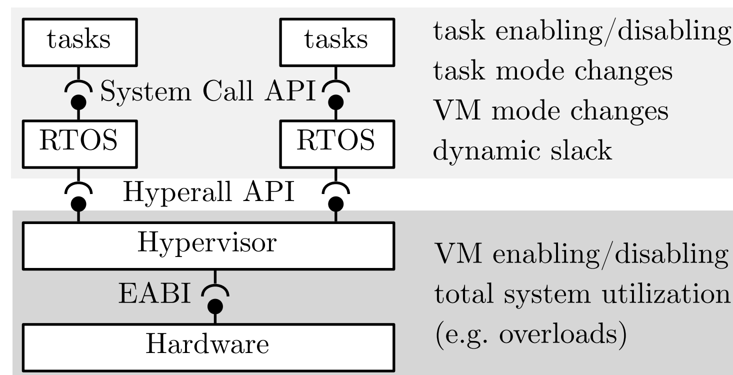


Figure 6.14: Availability of information with influence on scheduling for hypervisor and guest systems

able to execute it within a virtual machine, it is called *paravirtualization* [Barham et al., 2003]. In this case, the guest OS is aware of being executed in a virtualized manner and uses *hypercalls* to request hypervisor services.

Figure 6.14 illustrates the different levels of the system stack and the interfaces between them. The operating system offers a system call API to the application tasks, a programming interface to the services provided by the OS (e.g., access to I/O devices or inter-process communication). A paravirtualized hypervisor offers a hypercall API to the hosted operating systems. As an example, Section 3.3.4 introduced the hypercall interface of Proteus, including for example a hypercall to pass scheduling information. Finally, the embedded-application binary interface (EABI) of the processor architecture specifies for software and development tools conventions for data types (sizes and alignments), register usage, stack frame organization, function parameter passing etc. to assure compatibility.

The implementation of the proposed adaptive scheduling requires virtualization awareness of the OS, and thereby paravirtualization. An explicit communication between hypervisor and guest OS is mandatory and the OS has to be modified accordingly. The OS has to provide the hypervisor a certain level of insight in order to support the hypervisor's bandwidth assignment, which can be done only by a virtualization-aware OS. The hypervisor in turn informs the guest OSs about the assigned bandwidth share, since they need this information in order to distribute the bandwidth among their tasks.

Figure 6.14 shows as well the availability of scheduling related information above and below the border between hypervisor and guest OS. Communication from OS to hypervisor is mandatory to inform the hypervisor about adaptation triggering

events such as mode change of a task (incl. enabling/disabling) or mode changes of the entire guest system. In addition, instead of running the idle task, a paravirtualized guest OS can yield to pass the dynamic slack and enable the hypervisor to execute another ready VM. Due to this availability of certain scheduling related information only within a guest OS, the approach requires communication and thereby paravirtualization.

Actually, any adaptive scheduling technique with the ability to react to events that are only known within the guest system implies paravirtualization. This is true as well for feedback control based scheduling approaches, since the controlled variables such as number of deadline misses or the utilization of a VM are otherwise unknown to the hypervisor. Independently, two related works recognized the benefits of paravirtualization for real-time VM scheduling. Kiszka paravirtualized Linux in order to give the hypervisor (Linux with KVM) a hint about the internal states of its guests [Kiszka, 2011]. Lackorzynski et al. demonstrated the limitations of a hierarchical scheduling that handles guest scheduling as a black box for mixed-criticality systems [Lackorzynski et al., 2012]. As a consequence, they propose to *flatten* hierarchical scheduling by exporting scheduling information from the guest to the host.

Paravirtualization is anyway the prevailing approach in the embedded domain [Gu and Zhao, 2012]. The need to modify the guest OS is outweighed by the advantages in terms of efficiency (reduction of the overhead [King et al., 2003], also seen for Proteus in Section 3.4.2) and in terms of the benefits of an explicit communication and the hereby facilitated cooperation of hypervisor and guest OS. This cooperation is in addition to scheduling as well very helpful for resource sharing, e.g., of I/O devices. Finally, paravirtualization allows the efficient application of virtualization even on processor architectures that are not trap-and-emulate virtualizable according to the theorem of Popek and Goldberg, as discussed in Section 3.3.3.

The major drawback of paravirtualization is the need to port an OS to the hypervisor's interface, which involves modifications of critical kernel parts. If legal or technical issues preclude the modification of an OS, it is not possible to execute it in a paravirtualized manner on top of the hypervisor. For this reason, Proteus offers both paravirtualization and full virtualization (see Section 3.3.4). Paravirtualized guests can be executed next to fully virtualized guests, however, the adaptive scheduling technique is restricted to paravirtualized guests. Fully virtualized guests receive a fixed periodic computational bandwidth allocation.

6.7 Integration into Hypervisor and Operating System

Paravirtualization is a technical implication from the required communication between operating system and hypervisor. The communication is realized by two different techniques, namely hypercalls and shared memory. Hypercalls are used by the operating system to pass information and control immediately to the hypervisor. A context switch occurs and the hypervisor handles the hypercall. As introduced in Section 3.3.4, Proteus offers a hypercall interface with two scheduling-related hypercalls: `sched_set_param(void* param, void* val)` and `sched_yield()`. These two hypercalls are actually sufficient for the implementation of the required communication from OS to hypervisor. `sched_yield()` is called by an operating system to notify the hypervisor that it does not need the reserved worst-case computation time demand and would therefore start to idle. The hypercall `sched_set_param` is generic and used for all other communication purposes (based on the specification of multiple parameters `param`), especially in order to inform about a mode change to a mode with a differing laxity utilization U_{lax}.

Hypercalls are a one-way communication mechanism. Communication in the other direction is needed as well, as the hypervisor informs the guest OS about changes in bandwidth allocation. For this direction, shared memory communication is used. A memory region within the memory space that is assigned to a VM is dedicated to paravirtualization communication. It is accessible by hypervisor and corresponding VM, however not by any other VM. Even the access of the hypervisor is additionally restricted: the shared memory is only accessible by the hypervisor if it is executed on the same core as the VM. The need for a multicore-safe access synchronization is avoided, resulting in a lower execution time overhead. A paravirtualization communication library for shared memory access is provided. A guest OS includes this library and then calls the methods `para_write(void* val)` and `void* para_read()`. The correct implementation of the shared memory access is therefore not the responsibility of the software engineer who paravirtualizes the OS.

Main required modification for both hypervisor and operating system is the addition of the protocol-compliant passing of scheduling information. Figure 6.15 shows the basic process with a simplified example of just one guest (therefore, the call of the function `schedule` is missing). In case of a mode change, the guest OS informs the hypervisor by the hypercall `sched_set_param` and passes the desired utilization laxity of the new node. The hypervisor performs a bandwidth redistribution and writes the result into the shared memory. In addition, before resuming the execution of a guest, it sets a processor register in order to inform the guest about the changed

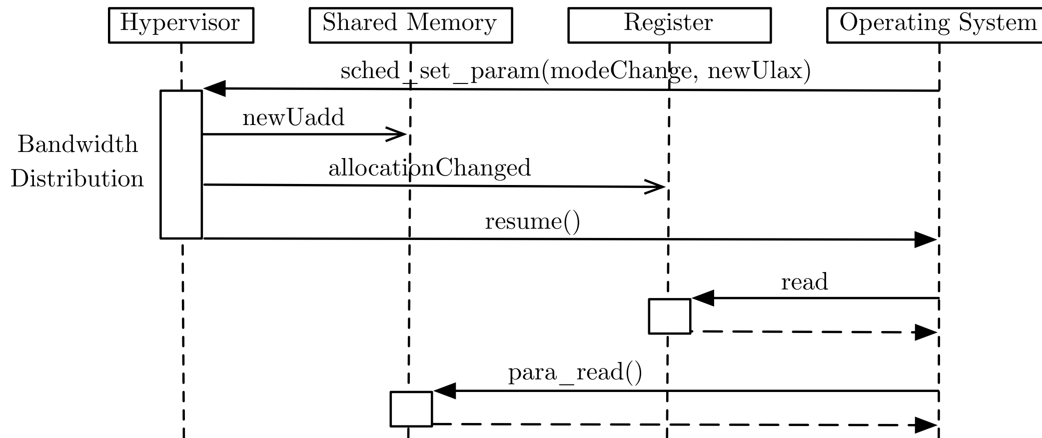


Figure 6.15: Interaction between hypervisor and operating system in the case of redistribution on mode change

bandwidth allocation. After the context switch from hypervisor to OS, the OS checks this register and, if the allocation was modified, reads out the shared memory.

6.8 Evaluation

The evaluation is based on two evaluation platforms: the hypervisor Proteus as introduced in Chapter 3 and a scheduling simulator. The implementation of the scheduler as part of the hypervisor enables the determination of real execution times on a typical embedded processor. This is of particular importance because this work is motivated by an improvement of the utilization of the processor. This is only the case if the benefit exceeds the overhead, which has to be evaluated by measuring the real execution times of a prototype.

Drawback of the evaluation with a prototype on real hardware is the high effort to conduct experiments with a large number of workloads and the limited possibility to configure the hardware platform. A scheduling simulator, on the other hand, enables the evaluation of multiple orders of magnitude larger numbers of workload configurations. The used scheduling simulator is introduced in the next section.

6.8.1 Scheduling Simulator

The real-time scheduling simulator *RTSIM* (developed at Retis Lab, Scuola Superiore Sant'Anna [Bartolini and Lipari, 2014]), was extended for our purpose. RTSIM is a discrete event simulator for scheduling algorithms and real-time tasks: all simulated

entities can change their state at certain discrete events. As a functional scheduling simulation, it abstracts from all details of hardware and software that are irrelevant for task scheduling. It models the processor just by the number of cores and tasks as schedulable entities without functionality. Tasks are characterized by a period (or a probabilistic arrival pattern), a phase (activation time of the first periodic instance), a worst-case execution time, and potentially by a fixed priority. Scheduling according to EDF, RM, fixed priorities, and first-in-first-out (FIFO) is supported. Resources can be defined and shared in a synchronized manner, with resource management based on first-come-first-serve or priority inheritance. Interrupts can arrive according to different probability distributions and trigger the execution of tasks. RTSIM includes a graphical user interface for the specification and modification of all properties of the system and a visualization of the schedule traces.

We built on this open source simulator and added the following functionality:

- Virtual Machines. VMs are specified by task set, task scheduler (RM, EDF, fixed priorities, or FIFO), U_{min} and U_{lax}, criticality, period and budget of the associated virtual processor, processor core affinity, whether it yields in case of not needing the worst-case demand or not, potentially a round robin slice, and potentially a fixed priority.

- Hierarchical Scheduling. Instead of a mere task scheduling, two dependent scheduling levels are simulated: the VM scheduler decides for each point in time and per core which VM to execute and the associated task scheduler schedules the execution of the VM's internal tasks. VMs can be scheduled according to fixed time slices within a repetitive cycle or based on computation time servers. In the latter case, it is possible to have fixed budgets, or redistribute in the event of a mode change, or redistribute in addition dynamic slack in case of a yield.

- Multi-mode. Both tasks and VMs can have multiple operational modes, which differ regarding computation bandwidth demand (see Section 4.1.2). Mode transitions can be triggered at fixed times, with a random probability, or conditionally (e.g., in case of a deadline miss).

- Generation of Random Workloads. It is possible to automatically generate an arbitrary number of synthetic workloads based on Brandenburg's toolkit SchedCAT, which was already introduced in Section 5.4 (originally, the workload configuration for a simulation run had to be entered manually). An individual simulation run is started for each configuration.

- Automated Analysis. Each simulation run is analyzed automatically regarding key metrics, such as number of deadline misses or processor utilization. Before, the simulator produced only a scheduling diagram, which does not allow to analyze many simulation runs or long simulation durations.

In a nutshell, we extended RTSIM by the virtual machine model as introduced in Section 4.1.2, hierarchical scheduling of VMs and tasks, VM scheduling algorithms (time-slice based and server based), and a pre- and post-processing of the actual simulation run that enables the automated simulation of many random configurations.

6.8.2 Execution Times

The approach was implemented on an IBM PowerPC 405 multicore processor, as introduced in Section 3.4.1. The execution times were determined with the IBM PowerPC Multicore Instruction Set Simulator. For the prototype, the Proteus hypervisor was extended (see Chapter 3). As guest operating system, the RTOS ORCOS, developed at the University of Paderborn [Kerstan, 2011], was executed by the hypervisor. ORCOS needs no external libraries and requires 18 kB to 32 kB of memory (dependent on the configuration). It is available for PowerPC405, Sparc Leon3, and ARM and it provides scheduling according to round robin, EDF, and RM.

In the following, the execution times of the main routines of the adaptive scheduler are presented. The `init` function initializes the data structures (servers, ready queue) and performs already an initial bandwidth distribution. `schedule` implements the scheduling policy: it determines which VM to execute next, for how long, sets the programmable interval timer accordingly, and finally calls the function to resume the selected VM. `distribute` computes the additional bandwidth allocations based on the current resource requirements (see Algorithm 3 in Section 6.4.2).

Figure 6.16 depicts the execution times of these main routines subject to the number of virtual machines executed on the same core (two to six). The execution times are all in the range of about 1 to 8 microseconds. The most frequently called routine `schedule` is characterized by a low execution time below 2 microseconds for all numbers of VMs. The execution time of `distribute` is between 1.5 microseconds for two VMs and about 4 microseconds for six VMs. `init` requires the longest execution time, 2.5 microseconds for two VMs, more than 7 microseconds for six VMs. Included is however already a bandwidth distribution, but no scheduling decision. `init` has to be executed only once at system start, which is why its execution time is

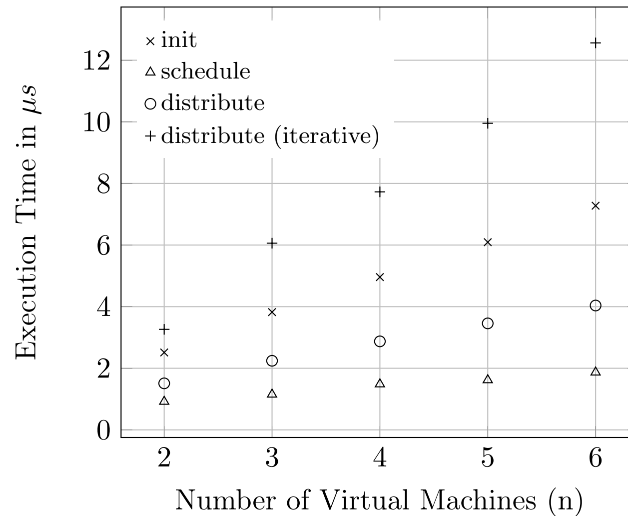


Figure 6.16: Execution times of scheduler routines subject to the number of virtual machines (PowerPC 405 @300 MHz)

less critical compared to the functions `distribute` and `schedule`, which are executed regularly at runtime.

For comparison purposes, we also show the execution times of an iterative version of the elastic bandwidth distribution, referred to as *distribute (iterative)*. As discussed in Section 6.4.1, state-of-the-art elastic bandwidth distribution algorithms perform the distribution in an iterative manner, whereas our algorithm requires only a single iteration through all VMs. A lower execution time for our algorithm can be observed for all numbers of VMs: the larger n, the larger the difference (up to 58% for six VMs). These figures corroborate the hypothesis that the new algorithm has a lower execution time overhead.

The execution time of all functions is dependent on the number n of VMs assigned to the same core. `init` and `distribute` have a computational complexity of $\mathcal{O}(n \cdot logn)$ (see Section 6.4.2). However, this is not observable for these small numbers of VMs, the impact of the sorting is apparently too low.

As a reminder (see Section 3.4.2), the hypercall `sched_yield` (voluntarily release the processor) has an execution time of about 0.5 microseconds, measured until the start of the hypervisor's `schedule` routine. By calling `sched_set_param`, the guest OS passes information to the hypervisor's scheduler, e.g., to inform about a mode change. The execution time of this hypercall is about 0.8 microseconds, with the measurement stopped when the calling VM resumes its execution. The worst-case execution times for a shared memory read and write are 2.2 microseconds and 1.8

Table 6.1: Thresholds for distribution of dynamic slack

Slack Threshold [μs]				
number of virtual machines n				
2	3	4	5	6
2.284	4.573	5.214	5.862	6.427

microseconds, respectively.

6.8.3 Overhead versus Benefit: Threshold for Slack Redistribution

Independent of whether structural or dynamic slack is handled, the overhead of bandwidth redistribution is the same. However, the redistribution of the two kinds of slack differs significantly in terms of probability of occurrence and length of period of validity. Dynamic slack is expected to arise much more frequently, but the bandwidth redistribution is only valid for a single period. Conversely, mode changes and as a consequence thereof the redistribution of structural slack are expected to happen possibly in a scale of seconds. The resulting distribution is valid for the entire interval between mode changes. Hence, the execution time costs of bandwidth redistribution are more critical for dynamic slack and actually crucial for the question whether a system can take advantage of dynamic slack.

Figure 6.17 shows the two different possibilities to react to a `sched_yield`, redistribution of the dynamic slack or no redistribution. Again, depicted is a simplified example of just one guest, without call of the function `schedule`. And it does in practice not make sense to resume the execution of a guest that just yielded. The overhead consists of the execution time of the function `distribute` by the hypervisor (which includes the write to shared memory and register) and the readout of the shared memory by the operating system. Both the call of the hypervisor function `schedule` and the check of the processor flag by the guest OS have to be performed regardless of whether the hypervisor redistributes or not.

Table 6.1 lists the thresholds for the redistribution of dynamic slack as a function of the number of VMs: if the amount of dynamic slack is greater than the threshold, the benefits of a redistribution exceed the costs. In case of two VMs, the dynamic slack can be passed directly to the other VM, without having to call the distribute function, resulting in a significantly lower threshold. For the redistribution among three to six VMs, the amount of dynamic slack has to be greater than 4.6 to 6.5 microseconds. These values are low enough to take effective advantage of dynamic slack in many practical circumstances.

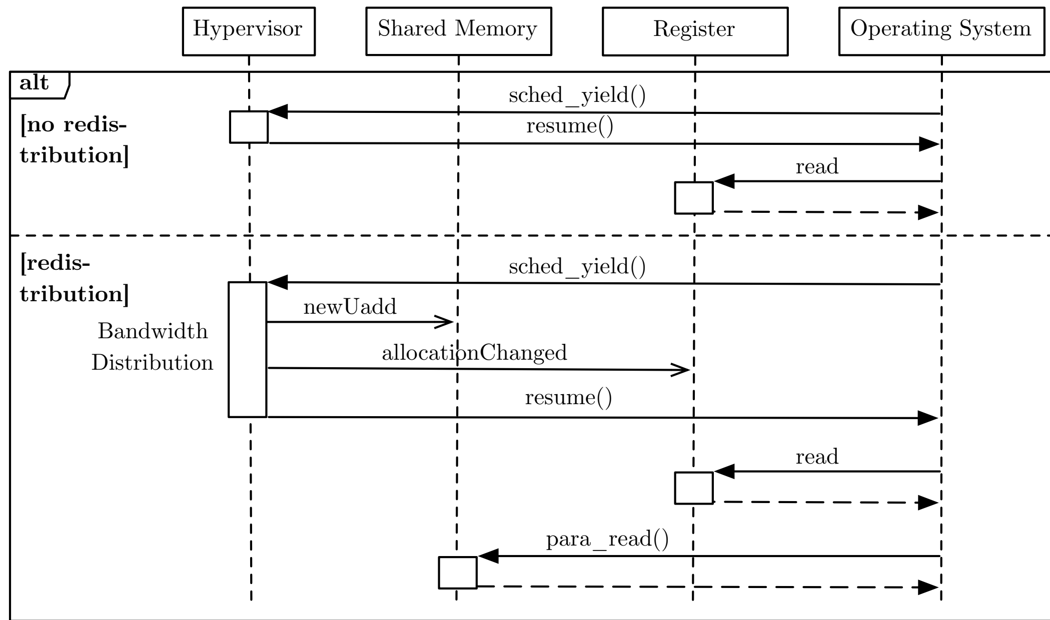


Figure 6.17: Interaction between hypervisor and operating system in the case of yield, with and without redistribution

6.8.4 Memory Footprint

As introduced in Chapter 3.4.3, Proteus can be configured statically depending on the requirements of the application. The module with the dynamic bandwidth redistribution functionality (scheduler and communication between hypervisor and OS) adds about 2.3 kB. If all features required for dynamic bandwidth management are enabled, which includes paravirtualization, the memory requirement of the hypervisor hosting two VMs sums up to about 14 kB (see Table 6.2). For each additional VM the memory requirement increases by 58 bytes.

Table 6.2: Memory footprint for scheduling functionality (2 virtual machines)

Feature	**Memory Footprint [bytes]**		
	text	data	total
Base Hypervisor	8224	2980	11204
Paravirtualization	252	148	400
Bandwidth Redistribution	2014	316	2330
Total	10490	3444	13934

6.8.5 Paravirtualization Effort

In order to paravirtualize an operating system for the presented adaptive scheduling, the scheduler has to be modified and a protocol-compliant communication with the hypervisor has to be added. The required communication between guest OS and hypervisor is realized by hypercalls, signaling via a register, and shared memory communication. The hypervisor informs the guest OS about bandwidth allocation changes via register and shared memory. Hypercalls are used by the guest OS to inform the hypervisor about a mode change or to voluntarily release the core and immediately pass control when it does not need the remaining assigned computation bandwidth in the current period and would otherwise idle (dynamic slack).

The communication functionality is provided by a library. Main modification is the addition of the protocol-compliant passing of scheduling information to the hypervisor. Instead of idling, the guest OS should yield. In case of a task mode change, the guest OS has to inform the hypervisor. In order to detect whether the hypervisor changed the bandwidth allocation, the control flow has to be adapted: after a context switch from hypervisor to OS, the OS has to check a processor flag and, if the allocation was modified, read out the shared memory. In case of ORCOS, the paravirtualization effort accounted for about 50 lines of C++ code.

6.8.6 Comparative Evaluation

There are four main approaches to provide temporal isolation between multiple VMs with real-time constraints, as introduced in Section 2.4.3:

1. dedicated processor core for each VM,
2. static precedence of a single real-time VM per core,
3. static cyclic schedule with fixed execution time slices,
4. execution-time servers.

In the following, these four solutions are compared qualitatively (see Table 6.3 for a summary). Subsequently, different server-based solutions are compared quantitatively by a scheduling simulation.

The first technique executes each VM on a dedicated core, both real-time and non-real-time VMs. By consequence, each guest system is executed at all times. The result is a required number of cores equal to the number of virtual machines and in most cases a low utilization of at least some cores. On the plus side, paravirtualization is not required: a VM that does not share the core must not know that it

is executed by a hypervisor. Moreover, the solution is highly adaptive as VMs can increase and decrease their bandwidth as desired, but adaptive measures between VMs are not possible. Finally, this is the only VM scheduling technique that does not require any information about the guest system, as long as it is known that it is schedulable on the core. The hypervisor does not schedule the VM and does therefore not need any information about task set or timing requirements. Use case for this solution is the realization of a partitioning of a multicore processor for as many guest systems as cores (provide single-core execution environments).

According to the second solution, a processor core can host only a single VM with real-time requirements. In contrast to the first solution, the real-time VM might share the core with non-real-time VMs, but definitely not with other real-time VMs. It is guaranteed that the temporal requirements of the real-time VM are met by executing it whenever it has a computation demand. Other VMs without real-time requirements can execute in background, not jeopardizing the response times of the real-time VM. Note, in the case of sharing the core, paravirtualization is required for the real-time VM, since is has to inform the hypervisor about its requirements. A safe solution would be the execution of non-real-time VMs only in time intervals that were explicitly released by the real-time VM, for example by a yield hypercall with idle interval length passed as parameter.

As it is the case for the first solution, this approach does not use the full potential of virtualization and its application is severely restricted. In many cases, it leads to both a low utilization of the processor cores and a high number of required cores (at least as many as there are real-time VMs). The real-time guest has to be paravirtualized, but without benefit for itself, just to allow the execution of other guests. Performance characteristics for the non-real-time VMs are hard to predict, even if detailed knowledge about the real-time guest's runtime behavior is available, and impossible to predict without.

The execution of the real-time VM whenever it has a computation demand is equivalent to highest adaptability. An adaptive scheduling of the non-real-time VMs is possible as well. Use cases for this scheduling solution are systems that include only a single real-time guest anyway or systems for which detailed information about the timing requirements is not available and cannot be derived, so that neither a design of a cyclic time slice schedule nor a dimensioning of a server is possible.

In the third solution, a static cyclic schedule [Baker and Shaw, 1989] is designed by analyzing the guests' task sets and assigning execution time windows within a repetitive cycle to the VMs based on the required utilization and execution frequency.

This static scheduling approach is for example part of the software specification ARINC 653 for avionics systems [Prisaznuk, 2008b]. It is well analyzable, highly predictable, does not require paravirtualization and can fully utilize each core, but lacks run-time flexibility. It is inadequate for applications with varying resource demand, since an adaptive measure is only possible by redesigning the schedule, which can seldom be done at runtime due to the high computation time overhead involved. If there is only a very limited number of combinations of active modes, it might be possible to construct at design time a specific schedule for each combination and switch between these combinations at runtime [Groesbrink et al., 2014b].

Our work is based on the fourth solution: the computation requirements of the VMs (for example available in terms of a demand bound function) are abstracted as execution time servers, which are scheduled by the hypervisor as periodic tasks. The hypervisor's scheduler enforces the server bandwidths. Real-time and non-real-time VMs can share a core and fully utilize it. Paravirtualization is not required for a static server-based scheduling. However, a periodic server with a fixed bandwidth [Sha et al., 1986] cannot react to mode changes and remains active when the associated guest system idles until its budget is exhausted. This chapter introduced an adaptive server-based scheduling, which requires paravirtualization.

In the following, the performance of four server-based approaches is compared, namely two approaches with fixed server bandwidths and two approaches with adaptive bandwidth distribution as proposed in this work. The comparison is carried out through simulation of synthetically generated workloads by the extended real-time scheduling simulator RTSIM. We used Brandenburg's toolkit SchedCAT [Brandenburg, 2014], which was already introduced in Section 5.4, to generate unbiased synthetic server sets. As the proposed approach is a partitioned multicore scheduling solution, we analyze the scheduling of a set of servers assigned to the same core.

Experiment I: Effectiveness of Adaptive Scheduling

In this first experiment, we investigate the effectiveness of the adaptive scheduling. 100,000 sets of VMs were generated according to the following parameter ranges:

- n uniformly distributed over $[2, 6]$
- $\bar{U}$ uniformly distributed over $[0.1, 0.2, ..., 0.9]$
- $U_{min}(V_i)$ distributed over $[0, \bar{U}]$ so that $\bar{U} = \sum_{i=1}^{n} U_{min}(V_i)$
- Π_i generated as harmonic within $[10\mu s, 1000\mu s]$ ($\Theta_i = U_{min}(V_i) \cdot \Pi_i$ follows)
- $U_{lax}(V_i)$ uniformly distributed over $[0, 0.20]$
- $p(V_i)$ uniformly distributed over $[0.1, 0.3]$

Characteristic / Technique	Number of Cores	Paravirtualization	Adaptiveness	Required Info about Guests
Dedicated Core	n	not required	high	none
Precedence	$\geq r$	required	high	utilization
Static Cyclic	$\lceil \sum_i U(V_i) \rceil$	not required	none	task set
Server-based	$\lceil \sum_i U(V_i) \rceil$	not required	high (demands paravirtualization)	demand bound function

Table 6.3: Qualitative comparison of virtual machine scheduling techniques (n: number of VMs, r: number of real-time VMs)

- $bdf(V_i)$ uniformly distributed over $[0.75, 1.00]$

A VM has three modes, differing regarding U_{lax}. For simplicity, the weights are based on $U_{lax}(V_i)$. $p(V_i)$ denotes the probability of a mode change at the beginning of each period of V_i. The bandwidth demand factor $bdf(V_i)$ represents a variable demand of the server budget within one specific server period, assuming a value in the interval $[bdf(V_i) * \Theta_i, \Theta_i]$. If $bdf(V_i) = 1$, V_i needs the worst-case demand in this server period. A smaller value results in idle time, which might be redistributed by the adaptive approach, but not by the fixed bandwidth management. Every configuration was simulated for $10s$ of simulated time.

To assess the effectiveness of our mechanism of distributing slack bandwidth according to the presented policy, we define the *relative error* δ of budget allocation, defined for the kth period of the server that executes V_i based on the assigned execution budget Θ_i^k and the actually desired budget Θ_i^{k*}:

$$\delta(V_i, k) = \frac{\Theta_i^k - \Theta_i^{k*}}{\Theta_i^{k*}} \tag{6.8}$$

The desired budget Θ_i^{k*} is defined by $U_{max}(V_i) = U_{min}(V_i) + U_{lax}(V_i)$ of the current mode and therefore not to be mistaken with the required budget $U_{min}(V_i)$ that guarantees schedulability. Including the bandwidth demand factor $bdf(V_i)$, the desired budget of V_i for the kth instance is a random value within the following interval:

$$\Theta_i^{k*} \in [bdf(V_i) \cdot U_{max}(V_i) \cdot \Pi_i,\ U_{max}(V_i) \cdot \Pi_i] \tag{6.9}$$

In case of a negative $\delta(V_i, k)$, the desired budget was not saturated. A positive $\delta(V_i, k)$ denotes idle time of V_i in the considered period and therefore unused budget. Finally, $\delta(V_i)$ is the average over all $|\delta(V_i, k)|$ for all periods k of VM $V_i \in V$. We keep track of the average values of each $\delta(V_i)$ and define δ as the average over all $\delta(V_i)$.

In a nutshell, the metric for this experiment is the relative error of allocated budget and desired budget. The desired budget changes constantly during runtime, based on both mode changes and a bandwidth demand that varies per instance, that is to say that the VMs might not need the worst-case demand in a specific instance. The smaller the relative error, the more effective the bandwidth allocation, since the relative error indicates either a non-saturated desired budget or an unused budget. The experiment with synthetically generated workloads investigates whether the adaptive approach is able to follow the varying computation bandwidth demands.

Expected Result: The adaptive algorithms follow the desired bandwidth more closely. The adaptive distribution of structural and dynamic slack leads to the small-

est relative error, followed by the adaptive redistribution of structural slack, followed by the non-adaptive distribution.

Table 6.4 lists the average values of δ and the 95% confidence intervals for four different server-based scheduling policies:

Only $\mathbf{U_{min}}$ allocates for each server period a static budget based on the minimum utilization of the associated guest.

Fixed allocates as well a static budget, but each VM receives in addition to U_{min} a share of U_{spare} based on the weights of the first mode (one-time offline distribution).

Adaptive (Structural Slack) denotes the adaptive distribution of structural slack only.

Adaptive (Structural + Dynamic Slack) denotes the adaptive distribution of both kinds of slack, structural and dynamic slack.

For any VM in all periods, the fixed bandwidth distribution based on U_{min} results in an average δ difference of 11.7% between desired budget and allocated budget. When performing an offline distribution of U_{spare} based on the first modes, the resulting average δ is 8.1%. When using the proposed adaptive bandwidth distribution, but redistributing structural slack only, the average δ falls significantly to 4.8%. This value falls even further to 1.1%, when using in addition the adaptive bandwidth distribution of dynamic slack. These lower values of the relative error δ confirm that the actual distribution of bandwidth follows closely the desired bandwidths, showing the effectiveness of the proposed adaptive scheduling in enforcing an elastic distribution. The adaptive measures reduce the relative error of bandwidth distribution with statistical significance.

Experiment II: Effect of Mode Change Probability

In this experiment, we investigate the effect of the mode change probability. For each value for the mode change probability $p \in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$ 500 sets of VMs were generated according to the following parameter ranges:

- n uniformly distributed over $[2, 6]$
- $\bar{U}$ uniformly distributed over $[0.1, 0.2, ..., 0.9]$
- $U_{min}(V_i)$ distributed over $[0, \bar{U}]$ so that $\bar{U} = \sum_{i=1}^{n} U_{min}(V_i)$

Table 6.4: Relative error δ of allocated bandwidth and desired bandwidth for fixed distribution and adaptive distribution

Policy	Arithmetic Mean [%]	95% Confidence Interval [%]
Only U_{min}	11.744	[11.710; 11,777]
Fixed	8.121	[8.083; 8.159]
Adaptive (Structural Slack)	4.828	[4.807; 4.850]
Adaptive (Structural + Dynamic Slack)	1.056	[1.046; 1.067]

- Π_i generated as harmonic within $[10\mu s, 1000\mu s]$ ($\Theta_i = U_{min}(V_i) \cdot \Pi_i$ follows)
- $U_{lax}(V_i)$ uniformly distributed over $[0, 0.20]$
- $bdf(V_i)$ uniformly distributed over $[0.75, 1.00]$

Expected Result: Compared to the non-adaptive algorithm **Fixed**, the algorithm **Adaptive (Structural Slack)** leads to a lower relative error. The exception is $p = 0$, for which both algorithms produce the same result.

Figure 6.18 depicts the resulting relative errors for the policies **Fixed** and **Adaptive (Structural Slack)**. As expected, they perform equally well for a mode change probability of 0 (exact same value), since there are no mode changes that trigger a redistribution. For each mode change probability $p > 0$ the adaptive distribution performs significantly better. Observable as well, **Adaptive (Structural Slack)** maintains a low relative error even in the case of high mode change probabilities.

Experiment III: Effect of Bandwidth Demand Factor

In this experiment, we investigate the effect of the bandwidth demand factor. For each bdf value $\in \{0, 0.2, 0.4, 0.6, 0.8, 1\}$ 500 sets of VMs were generated according to the following parameter ranges:

- n uniformly distributed over $[2, 6]$
- $\bar{U}$ uniformly distributed over $[0.1, 0.2, ..., 0.9]$
- $U_{min}(V_i)$ distributed over $[0, \bar{U}]$ so that $\bar{U} = \sum_{i=1}^{n} U_{min}(V_i)$
- Π_i generated as harmonic within $[10\mu s, 1000\mu s]$ ($\Theta_i = U_{min}(V_i) \cdot \Pi_i$ follows)
- $U_{lax}(V_i)$ uniformly distributed over $[0, 0.20]$
- $p(V_i)$ uniformly distributed over $[0.1, 0.3]$

Expected Result: Compared to the algorithm **Adaptive (Structural Slack)**, which redistributes only in case of a mode change, the algorithm **Adaptive (Struc-**

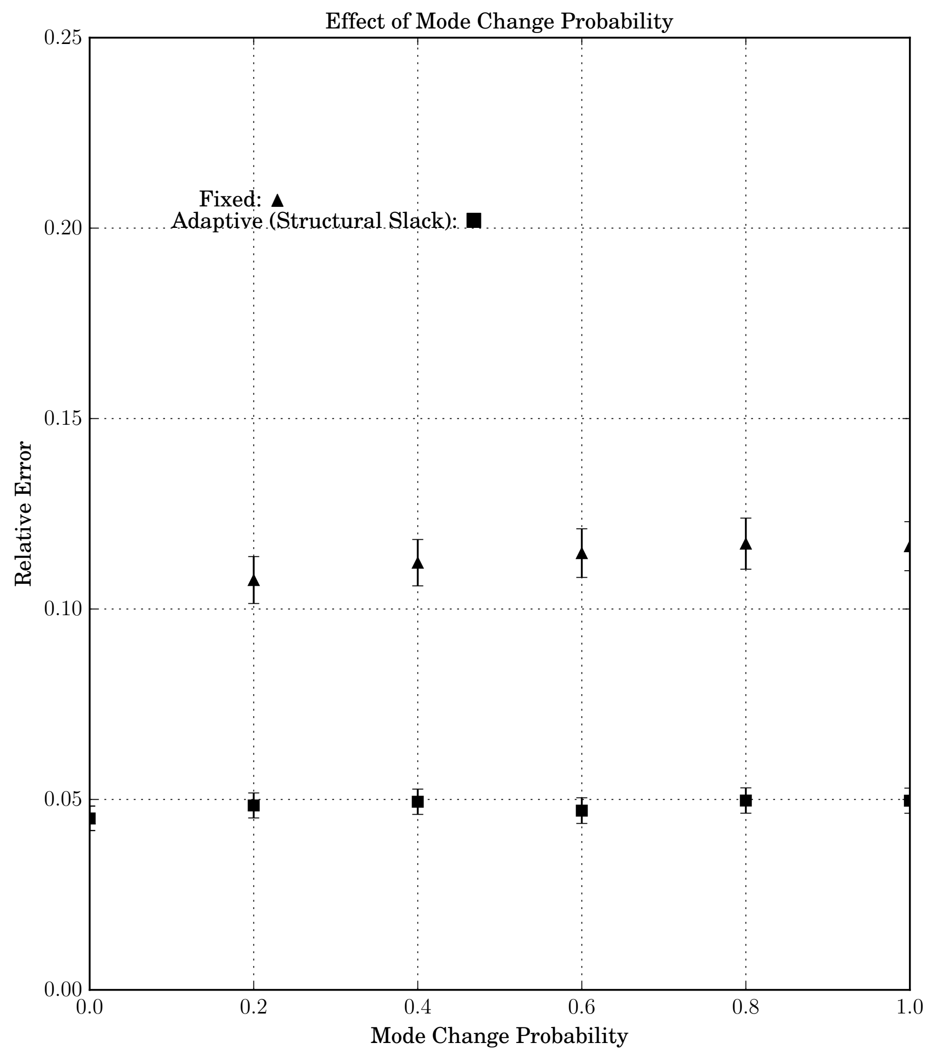


Figure 6.18: Effect of mode change probability

tural + Dynamic Slack) leads to a lower relative error. The lower the bandwidth demand factor, i.e., the larger the computation time variations, the larger the difference regarding relative error.

Figure 6.19 depicts the resulting relative errors for the policies **Adaptive (Structural Slack)** and **Adaptive (Structural + Dynamic Slack)**. As expected, the policy **Adaptive (Structural + Dynamic Slack)** leads to a lower relative error for all bandwidth demand factors. The difference becomes smaller with larger bandwidth demand factors, as the computation time variations become smaller. The redistribution of dynamic slack is less useful if the amount of dynamic slack becomes smaller.

6.9 Summary

Existing hypervisor-based virtualization solutions for embedded real-time systems apply static resource management policies. An adaptive virtual machine scheduling is of great and so far untapped potential, especially for systems that operate in highly dynamic environments. However, it is most important that the scheduling policy guarantees that all VMs meet their real-time requirements and that temporal isolation between guest systems is maintained, since this is crucial for the ability to integrate real-time systems, especially safety-critical ones.

This chapter proposed a virtual machine scheduling that combines adaptability and temporal isolation, defined not as an uninfluenced behavior, but as the guarantee that all guests are able to meet their timing constraints. Each guest system receives a guaranteed share of the processor time. Periodic execution time servers and the elastic task model combine analyzability at design time with adaptability at runtime. The correct execution of a virtual machine depends only on the server parameters and not on the behavior of other virtual machines, and is thus protected from potential overloads within another virtual machine.

The possibility to replenish the server budgets dynamically is exploited in an efficient way to implement the adaptive bandwidth measures, taking advantage of the slack generated by mode changes and shorter execution times than the reserved worst-case. The bandwidth distribution is carried out with fine-grained control according to the elastic model, supporting selected applications that can take advantage of higher computational bandwidth. A bandwidth redistribution is performed as well when an unforeseen worst-case execution time overrun of a critical guest results in an overload of the entire system. In this situation, it is attempted to protect the

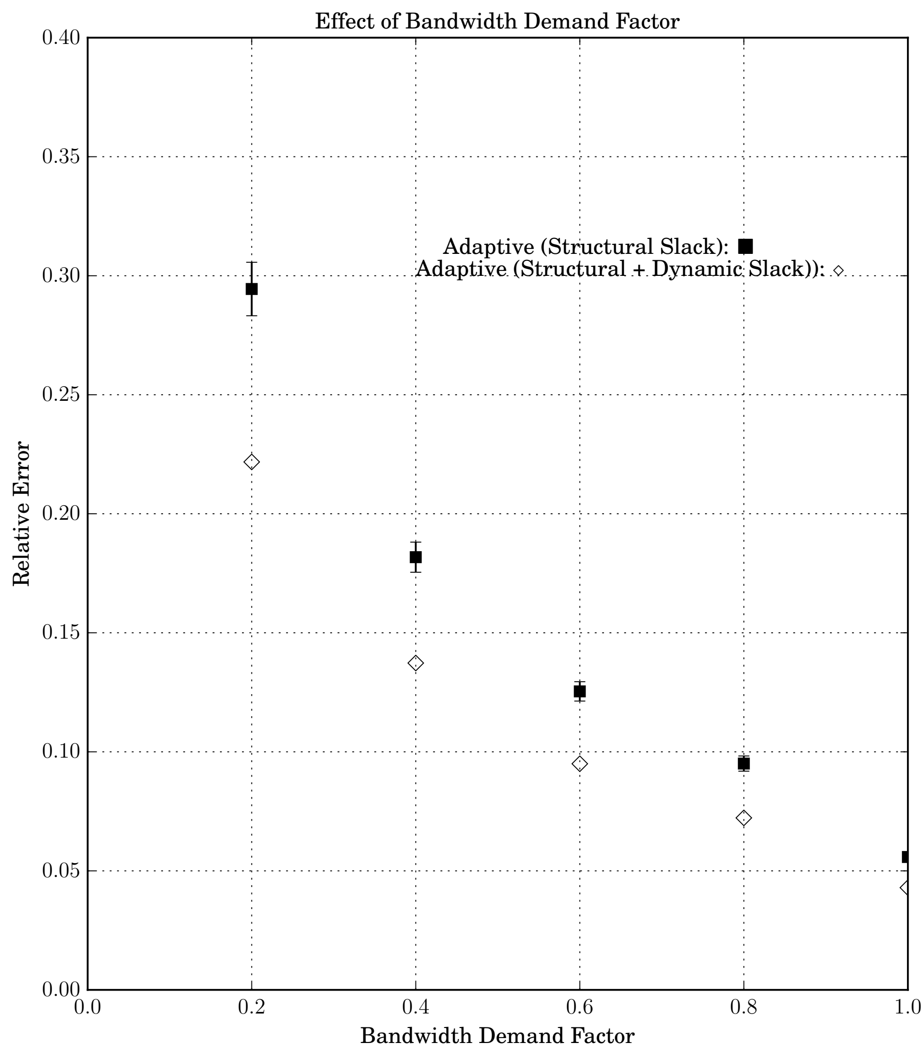


Figure 6.19: Effect of bandwidth demand factor

execution of the critical guest by stealing computation time from non-critical guests.

An analysis proved that the scheduling architecture guarantees a minimum bandwidth service. It is safe in steady state and during the transition phase from a bandwidth allocation to an updated one. This service guarantee is the basis for the architecture's real-time capability and a potential certification of functional safety.

A prototype was integrated into hypervisor and real-time operating system. The evaluation showed a reasonable paravirtualization effort, a low memory footprint, and a low execution time overhead that enables to make efficient use of slack. Different virtual machine scheduling approaches were compared qualitatively. Moreover, the proposed adaptive scheduling was compared to a static server-based scheduling. A scheduling simulation with synthetically generated workloads showed that the adaptive solution follows the varying bandwidth demand effectively.

Chapter 7

Real-Time Virtual Machine Migration

Contents

7.1 Problem Statement 163
7.2 Related Work 164
7.3 Design 167
7.3.1 Migration Policy 167
7.3.2 Integration into the Hypervisor 168
7.3.3 Protocol 169
7.3.4 Migration Test 171
7.3.5 Integration into Real-Time Virtual Machine Scheduling . . 173
7.4 Evaluation 175
7.4.1 Experimental Setup 175
7.4.2 Memory Footprint & Paravirtualization Effort 176
7.4.3 Execution Times & Downtime 177
7.4.4 Reliability Analysis 178
7.4.5 Case Study: Autonomous Rail Vehicle 182
7.5 Summary 186

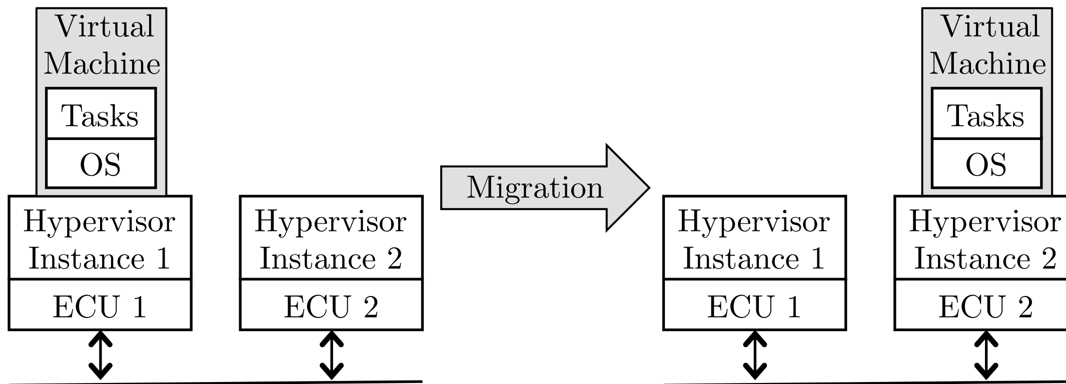


Figure 7.1: Virtual machine migration

Migration refers to the relocation of virtual machines (VMs) from one physical machine (PM) to another one at runtime [Smith and Nair, 2005c], as depicted in Figure 7.1. The execution is suspended on the source PM, the image of the VM (containing all data that is required to execute it) is transmitted to the target PM, where the execution is resumed at exactly the same point. Prerequisite are a computer network that allows the PMs to exchange data and the execution of an instance of the hypervisor on both source and target PM. It is assumed that the hardware configurations of source and target PM are identical.

Virtualization's architectural abstraction and encapsulation of guest systems provide flexibility and facilitate migration significantly. The hypervisor is fully aware of the resource usage and includes already functionality for saving and restoring the state of a guest, since this has to be done on each virtual machine context switch in order to suspend the execution of a VM and resume the execution of another one.

Benefits of migration are an increased reliability if applied in order to continue the functioning of a guest despite a hardware failure or a balanced load, especially for adaptive systems and systems that allow the addition of software at runtime. Virtualization became popular for embedded systems for the benefit of a better resource utilization, but fault tolerance is a new perspective, especially since consolidation creates a single point of failure. Applying VM migration to real-time systems demands a predictable timing behavior, especially regarding the downtime, during which the guest system is not executed. The following sections will be devoted to a VM migration approach for real-time systems.

7.1 Problem Statement

In the context of this work, virtual machine migration is examined as a fault tolerance technique. To improve the reliability, defined as the ability of a system to perform its required functions under stated conditions for a specified period of time [IEEE, 1990], i.e., continuity for correct service, migration is performed as a service restoration in response to hardware faults that prevent the guest system to comply with the functional and/or real-time specification. These faults are external to hypervisor and VM. In case of partially failed PMs, if the hardware failure still allows for a saving and transfer of the state of the VM, its operation can be continued on another PM. Examples for such hardware failures are partial memory failures, failures of coprocessors, hardware accelerators, or the graphics processing unit, as well as failures of I/O devices. Migration is not possible anymore in case of a breakdown of power supply, central processing unit, or the I/O connection to the potential target PMs. However, one can benefit from self-diagnosing hardware that signals upcoming failures on the basis of built-in self-tests (proactive fault tolerance), as often found in safety-critical embedded systems. The reliability regarding memory failures could be increased by checkpointing techniques, i.e., regularly saving VM states in a secondary storage, and then, in case of a memory failure, restoring from it (see for example [Kwak et al., 2001], [Punnekkat et al., 2001]).

Target architecture of this work are homogeneous multiprocessor platforms and distributed systems of multiple identical processors, each operating on an own random access memory, but connected via a network. Each unit of processor and memory is called physical machine (PM). The adaptive scheduling approach that was introduced in the last chapter addresses the resource management on a multicore platform, that is to say on a single PM. Migration is a dynamic resource management technique on the granularity of multiple PMs.

Each PM has to execute an instance of the same hypervisor. This is mandatory even in case of a fully virtualized guest system, since the migration protocol is based on communication among the involved hypervisors (see Section 7.3.3). Moreover, the transferred VM image is not executable on bare hardware. The hypervisor has to initialize the memory as well as the state of processor and used devices in order to resume at exactly the instruction at which the guest system was suspended on the source PM.

Highly dynamic virtualization solutions including VM migration are state of the art for the server market, but cannot be applied to embedded systems for lack of real-time response time guarantees. Most existing virtualization solutions for em-

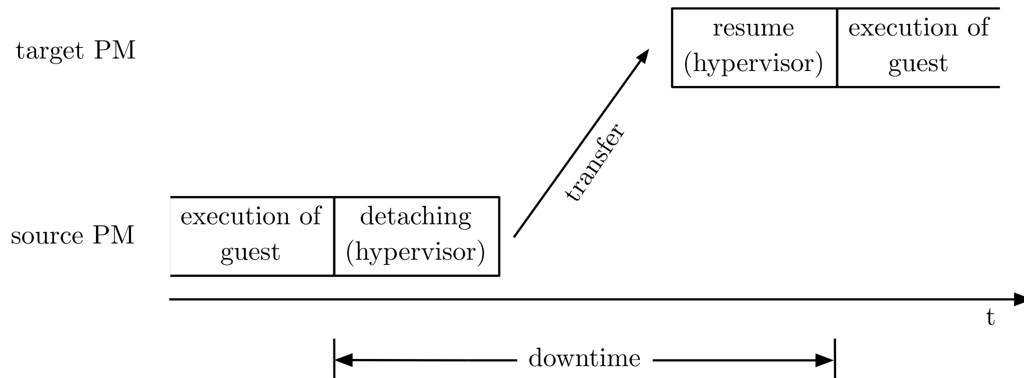


Figure 7.2: Downtime due to virtual machine migration

bedded systems do not exploit virtualization's flexibility and assign VMs statically to PMs [Gu and Zhao, 2012], due to the challenge of applying migration to real-time systems. The migration process induces a time interval in which the guest system is inactive. This *downtime* is composed of the time required by the hypervisor to prepare the migration process incl. the finding of a target, the transfer of the VM image to the target, and the time required by the target's hypervisor to restore the VM state and resume execution (see Figure 7.2). For an embedded systems with real-time constraints, this outage duration has to be predictable and should be minimized.

This work presents a migration approach for embedded real-time systems. The migration approach (Section 7.3.1) is aware of real-time requirements and addresses the real-time issues (1) service outage and (2) integration into the scheduling on the target PM. The focus is on the integration of migration functionality into a software stack and the co-design of hypervisor and paravirtualized guest OS. The evaluation investigates the overhead regarding execution time, memory footprint, and paravirtualization effort (Section 7.4). A reliability analysis is performed to motivate the work by quantifying the benefit (Section 7.4.4).

7.2 Related Work

There exist numerous works in the related field of process migration and basic results are transferrable to VM migration, but with a significant difference regarding overhead due to the larger amount of data to transfer. See [Milojíčić et al., 2000] for a survey on process migration. The *DEMOS/MP* operating system provided process migration with a location independent message-based communication. The entire

virtual memory is transferred in a copy-on-reference manner [Powell and Miller, 1983]. The *Sprite* operating system for networked workstations and file servers applied process migration to use idle machines, transparent to the user [Douglis and Ousterhout, 1987]. Stankovic and Ramamritham presented the *Spring Kernel*, a real-time operating system that provides process migration in order to balance load and adapt to varying environment conditions [Stankovic and Ramamritham, 1989].

Regarding VM migration for non-real-time systems, Clark et al. described the basics of VM migration and specified the data that has to be transferred to the target node [Clark, 2005]. Hansen and Jul introduced self-migration: the guest OS is aware of being executed in a virtualized manner, just as a paravirtualized OS is, and uses checkpointing to transfer its state by itself [Hansen and Jul, 2004]. In contrast, Nelson et al. presented migration that is transparent to a non-real-time OS [Nelson et al., 2005]. Kozuch and Satyanarayanan proposed *Internet Suspend/Resume*, a virtual machine based technique for the migration of the state of a user environment from one machine to another over the internet [Kozuch and Satyanarayanan, 2002]. The state is stored in a distributed file system, which is accessible by the target machine. It is loaded incrementally in modules and execution is resumed as soon as the necessary modules are present, before the entire state is transferred. Sapuntzakis et al. proposed a similar solution, however, with a direct transfer from source to target machine instead of using a distributed file system [Sapuntzakis et al., 2002]. Both works are implemented on the *VMWare GSX Server*, a hosted hypervisor (type 2) that runs either on Linux or Microsoft Windows. VMWare's *VMotion* performes VM migration in a local-area network in order to balance load (for response time reduction and power management), quarantine an attacked VM, consolidate communicating systems, and for fault tolerance and maintenance reasons [Smith and Nair, 2005c]. Prerequisite is an operation of source and target on shared disks in a storage-area network. Aalto presented with *DynOS SPUMONE* runtime migration of guest OSs to different cores on a multicore processor [Aalto, 2010]. Mitake et al. introduced with *vlk* a virtualization solution with an assignment of virtual CPUs to guest OSs [Li et al., 2012a]. It includes a dynamic mapping of virtual CPUs to physical CPUs. Since their target architecture has a global shared memory, only the register content has to be transferred.

In the real-time world, Checconi et al. addressed live migration of VMs with soft real-time constraints [Checconi et al., 2009]. The execution of the VM is not interrupted for the entire duration of the transfer. This is achieved by a pre-copying of memory pages, a technique introduced by Theimer et al. in order to reduce the

downtime [Theimer et al., 1985]. Memory pages that are modified by the source after the transfer have to be retransmitted and the authors present a stochastic model for the probability of a page to become dirty again and derive the expected migration time. (Similarly, post-copy refers to the resume of the VM before the entire memory was transferred [Hines and Gopalan, 2009]. Pages that are not required to resume are transferred in a copy-on-reference manner.) The implementation is based on Kernel-based Virtual Machine (KVM), a Linux kernel virtualization infrastructure. Regarding real-time VM scheduling, they refer to existing server-based scheduling solutions.

Baliga and Kumar presented *Etherware*, a middleware for wireless networked control systems [Baliga and Kumar, 2005]. The application software is composed of components. Etherware provides the abstraction that all components execute on a single computer, in fact, components can be dynamically migrated in order to balance communication and computational loads. The communication information at each machine or component that communicates with the migrating component is updated. The authors do not discuss real-time capabilities.

Rasche and Polze derived a calculation for the blackout time due to a reconfiguration (such as component migration) of real-time software [Rasche and Polze, 2005]. They propose to reserve processor resources for potential reconfiguration commands. Kalogeraki et al. introduced a resource management system for distributed soft real-time systems [Kalogeraki et al., 2008]. It operates on top of the CORBA middleware and distributes the application objects in order to maintain a uniform load on the different nodes.

In order to quantify the benefits for reliability, Kim et al. developed an availability model of a virtualized system for data centers, which incorporates both hardware and software failures as well as VM migration [Kim et al., 2009]. The results showed an increased steady state availability by applying virtualization. Melo et al. evaluated migration as a rejuvenation technique for cloud computing environments and could reduce the system downtime [Melo et al., 2013]. Ramasamy and Schunter quantified the impact of virtualization on node reliability [Ramasamy and Schunter, 2007]. Roy et al. addressed infrastructure failure and utilization issues by a VM migration approach that is aware of violations of service level agreement thresholds [Roy et al., 2013]. Nagarajan et al. studied migration triggered by node health monitoring (proactive fault tolerance) [Nagarajan et al., 2007].

7.3 Design

7.3.1 Migration Policy

The migration policy is introduced in the following by answering the relevant questions.

Which events trigger a migration?
In the context of this work, migration is only performed if the functioning of a VM can continue despite of a hardware failure, and not for load balancing. Migration involves a significant overhead and the transmission of a VM's memory causes a significant downtime, as analyzed in detail in Section 7.4.3. It implies changes for the hierarchical scheduling on both sides, especially the integration into the virtual machine scheduling at the target node (see Section 7.3.5). For these reasons, migration-based load balancing is for embedded real-time systems less attractive. Migration as a load balancing measure would potentially complicate certification, whereas, if applied in order to continue the functioning of a system in case of a hardware failure, VM migration actually increases the reliability (see Section 7.4.4) and might therefore support certification.

What has to be transferred?
The *image* of a VM includes all data that is required to execute the guest and includes:

- code, stack, and data segments belonging to operating system and application tasks,
- VM context (register values, condition codes, stack pointer, program counter etc.),
- data structures of the hypervisor associated with the VM (identifier, VM scheduling parameters etc.).

Which VM and which target is selected?
The approach migrates all VMs that have or will have a service outage due to the hardware failure. In case of different criticality levels, this is done in order of decreasing criticality. If data transfer rates between PMs differ, the PM with the highest rate is selected as migration target. As a tie-breaker in case of equal rates, the PM with the lowest CPU utilization is selected. The hypervisor maintains a list of potential target PMs, possibly VM-specific, if for example some guests require a co-processor that is not available on all PMs.

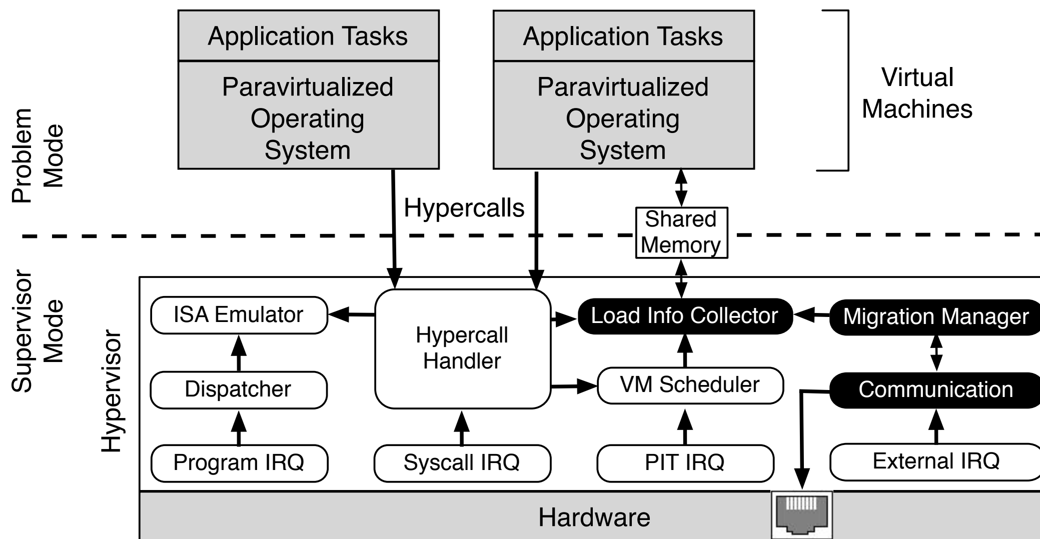


Figure 7.3: Integration of the migration functionality into the architecture of the hypervisor

Who takes the decisions?
The migration policy is realized in a distributed manner without central instance. Each hypervisor instance can initiate a migration. When a hypervisor instance starts the evaluation of a potential migration, all other hypervisor instances cannot do the same until this migration is finished or canceled (at the latest by a timeout).

When is the migration process initiated?
Triggered by a certain event, the hypervisor decides on whether to migrate a VM, but the VM itself determines the starting time.

7.3.2 Integration into the Hypervisor

The integration of the migration functionality into the hypervisor *Proteus* is depicted in Figure 7.3. The impact of the migration components is limited as much as possible. The behavior of the other components of the hypervisor is not affected until an event triggers the evaluation of a potential migration. The following components realize the migration functionality.

The *Load Information Collector* gathers data about the resource utilization at runtime. The hypervisor assigns the resources to the VMs and therefore has knowledge about the guests' memory and I/O usage. However, the hypervisor does not have any insight in the scheduling and the upcoming deadlines of the guests. Without this knowledge, it cannot evaluate whether a certain downtime invokes a deadline

miss or not. Consequently, an explicit communication between guest OS and hypervisor is required. When the migration manager considers a migration, the Load Information Collector invokes the OS to pass information about its task scheduling. The *Communication* module provides the functionality to communicate with hypervisor instances on other PMs. It is the Migration Manager's interface for the message-based communication, transparent of the underlying network technology. The *Migration Manager* organizes the process by implementing the policy as introduced in the previous section. It decides *which* virtual machine shall be migrated *where* and *when*, based on the data obtained from the Load Information Collector and other hypervisor instances.

The required communication between guest OS and hypervisor is realized by both hypercalls and shared memory communication. For the latter, a memory region within a VM's memory space is dedicated to paravirtualization communication. It is accessible by hypervisor and corresponding VM, however not by any other VM. Hypercalls are a technique for the guest OS to invoke communication. A hypercall leads to a preemption of the guest system; the hypervisor takes control and handles the hypercall. The hypervisor cannot call functions of the OS, but it writes code words to the shared memory. In the process of each context switch from hypervisor to guest, the OS reads out the shared memory and if necessary subsequently provides the hypervisor with the requested information. To pass control back to the hypervisor immediately, it executes the hypercall `yield`. With the hypercall `startMigration`, the OS signals the hypervisor that it can be suspended in order to start the data transfer.

7.3.3 Protocol

Figure 7.4 depicts the migration process as a UML sequence diagram. For clarity, the hypervisor is visualized as a single component and only the successful case is shown, without error or timeout handling etc. To start the migration process of the operating system *Guest*, the Migration Manager of the source broadcasts a message with the intent to migrate a VM to all other hypervisor instances (incl. information about the required resources). The Migration Managers of the receiving hypervisor instances perform an *acceptance test* that checks whether their remaining resources are sufficient to host the VM (explained in next section) and reply with the result, incl. information about their resource utilization. The source analyzes the replies, selects a target among the positive ones and informs the other hypervisors.

The source performs a *downtime test*, which checks whether the transfer of the

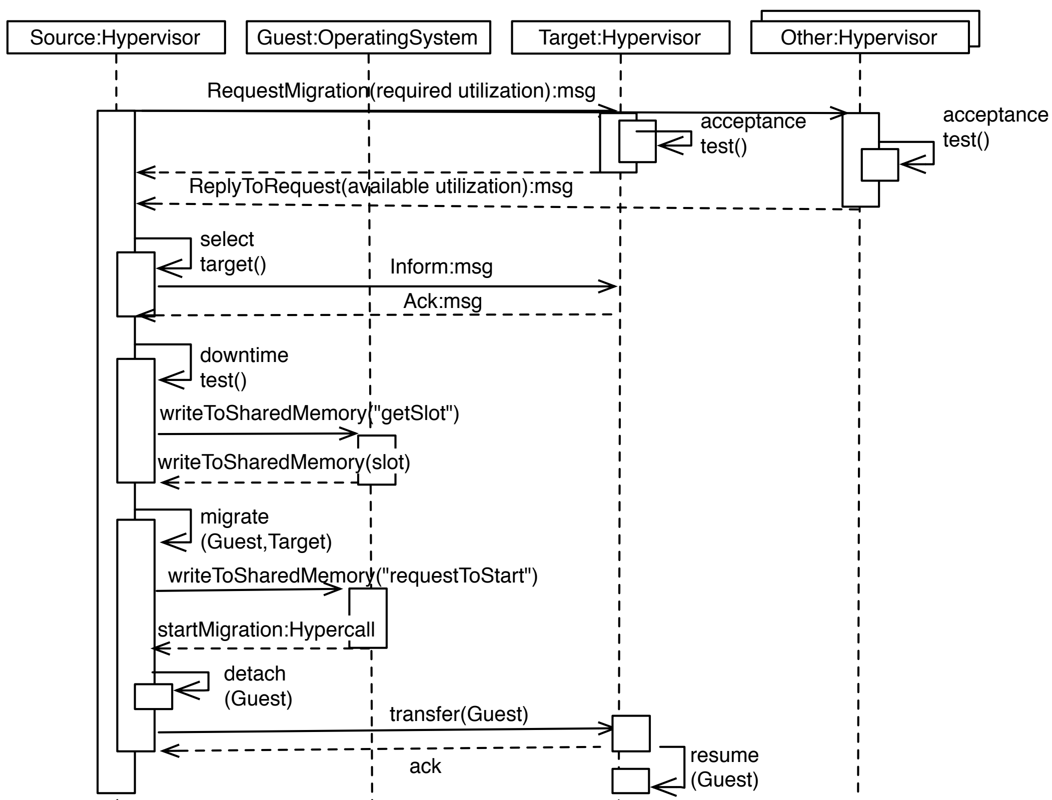


Figure 7.4: Migration protocol

guest to the target implies deadline misses (explained in next section). For this test, the hypervisor asks the OS for the maximum possible downtime (termed slot). The hypervisor informs the OS about the result of the downtime test and the OS might switch to a different mode for the case of predicted deadline misses. It resumes the guest OS and asks it to signal the starting time for the data transfer, since only the guest OS knows when the maximum downtime is available.

Once having received the signal, the hypervisor detaches the guest and transfers the VM image to the target PM, where the image is unwrapped, the memory is copied, and the processor state is reset. The execution is resumed at exactly the instruction at which it was suspended. The migration does not leave residual dependencies: the source hypervisor does not (and does not have to) continue to provide data or services for the VM after the migration. Residual dependencies are undesirable since they impact the performance of both the source (need to maintain data structures and provide functionality) and the target (overhead of continuing communication) [Douglis and Ousterhout, 2006]. In addition, they decrease reliability, since a correct functioning of the communication is mandatory and a failure on one PM affects VMs on other PMs.

7.3.4 Migration Test

There are two classes of deadline misses: the VM to be migrated could miss a deadline or another VM on the target PM could miss a deadline because the VM to be migrated is added and receives resource shares. We can derive two necessary conditions for a migration that does not provoke deadlines misses:

1. The resource utilization (CPU, memory, I/O) of the target PM must permit the addition of the VM (checked by the potential target through the *acceptance test*). This includes schedulability of both the VM to be migrated and the VMs that are already executed on the target PM.

2. The downtime imposed by the migration process must be short enough to exclude a deadline miss of the VM to be migrated (checked by the source through the *downtime test*).

A failed acceptance test leads to a cancellation, whereas a failed downtime test does not, since the alternative is a continued and not a temporal service outage.

Acceptance Test

The acceptance test checks whether the remaining resources of a potential target PM are sufficient to fulfill the requirements of the arriving VM. Regarding memory, a simple comparison of unassigned memory and demanded memory is needed, since the hypervisor assigns static adjacent chunks of memory to the VMs (fragmentation of memory not considered due to the realistic low numbers of both VMs and migrations). The I/O acceptance test is highly device specific and out of the scope of this thesis. The test regarding the resource computation bandwidth is introduced in detail in Section 7.3.5.

Downtime Test

The VM downtime must be predictable for a real-time system. Knowing the communication costs c_{com} as a function of the VM image size, the source hypervisor can estimate the downtime t_{down}:

$$t_{down}(V_i) = t_{detach} + c_{com}(size(V_i)) + t_{resume} + t_{service} \tag{7.1}$$

t_{detach} is the time required for the hypervisor to create the image for the transfer. t_{resume} refers to the time to unwrap the image at the target and set up the data structures, so that the VM becomes executable. Due to the integration into the VM schedule, the VM might have to wait before its execution starts, denoted by $t_{service}$ and derived in the following section.

The hypervisor asks the OS for its maximum affordable interval of inactivity $t_{down}^{max}(V_i)$ (largest possible service delay) and the downtime test is passed if:

$$t_{down}(V_i) \leq t_{down}^{max}(V_i)$$

The related value Δ_i^{max} is already required for the correct dimensioning of the associated periodic resource (see Section 5.3.1). The OS (or system designer) has to calculate the maximum possible downtime for the specific task set and the applied task scheduler. This calculation is directly associated with the calculation of slack, also called laxity, defined for a point in time t as the maximum time a job of a real-time task can be delayed without causing its deadline to be missed. For example, at the activation of a job, the slack is given as $X_j = d_j - a_j - C_j$. A slack function $A(t, u)$ returns the maximum amount of computation time that is available in the interval $[t, u]$ without leading to deadline misses of periodic tasks. [Buttazzo, 2004]

The maximum of the slack function over all intervals within the hyperperiod defines the maximum possible downtime of the VM ($lcm(V_i)$ denotes the least common

multiple of all task periods, i.e., the length of the hyperperiod):

$$t_{down}^{max}(V_i) = max_{t \in [0, lcm(V_i)], u \in [t+1, lcm(V_i)]} A(t, u) \ .$$

It is possible that in a specific runtime situation (based on the mode that is active at this time), the maximum affordable interval of inactivity is larger than Δ_i^{max}, since Δ_i^{max} is mode-independent (and has to be for the dimensioning of the periodic resource). If the OS is able to compute this larger value at runtime (or offline and store it), it can be used. See [Liu, 2000] for detailed information on slack computation. It should not be unmentioned that offline slack computations might not be applicable in case of release-time jitters.

7.3.5 Integration into Real-Time Virtual Machine Scheduling

As introduced in Section 2.4, the execution of VMs with real-time requirements demands an appropriate hierarchical real-time scheduling if the number of VMs exceeds the number of processor cores. In the previous Chapter 6, an adaptive scheduling was introduced, which implements virtual processors as scheduling servers $\Gamma_i(\Pi_i, \Theta_i)$ characterized by period and execution time budget. The assignment of a dedicated server to each VM guarantees a minimum but bounded computation time share for each VM in a specified time span.

In case of a multi-core platform, VMs are statically assigned to cores. On each core, the hypervisor schedules the assigned servers by static priorities according to the Rate Monotonic (RM) policy: the higher the request rate (the smaller Π), the higher the priority. The schedulability of the VMs is guaranteed if the sum of the VMs' resource requirements is smaller than or equal to the least upper bound of the processor utilization U_{lub}. In order to fully utilize the core ($U_{lub} = 1$), the partitioning algorithm introduced in Chapter 5 transformed the server periods to harmonic periods, i.e., the period of each server is an exact multiple of the periods of every other server with a shorter period. This harmonization is done for all servers that at design time are assigned to the same core, not for the system's entire set of virtual machines. For this reason, the period of the server of the migrated VM (in the following referred to as $V_{arriving}$) is not necessarily harmonic to the periods of the VMs on the target PM. The harmonic relationship might be lost and, by consequence, schedulability is no longer guaranteed. In order to maintain the harmonic relationship, the arriving VM has to be integrated. Algorithm 1 as introduced in Chapter 5 could be applied, but one had to rerun it for all VMs, since its correctness depends on considering the VMs in order of increasing Π^{opt}. The

arriving VM can be integrated without touching the dimensioning of the already hosted servers by the following equations:

$$\Pi_{V_{arriving}} = max(\{\Pi_i | V_i \text{ executed on considered core and } \Pi_i \leq \Pi^{opt}_{V_{arriving}}\})$$

$$\Theta_{V_{arriving}} = \Pi_{V_{arriving}} \cdot PCB_{V_{arriving}}(\Pi_{V_{arriving}}, A)$$

The period is set to the largest already existing period among those periods smaller than the maximum possible period $\Pi^{opt}_{V_{arriving}}$ (derived from the reactivity requirements). Consequently, this value and the optimal periodic capacity bound $PCB_{V_{arriving}}(\Pi, A)$ have to be included in the transmission from source to target hypervisor.

Schedulability on the target PM has to be guaranteed by the acceptance test. Regarding the resource computation time, this test is based on Equation 6.1 (see Chapter 6.3.2):

$$U_{min}(V_{arriving}) + \sum_{i=1}^{n} U_{min}(V_i) \leq U_{lub} = 1$$

It is still possible to fully utilize the processor ($U_{lub} = 1$), since we maintained the harmonic relationship between the servers' periods. This schedulability condition considers only the minimum bandwidths U_{min}, but the distribution of the spare bandwidth U_{spare} might have resulted in a higher bandwidth allocation among the already hosted VMs than just the sum of the minimum bandwidths. For this reason, we cannot instantly allocate $U_{min}(V_{arriving})$, but first, have to perform a bandwidth redistribution for this new VM set of all VMs that were already executed on the target core plus the arriving VM. The hypervisor starts to provide bandwidth (at least equal to $U_{min}(V_{arriving})$, potentially plus a share of the spare bandwidth) to the arriving VM at the end of the last finishing instance of all currently executed VMs, depicted as point in time t_1 in Figure 7.5.

At this point in time, it is not required to allocate only the minimum bandwidths to the VMs and U_{spare} is recalculated and distributed. For this reason, Θ'_1 and Θ'_2 in Figure 7.5 might not be based only on the minimum bandwidths, but as well on the just calculated values for U_{add}. A distribution of the spare bandwidth is safe at this point in time, since all VMs incl. $V_{arriving}$ receive their minimum bandwidth first and the remaining U_{spare} is distributed subsequently.

The arriving VM has to wait in the worst case for $\Pi_{V_{arriving}} - \Theta_{V_{arriving}}$ before receiving computation time:

$$t_{service} = t_1 + \Pi_{V_{arriving}} - \Theta_{V_{arriving}}$$

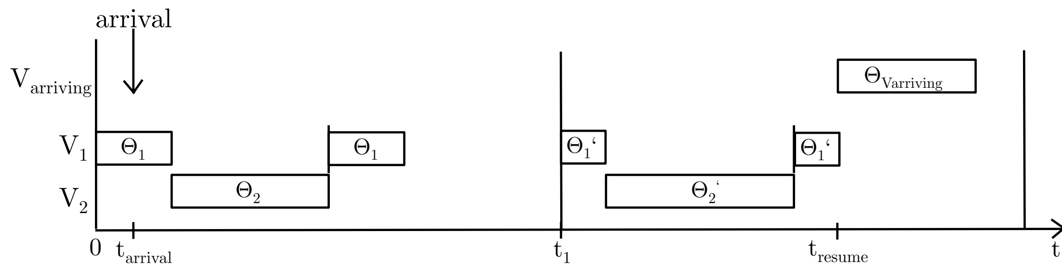


Figure 7.5: Additional delay by virtual machine scheduling: example

This delay has to be considered in addition to the downtime when checking whether the migration causes deadline misses. For the downtime test, it can be overestimated by $2 \times \Pi^{max}$, with Π^{max} being the largest period of the already hosted servers on a potential target PM. Π^{max} must therefore be communicated by a potential target hypervisor to the source hypervisor as response to a `RequestMigration` message.

7.4 Evaluation

7.4.1 Experimental Setup

For the following evaluation, RAPTOR prototyping boards [Porrmann et al., 2009] with IBM PowerPC 405 single-core processors were connected via a standard Ethernet network (100 Mbps). An Ethernet system with standard hardware is characterized by a non-deterministic timing behavior and for a real application had to be replaced by an industrial real-time Ethernet standard such as PROFINET [Feld, 2004] (incl. clock synchronization). The controlled testing network environment enabled nevertheless the empirical evaluation.

The execution times were measured with a logic analyzer. Software routines were added to the source code that set General Purpose Input/Output (GPIO) pins. Probes of the logic analyzer were connected to these dedicated GPIO pins and the logic analyzer captured the signals and was triggered on a rising edge of a dedicated signal. The input signals were sampled at regular intervals with a sampling period of 1.25 ns (sampling frequency of 800 MHz).

Table 7.1: Memory footprint for migration functionality

Feature	Memory Footprint [bytes]		
	text	data	total
Base Hypervisor	8224	2980	11204
Paravirtualization	252	148	400
Bandwidth Redistribution	2014	316	2330
Ethernet Driver	1068	1522	2590
Migration	2038	656	2694
Total	13596	5622	19218

7.4.2 Memory Footprint & Paravirtualization Effort

The Ethernet driver adds less than 3 kB to the Proteus hypervisor as introduced in Chapter 3. The migration functionality consisting of Migration Manager, Load Information Collector, and Communication module accounts for another 3 kB. If all features required for migration are enabled, plus Bandwidth Redistribution as introduced in the previous chapter, the memory requirement of the hypervisor sums up to about 19 kB (see Table 7.1).

The implementation of the presented approach requires paravirtualization: functionality for the interaction with the hypervisor interface has to be added to the guest OS, since it has to pass information to the hypervisor. Because of the limited applicability of paravirtualization, the hypervisor supports both kinds and the concurrent hosting of paravirtualized guests and fully virtualized guests is possible without restriction. Migration is however confined to paravirtualized guests.

In order to paravirtualize an OS for the presented migration approach, the OS has to implement the protocol-compliant passing of scheduling information to the hypervisor. The shared memory communication primitives are provided by a library. The function to compute the maximum possible downtime has to be implemented. The control flow has to be adapted: after a context switch from hypervisor to OS, the OS has to read out the shared memory and respond with the appropriate function calls, write accesses to shared memory, and hypercalls. In case of our real-time OS and a simple static scheduling algorithm, the paravirtualization effort accounted for 450 lines of C++ code. Please note that the paravirtualization effort is highly dependent on the characteristics of the applied scheduling algorithm, particularly the complexity of slack computation.

Table 7.2: Execution times of migration routines

Process Step	Time [μs]
Acquisition of Data from Target (incl. Acceptance Test)	61
Select Target	1
Downtime Test	18
Initiation of Transfer by VM	8
Detach t_{detach}	4
Resume t_{resume}	4

7.4.3 Execution Times & Downtime

Table 7.2 lists the execution times for the different routines of the migration process as measured with the prototype. The execution time of all routines is in the range of 1 to 61 microseconds. All steps except of the transfer sum up to less than 100 microseconds. The time for the transfer is heavily dependent on the size of the VM image and not denoted. It might in addition be delayed by other communication load, e.g., between cooperating systems on different PMs. An average transfer rate of 35 Mbps was measured, but there is an additional communication overhead for the handling of the Ethernet frames.[1]

Virtual machine configurations with a size of 1 kB, 4 kB, 16 kB, 64 kB, 256 kB, 1 MB, 4 MB, and 16 MB were migrated. VM context and hypervisor data associated with the VM add 412 bytes, independent of the memory size. Table 7.3 lists the total migration time for each configuration (from the very beginning of the protocol to completion), the measured downtime ($t_{service}$ not included, so $t_{down} - t_{service}$), and the time for only the transfer. As a simplification, the hypervisor on both source and target PM can be executed immediately and all the time and does not have to interrupt itself in order to schedule other guests (and does not schedule the migrating VM anymore). A real implementation had to integrate the migration processor into the VM schedule, e.g. by reserving dedicated computation bandwidth. In order to avoid that this bandwidth is unused when no migration process has to be managed, a mode change with redistribution should be performed. But the required utilization has to be included in the system's overall sum of minimum bandwidths. Moreover, there is no other communication load on the Ethernet connection, so the transfer is

[1] The nominal rate of 100 Mbps is not reached due to the connection of the Ethernet device via a Processor Local Bus interface and memory-mapped I/O, as well as due to a rudimentary Ethernet driver.

Table 7.3: Migration time for different virtual machine sizes

VM Size	1 kB	4 kB	16 kB	64 kB
Total Migration Time [ms]	0.915	1.479	5.614	21.310
Downtime [ms]	0.809	1.374	5.508	21.198
Transfer Time [ms]	0.801	1.366	5.500	21.190
VM Size	256 kB	1 MB	4 MB	16 MB
Total Migration Time [ms]	83.690	334.400	1336.340	5389.630
Downtime [ms]	83.588	334.308	1336.208	5389.508
Transfer Time [ms]	83.580	334.300	1336.200	5389.500

uninterrupted.

The measured downtimes indicate that the overhead of the presented implementation is too high for real-time systems with a maximum affordable downtime below 1 ms. For a memory size of 4 kB, it is feasible if the VM can be suspended for at least 1.4 ms. A VM with 16 kB memory requires at least 5.5 ms. For VMs with a memory of size greater than or equal to 4 MB, the downtime exceeds already one second.

Clearly, and not surprising, the time for the transfer of the VM image is the lion's share of the migration costs. Detaching and resuming contribute only 8 microseconds to the downtime. In order to reduce the downtime and increase the applicability of migration, one can either increase the transfer rate or decrease the amount of data that has to be transferred. One possibility is an offline distribution of the code segments on all PMs. Drawback is a higher memory usage for the redundant storage. The avoidance of a single deadline miss might not be achievable for a given VM size and communication speed. Especially with the motivation to increase the availability, that is to say continue the functioning of a system that otherwise would not be available anymore, a limited number of deadline misses could be acceptable for some systems, since a non-functioning system misses all of its deadlines.

7.4.4 Reliability Analysis

A reliability model is used in the following in order to quantify the impact of migration on virtual machine reliability. We perform a comparative reliability analysis of the fault-tolerant architecture with migration and the migration-less design. Figure 7.6 depicts the system with regard to how the reliability of its components affects the system reliability as a series-parallel reliability block diagram (RBD) [Ebeling,

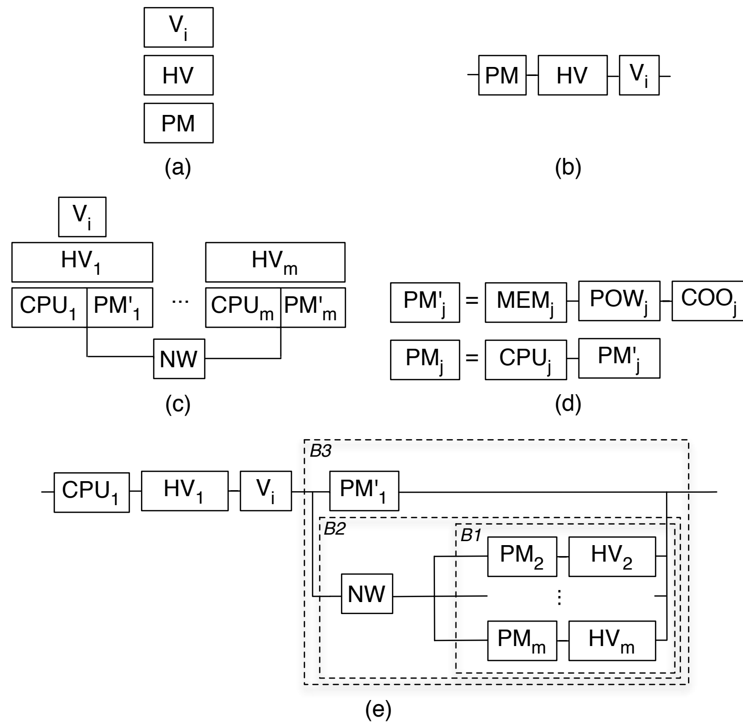


Figure 7.6: (a) System architecture of virtualization without migration: physical machine (PM), hypervisor (HV) and virtual machine (V); (b) reliability block diagram of system without migration; (c) system architecture with connected PMs; (d) reliability block diagrams of submodules; (e) reliability block diagram of system with migration

1997]. It includes structure and RBD of the architecture without migration (Figure 7.6(a),(b)) and with migration (Figure 7.6(c)–(e)). Each component is characterized by a constant failure rate, i.e., the failures are independent of time. The failures of all components are mutually independent. Components either operate correctly or fail (bi-modal). All m PMs have the same hardware failure rate.

The hardware of a PM is composed of CPU, memory (MEM), power supply (POW), and cooling system (COO) (Figure 7.6 (d)). This subdivision is useful, since we differentiate between failures that still permit a migration and those that do not. It is assumed that failures of memory, power supply, and cooling system are detected early enough by built-in self-tests to permit migration. A failure of CPU or the software entities Hypervisor (HV) and the VM itself (i.e., the operating system) render a migration impossible. The overall system succeeds, if any path through the system is successful, otherwise it fails. A VM fails if either the PM, the HV, or the VM itself fails. If a working combination of PM and HV remains, migration

Table 7.4: Mean time to failure for all components

Component	**MTTF [h]**	**Component**	**MTTF [h]**
CPU	2,500,000	Network (NW)	1,000,000
Memory (MEM)	480,000	HV	876,000
Power supply (POW)	670,000	VM (V)	876,000
Cooling system (COO)	3,100,000		

can enable the continuing of the VM's service, however, only if the network (NW) is working. A failure of a VM does not cause a failure of the other VMs (the RBD depicts only a single VM).

Table 7.4 lists the mean time to failure (MTTF) for all components of the system. All values for hardware components are retrieved from [Kim et al., 2009], since the vendors of the used hardware do not provide reliability information. Increased is the MTTF of the network, since similar network facilities for industrial automation are characterized by MTTFs in the range of decades.

The failure rate λ is the arrival rate of failure:

$$MTTF = \frac{1}{\lambda} = \int_0^\infty R(t)dt$$

The failure rates λ of the blocks $B1$, $B2$, $B3$ (see Figure 7.6(e)), and the overall system can be derived as:

$$\begin{aligned}
\lambda(B1) &= \frac{1}{\sum_{j=1}^{m-1} \frac{1}{j\cdot(\lambda(PM)+\lambda(HV))}} \\
\lambda(B2) &= \lambda(NW) + \lambda(B1) \\
\lambda(B3) &= \frac{1}{\frac{1}{\lambda(PM_1')} + \frac{1}{\lambda(B2)} - \frac{1}{\lambda(PM_1')+\lambda(B2)}} \\
\lambda &= \lambda(CPU) + \lambda(HV) + \lambda(V) + \lambda(B3)
\end{aligned}$$

Figure 7.7 plots the reliability function for configurations with one to six PMs according to the exponential failure distribution $R(t) = e^{-\lambda t}$. It underlines the effectiveness of migration as a measure to increase reliability. Table 7.5 lists the MTTFs for the different configurations and the percentage increase compared to the architecture with a single PM and consequently no migration. The increase becomes less as the number of PMs increases, since more PMs increase only the reliability of block $B1$, but the overall reliability is highly dependent on the components whose working is prerequisite for migration (network, CPU, hypervisor, VM).

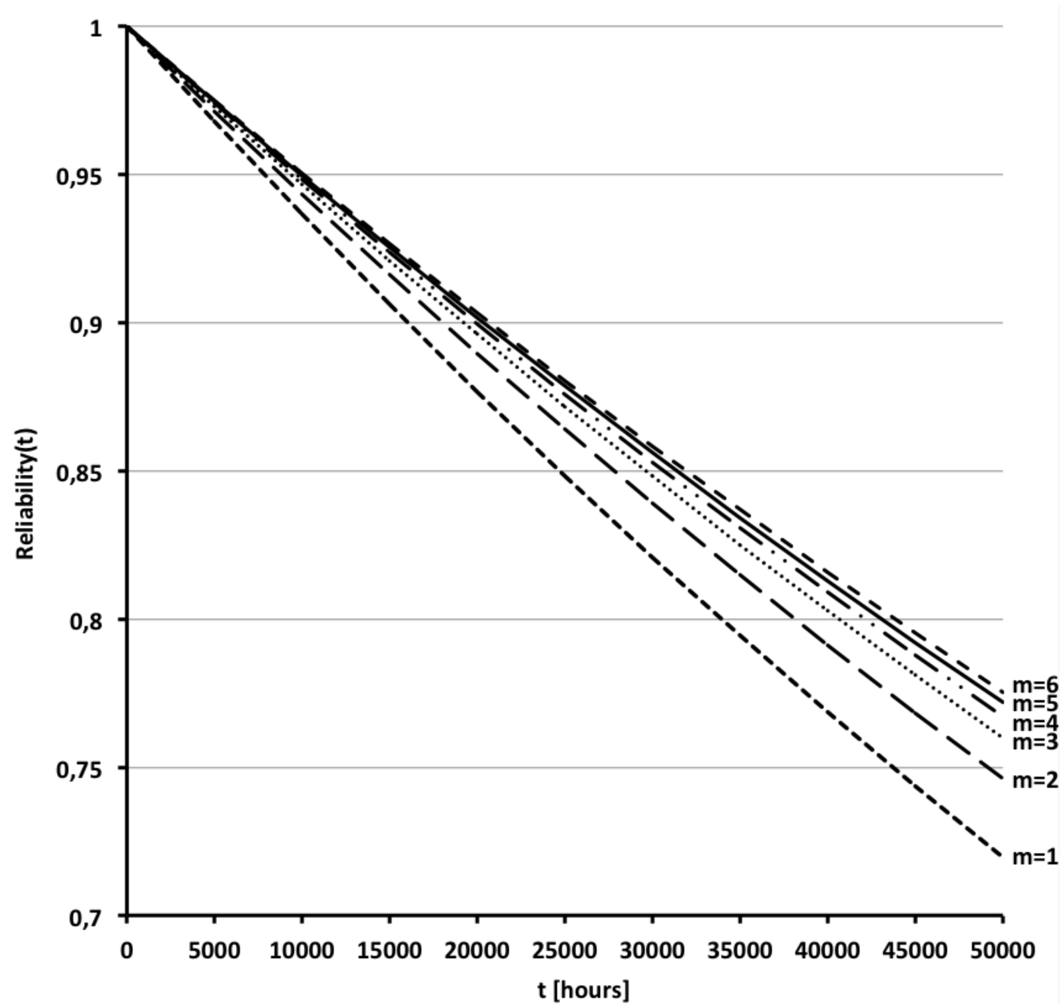


Figure 7.7: Reliability function for m=1 to 6 PMs

Table 7.5: Resulting mean time to failure

Number of PMs	**1**	**2**	**3**	**4**	**5**	**6**
MTTF [d]	6331	7114	7580	188510	7855	8180
Gain	-	12%	20%	24%	27%	29%

Figure 7.8: RailCabs on test track

7.4.5 Case Study: Autonomous Rail Vehicle

In the following case study, the applicability of virtual machine migration for the *RailCab* is evaluated. The RailCab project[2] of the University of Paderborn developed an innovative railway system with the goal to combine the flexibility of individual transport with the ecological efficiency of public transport [Gausemeier et al., 2014]. Compared to trains, RailCabs are smaller and able to transport either 10 to 20 passengers or a standard 20-foot-long intermodal container [Lückel et al., 2008]. They are autonomous and driverless and operate in a demand-driven manner, not based on schedules. The RailCab features a doubly-fed linear motor, modern chassis technology with active steering, an active suspension system, and allows convoy driving without mechanical coupling. A test track was built for the validation of this complex mechatronic system (see Figure 7.8).

In the following, the architecture of the the RailCab is introduced in order to evaluate the applicability of virtual machine migration and to identify options for the embedding of this technique. The general control architecture of the RailCab is based on the Operator Controller Module (OCM) [Hestermeyer et al., 2004] as depicted in Figure 7.9. On the highest level of this hierarchical architecture, the cognitive operator applies machine learning approaches and planning algorithms to optimize the behavior of the system. On the intermediate level of the OCM, the reflective operator represents the interface between underlying controller and cognitive operator. It receives input from the cognitive operator and monitors and configures the controller. Finally, the controller on the lowest level interfaces directly with the

[2] http://www.railcab.de/

controlled process by actuators and sensors.

Since such a transportation system is safety-critical, a modular safety system has been integrated into the system architecture [Henke et al., 2008]. In order to detect faults and to describe the state of the system, all signals, components, and modules have to be checked continuously during runtime. The reflective operator of the Operator Controller Module handles hazardous incidents, as depicted in Figure 7.10. Failures are evaluated by means of a hazard list, which is the result of a hazard analysis of the RailCab system. Hazards must be either eliminated, reduced, or controlled, or the damage caused must be minimized [Leveson, 1995].

Virtual machine migration is an additional technique to control hazards, that is to say to reduce the likelihood that the hazards pose a threat to life, health, property, or environment. A typical hazard reaction is the transition to fail-safe mode, in which it will cause no harm. Fail-safe means an immediate stopping of the vehicle. However, this hazard reaction has a negative impact on other vehicles, since it implies a blocking of a segment of the track. The RailCab project envisions networks of thousands of autonomous vehicles, building convoys where possible to save energy. Migration's potential to keep a software component alive despite of a hardware failure could be used to implement an additional fail-operational approach, which realizes in some cases a controlled driving to a maintenance facility.

The applications on the different levels of the OCM have differing timing requirements. The actuator controlling software on the lowest level and the supervising reflective operator have to meet hard real-time requirements. The self-optimization procedures of the cognitive operator are characterized by soft real-time requirements, since a violation of the timing requirements does not lead to a safety-critical malfunctioning of the mechatronic system. This highest level of the hierarchical structure is characterized by the weakest real-time requirements, and migration with the involved downtime is feasible. However, two examples demonstrate that migration is actually applicable as well to actuator-controlling low-level software, despite of hard real-time requirements: a VM with the linear motor control and a VM with the active guidance module.

The linear motor is controlled by two current controller tasks τ_{c1} and τ_{c2} and a speed controller task τ_s, executed according to a fixed-priority policy on a dSPACE DS1005 PPC board with a PowerPC 750GX processor of 1 GHz [Kerstan, 2011]. Their execution times and periods are stated in Table 7.6.

The maximum affordable downtime is 2.5 ms (see Figure 7.11), resulting in a feasible real-time migration of a VM with a memory size of up to 4 kB (according to

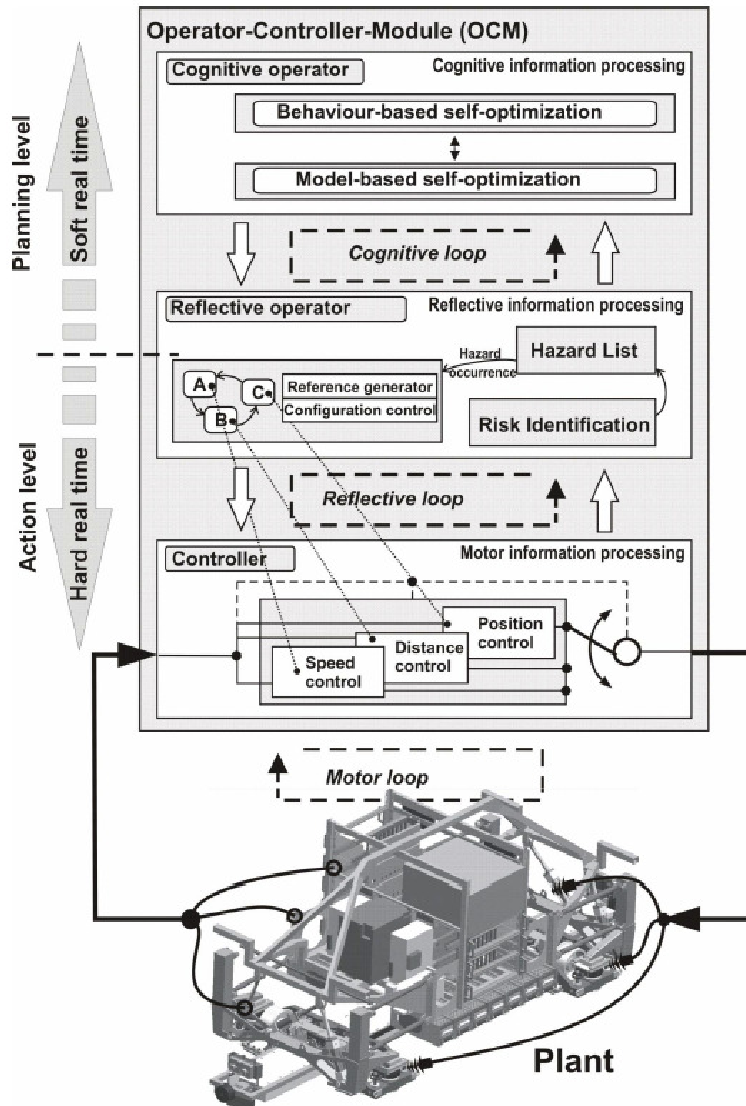


Figure 7.9: Control architecture for self-optimizing mechatronic systems: operator controller module [Hestermeyer et al., 2004]

Linear Motor Control	Task	WCET [ms]	Period [ms]
Current Control	τ_{c1}	0.25	3
	τ_{c2}	0.25	3
Speed Control	τ_s	0.25	42

Table 7.6: Case study: electrical drive engineering - linear motor control

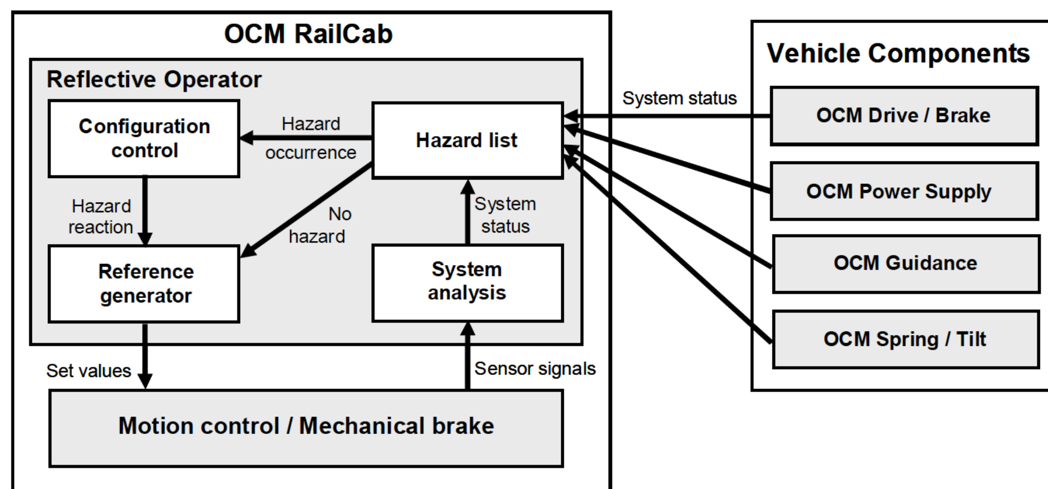


Figure 7.10: Hazard handling of the RailCab [Henke et al., 2008]

Table 7.3). With 18 kB, the size of the operating system ORCOS exceeds already this limit. In order to apply migration nevertheless, one can preload the code segments of both operating system and tasks to all potential target PMs. In this case, only the data that defines the state of the controller tasks ($<$ 1 kB) has to be transferred. The big drawback is the additional memory requirement, which grows exponentially with the overall number of VMs in the distributed system.

RailCabs group to convoys in order to reduce reduce drag and, consequently, reduce energy consumption. The RailCabs are not mechanically coupled, but drive with low distance from each other. To enable a RailCab to leave such a convoy at high velocities of 160 km/h, the railroad switch is passive and the RailCab steers actively onto the target track. The control of the steering angle of the axles is realized by the *active guidance module*. This module controls the steering angle not only in case of a railroad switch, but permanently, in order to reduce wear on wheels and rails by compensating track irregularities [Sondermann-Wolke and Sextro, 2009]. The controller is characterized by a period of 80 ms and a WCET of 25 ms [Geisler, 2014]. Referring again to Table 7.3, a virtual machine of size 64 kB can be migrated in a timely manner. This limit actually allows for the migration of the entire VM, incl. operating system.

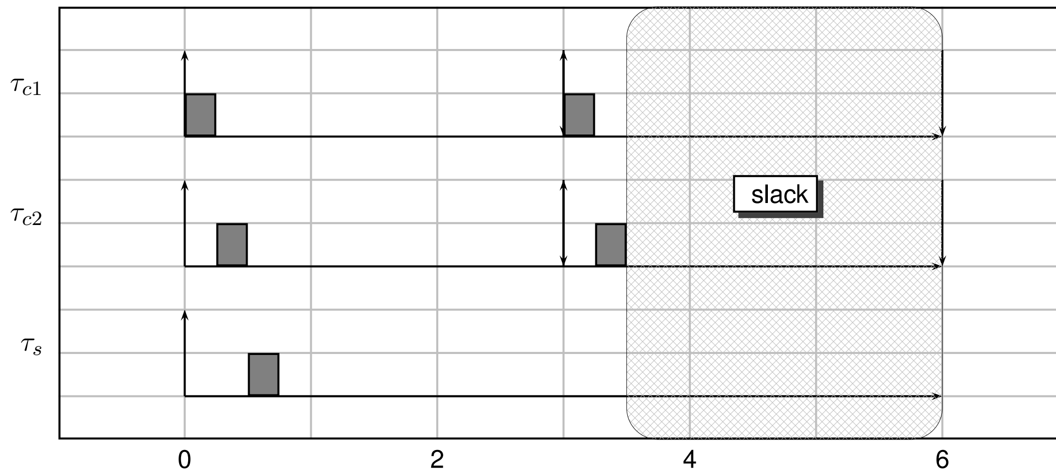


Figure 7.11: Linear motor control case study: maximum downtime

7.5 Summary

Migration refers to the relocation of a virtual machine from one physical machine to another one at runtime. Virtualization's architectural abstraction and encapsulation of guest systems in virtual machines facilitate migration, but existing virtualization solutions for real-time embedded systems are characterized by a static mapping of virtual machines to processors. As a coarse-grained approach with a significant overhead, particularly regarding the downtime due to the transmission from source to target physical machine, this chapter proposed to apply it as a fault tolerance technique in order to continue the functioning of a subsystem despite a hardware failure.

This work studied migration of virtual machines with real-time constraints on homogeneous multiprocessor architectures. The migration policy respects real-time requirements and predicts deadline misses based on a preceding comparison of downtime caused by the migration and slack-based computation of the virtual machine's maximum affordable downtime. The distributed design is characterized by a communication between the paravirtualized operating system and the hypervisor in order to provide the required scheduling information. A prototype was integrated into hypervisor and real-time operating system. The evaluation showed a reasonable paravirtualization effort, a low overhead of the migration protocol, and dominating costs for the data transfer. The transfer of the memory content was indeed identified as the limiting factor for real-time migration. On the low-performance test hardware, the approach is applicable to real-time systems with a virtual machine size up to

about 64 kB and deadlines in the range of multiple milliseconds. A reliability analysis based on a combinatorial model quantified the positive impact of migration on reliability.

Chapter 8

Conclusion & Future Work

The increasing number of functions and the goal to reduce or at least maintain hardware costs, space, weight, and power consumption lead in industries such as the automotive or aerospace industry to integrated architectures, which consolidate multiple functions on a shared electronic control unit. Multicore processors provide an ideal hardware platform to reconcile these opposing trends: realize complex functionality, but at the same time reduce the number of control units. The major challenges for integrated architectures are robust encapsulation (to prevent that the integrated systems corrupt each other) and resource management (to ensure that each system receives sufficient resources).

System virtualization is a promising integration technique that can provide the required reliability and resource management features. It integrates multiple software systems (operating system and application tasks) in an encapsulated manner. The hypervisor manages the hardware resources and provides multiple execution environments. The benefits are hardware consolidation, operating system heterogeneity, easy migration of single-core software to multicore processors, secure partitioning, and incremental certification. The hypervisor has to implement the encapsulation and is responsible for the management of the hardware resources. Temporal isolation is a prerequisite for the integration of real-time systems and the certification of safety-critical systems, but adaptability is desired in order to utilize the shared processor efficiently.

8.1 Summary of Results

In this context of hypervisor-based virtualization for embedded real-time systems, this thesis investigated the adaptive management of the resource computation time

and made the following contributions:

An algorithm for the partitioning of virtual machines to processor cores. (Chapter 5) An algorithmic solution is in contrast to the manual partitioning, which is state of the art. The algorithm includes the correct dimensioning of the periodic resources and guarantees in combination with the scheduling algorithm of Chapter 6 the schedulability of all guest systems. It is original in that it minimizes the overall required computation bandwidth by exploiting the freedom of server design to create favorable period relationships. In addition, it considers criticality levels and produces partitions that provide more possibilities to protect a safety-critical guest system and to benefit from an adaptive scheduling.

An adaptive virtual machine scheduling architecture. (Chapter 6) It advances the state of the art by combining temporal isolation and real-time guarantees with adaptive bandwidth management. The technique overcomes the limitations of static resource allocation. Mode changes and varying execution times trigger a redistribution and the approach attempts to protect critical guest systems in case of a worst-case execution time overrun. A novel elastic bandwidth management algorithm is non-iterative, in contrast to existing ones, and therefore characterized by a smaller and more predictable execution time overhead. An analysis proved the service guarantee of the scheduling architecture, which is the basis for its real-time capability and a potential certification.

A technique for real-time virtual machine migration. (Chapter 7) It advances the state of the art, in that the technique is aware of real-time requirements and addresses the issues service outage due to the network transfer and the integration into the scheduling on the target control unit. The migration protocol, the co-design of hypervisor and paravirtualized guest operating system, and an efficient implementation were presented.

In addition, an architecture of a multicore hypervisor that provides real-time capability, safe and secure partitioning, and support of both paravirtualization and full virtualization was presented. The implementation meets the requirements of embedded real-time systems regarding execution time overhead, latencies, and memory footprint. This hypervisor served as a platform for the prototype-based evaluation of the contributions. It is not a scientific contribution on its own, but represents the state of the art.

Together, these contributions enable the integration of independently developed and validated guest systems on top of the hypervisor. The hypervisor's virtual machine scheduling guarantees that all guest systems receive sufficient computation time in order to meet their real-time requirements. This includes that the execution of a guest is never interrupted for a longer time than allowed by its reactivity requirements. The redistribution of the bandwidth in case of mode changes and execution time variations increases the utilization and supports applications that can take advantage of additional computational bandwidth. Adaptive measures are taken as well to protect critical guest systems. In case of a worst-case execution time overrun of a critical guest system, it is attempted to protect its execution by assigning additional computation time, which is stolen from non-critical guests. In case of a hardware failure, migration is performed to continue the operation of guest systems on other processors.

8.2 Outlook

In future work, we plan to remove the constraint that only independent virtual machines are considered. If systems that have to communicate are consolidated, inter-VM communication is required. The hypervisor provides already the required functionality, but partitioning, scheduling, and migration do not yet include this aspect. The partitioning has a direct influence on communication latencies and overhead. The scheduling might be influenced by precedence constraints (a guest system cannot continue before another guest system finished a certain task). If migration separates systems that formerly shared a processor, the local inter-VM communication has to be replaced by inter-processor communication, preferably transparent to the guest systems.

In addition, guest systems so far have to obtain a dedicated I/O device. Support for I/O device sharing between guest systems should be added. There are different options for the implementation, as introduced in Chapter 2.2.3, e.g., emulation by the hypervisor, paravirtualized device drivers, or hardware multiplexing. As a next step, the I/O bandwidth could be managed as well in an adaptive manner, for example by a server-based approach [Santos et al., 2011]. It would then be interesting to explore the relationship between adaptively managed computation bandwidth and adaptively managed communication bandwidth, and their mutual impact. First works in this direction exist, but only for non-real-time systems [Cherkasova and Gardner, 2005, Gupta et al., 2006, Ongaro et al., 2008].

Appendix A

Publications

Hypervisor (Chapter 3)

K. Gilles, S. Groesbrink, D. Baldin, T. Kerstan: *Proteus Hypervisor: Full Virtualization and Paravirtualization for Multi-Core Embedded Systems.* Proc. of the 4th IFIP International Embedded Systems Symposium, Paderborn, Jun. 2013.

Virtual Machine Partitioning (Chapter 5)

S. Groesbrink: *On the Homogeneous Multiprocessor Virtual Machine Partitioning Problem.* Proc. of the 4th IFIP International Embedded Systems Symposium, Paderborn, Jun. 2013.

S. Groesbrink, L. Almeida: *A Criticality-Aware Mapping of Real-time Virtual Machines to Multicores.* Proc. of the 19th IEEE International Conference on Emerging Technology & Factory Automation, Barcelona, Sep. 2014.

Adaptive Virtual Machine Scheduling (Chapter 6)

S. Groesbrink, L. Almeida, M. de Sousa, S. M. Petters: *Fair Bandwidth Sharing among Virtual Machines in a Multi-criticality Scope.* Proc. of the 5th Workshop on Adaptive and Reconfigurable Embedded Systems, CPSWeek 2013, Philadelphia, Apr. 2013.

S. Groesbrink, L. Almeida, M. de Sousa, S. M. Petters: *Towards Certifiable Adaptive Reservations for Hypervisor-based Virtualization*, Proc. of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium, Berlin, Apr. 2014.

Virtual Machine Migration (Chapter 7)

S. Groesbrink: *Basics of Virtual Machine Migration on Heterogeneous Architectures for Self-optimizing Mechatronic Systems - Necessary Conditions and Implementation Issues.* Production Engineering Research & Development, Volume 7, Issue 1, pp. 69-79, Springer, Jan. 2013.

S. Groesbrink: *Virtual Machine Migration as a Fault Tolerance Technique for Embedded Real-Time Systems.* Proc. of the 8th IEEE Reliability Society's International Conference on Software Security and Reliability, San Francisco, Jun. 2014.

Applications

S. Groesbrink, F. Rammig: *Safe Self-Evolving Embedded Software via System Virtualization.* Proc. of the 3rd SBC Workshop on Autonomic Distributed Systems, Brasilia, May 2013.

S. Groesbrink, D. Baldin, S. Oberthür: *Architecture for Adaptive Resource Assignment to Virtualized Mixed-Criticality Real-Time Systems.* ACM SIGBED Review, Volume 10, Mar. 2013.

S. Groesbrink, S. Korrapati, A. Schmitz, A. Schreckenberg: *Hypervisor-based Consolidation for Automated Teller Machines.* Proc. of the 12th Embedded World Conference, Nürnberg, Feb. 2014.

S. Groesbrink: *Increasing the Reusability of Embedded Real-time Software by a Standardized Interface for Paravirtualization.* Proc. of the 6th GI Design For Future Workshop, Bad Honnef, Apr. 2014.

F. Rammig, S. Groesbrink, K. Stahl, Y. Zhao: *Designing Self-Adaptive Embedded Real-time Software - Towards System Engineering of Self-Adaptation.* Proc. of the 4th Brazilian Symposium on Computing Systems Engineering, Manaus, Nov. 2014.

List of Figures

1.1 Candidate functions for hypervisor-based head-unit integration . . . 5

2.1 Computer system with operating system 15

2.2 Parameters of a real-time task (upward arrow indicates task arrival, downward arrow indicates deadline) 18

2.3 Platform replication by system virtualization 23

2.4 (a) Native hypervisor (type I) vs. (b) hosted hypervisor (type II) . . 25

2.5 Popek and Goldberg's requirement for a virtualizable instruction set architecture: is the set of sensitive instructions a subset of the set of privileged instructions? (a) not fulfilled; (b) fulfilled 27

2.6 Single-bus shared memory multicore with private caches 33

2.7 System software's core management: symmetric multiprocessing, asymmetric multiprocessing, and hypervisor-based virtualization 34

2.8 System virtualization's hierarchical scheduling: virtual machine scheduling by hypervisor and task scheduling by operating systems 38

2.9 Progress of virtual machines: (a) ideal; (b) in practice (cf. [Kaiser, 2008]) 39

3.1 Design of the Proteus hypervisor (cf. [Baldin and Kerstan, 2009]) . . 51

3.2 Preprocessor-based configuration by conditional compilation 54

3.3 Execution mode differentiation and mechanisms for mode transition: extension of the processor's two modes by two virtual modes 56

3.4 Execution time of routines for protected access to a shared resource . 63

3.5 Impact of individual components on memory footprint 66

4.1 Distribution of execution times of a task between best-case and worst-case execution time 71

4.2 Demand bound function for two EDF-scheduled tasks $\tau_1 = (T = 2, C = 1), \tau_2 = (T = 5, C = 2)$ (the dashed line depicts the schedulability bound on a dedicated processor) 75

4.3 Service delay of a periodic resource $\Gamma = (\Pi, \Theta)$ 77

4.4 Schedulability analysis by comparing demand bound function and supply bound function. Example: two EDF-scheduled tasks $\tau_1 = (T = 5, C = 1), \tau_2 = (T = 15, C = 2)$ and periodic resource $\Gamma = (\Pi = 2, \Theta = 1)$ 79

5.1 Tree-based visualization of the systematic enumeration of candidate solutions — Example: on level 2, V_2 might either be added to the same partition as V_1 or a second partition is created. 94

5.2 Existence of multiple periodic resources. Example: two EDF-scheduled tasks $\tau_1 = (T = 5, C = 1), \tau_2 = (T = 15, C = 2)$ and two periodic resources $\Gamma_1 = (\Pi = 2, \Theta = 1)$, $\Gamma_2 = (\Pi = 5, \Theta = 3)$, both guarantee schedulability 95

5.3 Schedulable region of a periodic resource. Example: two EDF-scheduled tasks $\tau_1 = (T = 50, C = 7), \tau_2 = (T = 75, C = 9)$. Integral resource allocation minimums for $1 \leq \Pi \leq 10$ 96

5.4 Mappings for different optimization goals 104

5.5 Average number of processor cores for different partitioning goals: minimize number of processor cores (NP), minimize number of processor cores incl. transformation to harmonic server periods (NPH); maximize criticality distribution (CD), incl. transformation to harmonic server periods (CDH); minimize number of processor cores and assign cores exclusively to critical VMs (EX), incl. transformation to harmonic server periods (EXH); (a) VM Bandwidths between 0.1 and 0.9, (b) VM Bandwidths between 0.1 and 0.5 106

5.6 Average criticality distribution for the partitioning goal minimize number of processor cores, with (NPH) and without (PH) transformation to harmonic server periods; (a) VM bandwidths between 0.1 and 0.9, (b) VM bandwidths between 0.1 and 0.5 107

6.1 Server-based partitioned hierarchical scheduling 115

6.2 Server-based scheduling of two virtual machines V_1 and V_2: the virtual processors $\Gamma_1(2, 1)$ and $\Gamma_2(5, 2)$ are implemented by periodic servers . 116

6.3 State transition diagram for the periodic server 117

6.4 Example of overload reaction: overload of V_1 at $t = 7$; (a) next-to-dispatch VM V_2 is non-critical; (b) next-to-dispatch VM V_2 is critical 125
6.5 Mode change request of V_1 at $t = 4$: immediate mode change leads to overallocation 130
6.6 Case 1: request to enable V_1 at $t = 1$, activation at $t = 16$ 131
6.7 Case 2: request to disable V_2 at $t = 14$, activation at $t = 16$ (end of V_2's instance) 132
6.8 Case 3: mode change request of V_2 at $t = 3$ (from a mode with $U_{lax}(V_2) = 0$ to a mode with $U_{lax}(V_2) = 0.125$; assumed is a higher weight of V_2 compared to V_3): the allocation to V_1 of higher criticality is unchanged, V_2's larger allocation is activated at the end of V_3's instance (V_3's allocation is reduced at the same point in time) 132
6.9 Limited validity duration of dynamic slack: timing violation due to use of dynamic slack later than the end of the period of the giving virtual machine 134
6.10 Limited validity duration of dynamic slack: timing violation of VM of intermediate priority due to use of dynamic slack later than the end of the instance of the receiving VM (example: $U_{min}(V_1) = 1/4$, $U_{lax}(V_1) = 3/4$, $U_{min}(V_2) = 1/8$, $U_{lax}(V_2) = 0$, $U_{min}(V_3) = 1/24$, $U_{lax}(V_3) = 23/24$) 135
6.11 Categorization of all VMs based on priority relative to giving virtual machine and receiving virtual machine for the case that V_{giving} is of lower priority 137
6.12 Need to activate virtual machines in a synchronized manner (fractions denote Θ and Π) 138
6.13 Categorization of all virtual machines based on priority relative to giving virtual machine and receiving virtual machine for the case that V_{giving} is of higher priority 139
6.14 Availability of information with influence on scheduling for hypervisor and guest systems 141
6.15 Interaction between hypervisor and operating system in the case of redistribution on mode change 144
6.16 Execution times of scheduler routines subject to the number of virtual machines (PowerPC 405 @300 MHz) 147
6.17 Interaction between hypervisor and operating system in the case of yield, with and without redistribution 149

6.18 Effect of mode change probability 157
6.19 Effect of bandwidth demand factor 159

7.1 Virtual machine migration 162
7.2 Downtime due to virtual machine migration 164
7.3 Integration of the migration functionality into the architecture of the hypervisor 168
7.4 Migration protocol 170
7.5 Additional delay by virtual machine scheduling: example 175
7.6 (a) System architecture of virtualization without migration: physical machine (PM), hypervisor (HV) and virtual machine (V); (b) reliability block diagram of system without migration; (c) system architecture with connected PMs; (d) reliability block diagrams of submodules; (e) reliability block diagram of system with migration 179
7.7 Reliability function for m=1 to 6 PMs 181
7.8 RailCabs on test track 182
7.9 Control architecture for self-optimizing mechatronic systems: operator controller module [Hestermeyer et al., 2004] 184
7.10 Hazard handling of the RailCab [Henke et al., 2008] 185
7.11 Linear motor control case study: maximum downtime 186

List of Tables

4.1 Suitability of the model: exemplary virtual machine set 81

5.1 Example for partitioning: set of virtual machines 103

6.1 Thresholds for distribution of dynamic slack 148
6.2 Memory footprint for scheduling functionality (2 virtual machines) . 149
6.3 Qualitative comparison of virtual machine scheduling techniques (n: number of VMs, r: number of real-time VMs) 153
6.4 Relative error δ of allocated bandwidth and desired bandwidth for fixed distribution and adaptive distribution 156

7.1 Memory footprint for migration functionality 176
7.2 Execution times of migration routines 177
7.3 Migration time for different virtual machine sizes 178
7.4 Mean time to failure for all components 180
7.5 Resulting mean time to failure 181
7.6 Case study: electrical drive engineering - linear motor control 184

List of Algorithms

1 Harmonization 98
2 Partitioning Algorithm 102
3 Bandwidth Distribution 122

Bibliography

[Aalto, 2010] Aalto, A. (2010). Dynamic Management of Multiple Operating Systems in an Embedded Multi-Core Environment. Master's thesis, Aalto University.

[Abeni et al., 2005] Abeni, L., Cucinotta, T., Lipari, G., Marzario, L., and Palopoli, L. (2005). QoS Management through Adaptive Reservations. *Real-Time Systems*, 29:131–155.

[Adams and Agesen, 2006] Adams, K. and Agesen, O. (2006). A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13.

[Almeida and Pedreiras, 2004] Almeida, L. and Pedreiras, P. (2004). Scheduling within Temporal Partitions: Response-time Analysis and Server Design. In *Proc. of the Conference on Embedded Software.*

[Almeida et al., 2002] Almeida, L., Pedreiras, P., and Fonseca, J. (2002). The FTT-CAN Protocol: Why and How. *IEEE Transactions on Industrial Electronics*, 49(6):1189–1201.

[Anderson et al., 2009] Anderson, J., Baruah, S., and Brandenburg, B. (2009). Multicore Operating-System Support for Mixed Criticality. In *Proc. of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification.*

[Andersson and Johnsson, 2000] Andersson, B. and Johnsson, J. (2000). Fixed-priority Preemptive Multiprocessor Scheduling: to Partition or not to Partition. In *Proc. of the International Workshop on Real-Time Computing Systems and Applications.*

[Arzen et al., 2011] Arzen, K., Segovia, V. R., Schorr, S., and Fohler, G. (2011). Adaptive Resource Management Made Real. In *Proc. of the Workshop on Adaptive and Reconfigurable Embedded Systems.*

[Asberg et al., 2009] Asberg, M., Behnam, M., Nemati, F., and Nolte, T. (2009). Towards Hierarchical Scheduling in AUTOSAR. In *Proc. of the IEEE Conference on Emerging Technologies and Factory Automation*.

[Audsley et al., 1991] Audsley, N., Burns, A., Richardson, M., and Wellings, A. (1991). Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proc. of the IEEE Workshop on Real-Time Operating Systems*.

[Augier, 2007] Augier, C. (2007). Real-Time Scheduling in a Virtual Machine Environment. In *Proc. of the Junior Researcher Workshop on Real-Time Computing*.

[Baker and Shaw, 1989] Baker, T. and Shaw, A. (1989). The Cyclic Executive Model and Ada. *Real-Time Systems*.

[Baldin, 2009] Baldin, D. (2009). Entwurf und Implementierung einer komponentenbasierten Virtualisierungsplattform für selbstoptimierende eingebettete mechatronische Systeme. Master's thesis, University of Paderborn.

[Baldin and Kerstan, 2009] Baldin, D. and Kerstan, T. (2009). Proteus, a Hybrid Virtualization Platform for Embedded Systems. In *Proc. of the International Embedded Systems Symposium*.

[Baliga and Kumar, 2005] Baliga, G. and Kumar, P. (2005). A Middleware for Control over Networks. In *Proc. of the IEEE Conference on Decision and Control*.

[Barham et al., 2003] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the Art of Virtualization. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*.

[Bartolini and Lipari, 2014] Bartolini, C. and Lipari, G. (2012 [accessed: Sep. 10, 2014]). RTSIM - Real-Time system SIMulator. http://rtsim.sssup.it/.

[Baruah, 2004] Baruah, S. (2004). Task Partitioning upon Heterogeneous Multiprocessor Platforms. In *Proc. of the Real-Time and Embedded Technology and Applications Symposium*.

[Baruah and Burns, 2006] Baruah, S. and Burns, A. (2006). Sustainable Scheduling Analysis. In *Proc. of the 27th IEEE International Real-Time Systems Symposium*.

[Baruah and Fohler, 2011] Baruah, S. and Fohler, G. (2011). Certification-Cognizant Time-Triggered Scheduling of Mixed-Criticality Systems. In *Proc. of the 32nd IEEE Real-Time Systems Symposium*.

[Baruah et al., 2010a] Baruah, S., Haohan, L., and Stougie, L. (2010a). Towards the Design of Certifiable Mixed-Criticality Systems. In *Proc. of the Real-Time and Embedded Technology and Applications Symposium*.

[Baruah et al., 2010b] Baruah, S., Li, H., and Stougie, L. (2010b). Mixed-Criticality Scheduling: Improved Resource-Augmentation Results. In *Proc. of the Conference on Computers and Their Applications*.

[Baruah et al., 1990] Baruah, S., Rosier, L., and Howell, R. (1990). Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on one Processor. *Real-Time Systems*, 2:301–324.

[Bate and Kelly, 2003] Bate, I. and Kelly, T. (2003). Architectural Considerations in the Certification of Modular Systems. *Reliability Engineering & System Safety*, 81(3):303–324.

[Behnam et al., 2008] Behnam, M., Nolte, T., Shin, I., and Asberg, M. (2008). Towards Hierarchical Scheduling in VxWorks. In *Proc. of the Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*.

[Bernat and Burns, 2002] Bernat, G. and Burns, A. (2002). Multiple Servers and Capacity Sharing for Implementing Flexible Scheduling. *Real-Time Systems*, 22:49–75.

[Bini et al., 2009] Bini, E., Buttazzo, G., and Bertogna, M. (2009). The Multi Supply Function Abstraction for Multiprocessors. In *Proc. of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

[Bini et al., 2011] Bini, E., Buttazzo, G., Eker, J., Schorr, S., Guerra, R., Fohler, G., Arzen, K.-E., Segovia, V. R., and Scordino, C. (2011). Resource Management on Multicore Systems: The ACTORS Approach. *IEEE Micro*, 31:72–81.

[Block et al., 2008] Block, A., Brandenburg, B., Anderson, J., and Quint, S. (2008). An Adaptive Framework for Multiprocessor Real-Time Systems. In *Proc. of the Euromicro Conference on Real-Time Systems*, pages 23–33.

[Bobroff et al., 2007] Bobroff, N., Kochut, A., and Beaty, K. (2007). Dynamic Placement of Virtual Machines for Managing SLA Violations. In *Proc. of the International Symposium on Integrated Network Management*.

[Bouyssonouse and Sifakis, 2005] Bouyssonouse, B. and Sifakis, J. (2005). *Embedded Systems Design: The ARTIST Roadmap for Research and Development*, volume 3436 of *Lecture Notes in Computer Science*. Springer.

[Brakensiek et al., 2008] Brakensiek, J., Droege, A., Botteck, M., Haertig, H., and Lackorzynski, A. (2008). Virtualization as an Enabler for Security in Mobile Devices. In *Proc. of the 1st Workshop on Isolation and Integration in Embedded Systems*.

[Brandenburg, 2014] Brandenburg, B. (2013 [Jan. 6, 2014]). Schedcat: the Schedulability Test Collection and Toolkit.

[Briand and Roy, 1999] Briand, L. and Roy, D. (1999). *Meeting Deadlines in Hard Real-Time Systems - The Rate Monotonic Approach*. IEEE Computer Society.

[Broy et al., 2007] Broy, M., Kruger, I., Pretschner, A., and Salzmann, C. (2007). Engineering Automotive Software. *Proceedings of the IEEE*, 95(2):356–373.

[Bruns et al., 2010] Bruns, F., Traboulsi, S., Szczesny, D., Gonzalez, E., Xu, Y., and Bilgic, A. (2010). An Evaluation of Microkernel-based Virtualization for Embedded Real-time Systems. In *Proc. Euromicro Conference on Real-Time Systems*, pages 57–65.

[Bucher et al., 2003] Bucher, T., Curio, C., Edelbrunner, J., Igel, C., Kastrup, D., Leefken, I., Lorenz, G., Steinhage, A., and von Seelen, W. (2003). Image Processing and Behavior Planning for Intelligent Vehicles. *IEEE Transactions on Industrial Electronics*, 50(1):62–75.

[Bunzel, 2011] Bunzel, S. (2011). AUTOSAR–The Standardized Software Architecture. *Informatik-Spektrum*, 34(1):79–83.

[Burchard et al., 1995] Burchard, A., Liebeherr, J., Oh, Y., and Son, S. H. (1995). New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems. In *IEEE Transactions on Computers*, volume 44, pages 1429–1442.

[Burns and Baruah, 2008] Burns, A. and Baruah, S. (2008). Sustainability in Real-Time Scheduling. *Journal of Computing Science and Engineering*, 2(1):72–94.

[Buttazzo, 2004] Buttazzo, G. (2004). *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2 edition.

[Buttazzo, 2006] Buttazzo, G. (2006). Research Trends in Real-Time Computing for Embedded Systems. *ACM SIGBED Review*, 3(3):1–10.

[Buttazzo et al., 2011] Buttazzo, G., Bini, E., and Wu, Y. (2011). Partitioning Real-Time Applications over Multi-Core Reservations. In *IEEE Transactions on Industrial Informatics*, volume 7, pages 302–315.

[Buttazzo et al., 1999] Buttazzo, G., Lipari, G., and Abeni, L. (1999). Elastic Task Model for Adaptive Rate Control. In *Proc. of the Real-Time Systems Symposium*.

[Buttazzo et al., 2002] Buttazzo, G., Lipari, G., Caccamo, M., and Abeni, L. (2002). Elastic Scheduling for Flexible Workload Management. *IEEE Transactions on Computers*, 51(3):289–302.

[Buttazzo, 2000] Buttazzo, G. C. (2000). *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers.

[Caccamo et al., 2005] Caccamo, M., Buttazzo, G., and Thomas, D. (2005). Efficient Reclaiming in Reservation-based Real-time Systems With Variable Execution Times. *IEEE Transactions on Computers*, 54:198–213.

[Calandrino et al., 2007] Calandrino, J., Anderson, J., and Baumberger, D. (2007). A Hybrid Real-time Scheduling Approach for Large-Scale Multi-Core Platforms. In *Proc. of the Euromicro Conference on Real-Time Systems*.

[Carpenter et al., 2004] Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, J., and Baruah, S. (2004). A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models*.

[Carrascosa et al., 2013] Carrascosa, E., Masmano, M., Balbastre, P., and Crespo, A. (2013). XtratuM Hypervisor Redesign for LEON4 Multicore Processor. In *Proc. Workshop on Virtualization for Real-time Embedded Systems*.

[Catanzaro, 1994] Catanzaro, B. (1994). *Multiprocessor System Architectures: A Technical Survey of Multiprocessor/Multithreaded Systems Using Sparc, Multilevel Bus Architectures and Solaris*. PTR Prentice Hall.

[Checconi et al., 2009] Checconi, F., Cucinotta, T., and Stein, M. (2009). Real-Time Issues in Live Migration of Virtual Machines. In *Proc. of the International Conference on Parallel Processing*.

[Cherkasova and Gardner, 2005] Cherkasova, L. and Gardner, R. (2005). Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor. In *Proc. of the USENIX Annual Technical Conference*, pages 24–24.

[Choi et al., 2012] Choi, J., Oh, H., Kim, S., and Ha, S. (2012). Executing Synchronous Dataflow Graphs on a SPM-based Multicore Architecture. In *Proc. of the Design Automation Conference.*

[Clark, 2005] Clark, C. (2005). Live Migration of Virtual Machines. In *Proc. of the Symposium on Networked Systems Design & Implementation.*

[Coffman et al., 1996] Coffman, E., Garey, M., and Johnson, D. (1996). Approximation Algorithms for Bin Packing: a Survey. In *Approximation Algorithms for NP-hard Problems*, pages 46—93.

[Cucinotta et al., 2011a] Cucinotta, T., Anastasi, G., and Abeni, L. (2011a). Respecting Temporal Constraints in Virtualised Services. In *Proc. of the 4th IEEE International Workshop on Real-Time Service-Oriented Architecture and Applications.*

[Cucinotta et al., 2011b] Cucinotta, T., Giani, D., Faggioli, D., and Checconi, F. (2011b). Providing Performance Guarantees to Virtual Machines Using Real-Time Scheduling. *Lecture Notes in Computer Science*, 6586:657–664.

[Culler et al., 1999] Culler, D. E., Singh, J. P., and Gupta, A. (1999). *Parallel Computer Architecture - a Hardware / Software Approach.* Morgan Kaufmann.

[Dasari et al., 2011] Dasari, D., Andersson, B., Nelis, V., Petters, S., Easwaran, A., and Lee, J. (2011). Response Time Analysis of COTS-Based Multicores Considering the Contention on the Shared Memory Bus. In *10th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1068–1075.

[Davis and Burns, 2005] Davis, R. and Burns, A. (2005). Hierarchical Fixed Priority Pre-emptive Scheduling. In *Proc. of the 26th IEEE International Real-Time Systems Symposium.*

[Davis and Burns, 2006] Davis, R. and Burns, A. (2006). Resource Sharing in Hierarchical Fixed Priority Pre-emptive Systems. In *Proc. of the IEEE Real-time Systems Symposium.*

[Davis and Burns, 2010] Davis, R. and Burns, A. (2010). A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. In *ACM Computing Surveys*, volume 43.

[de Niz et al., 2001] de Niz, D., Abeni, L., Saewong, S., and Rajkumar, R. (2001). Resource Sharing in Reservation-Based Systems. In *Proc. of the IEEE Real-Time Systems Symposium.*

[de Niz et al., 2009] de Niz, D., Lakshmanan, K., and Rajkumar, R. (2009). On the Scheduling of Mixed-Criticality Real-Time Task Sets. In *Proc. of the Real-Time Systems Symposium.*

[Deng and Liu, 1997] Deng, Z. and Liu, J. (1997). Scheduling real-time applications in an open environment. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 308–319.

[Deng et al., 1996] Deng, Z., Liu, J., and Sun, J. (1996). Dynamic scheduling of hard real-time applications in open system environment. In *Proc. of the Real-Time Systems Symposium.*

[Deng et al., 1997] Deng, Z., Liu, J., and Sun, L. (1997). A Scheme for Scheduling Hard Real-Time Applications in Open System Environment. In *Proc. of the Euromicro Conference on Real-Time Systems.*

[Dertouzos, 1974] Dertouzos, M. (1974). Control robotics: The Procedural Control of Physical Processes. In *Information Processing*, volume 74.

[Dhall and Liu, 1978] Dhall, S. and Liu, C. (1978). On a Real-Time Scheduling Problem. In *Operations Research*, volume 26, pages 127–140.

[Dijkstra, 1965] Dijkstra, E. W. (1965). Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9).

[Douglis and Ousterhout, 1987] Douglis, F. and Ousterhout, J. (1987). Process Migration in the Sprite Operating System. In *Proc. of the Real-Time Systems Symposium.*

[Douglis and Ousterhout, 2006] Douglis, F. and Ousterhout, J. (2006). Transparent Process Migration: Design Alternatives and the Sprite Implementation. In *Software: Practice and Experience.*

[Easwaran et al., 2007] Easwaran, A., Anand, M., and Lee, I. (2007). Compositional Analysis Framework using EDP Resource Models. In *Proc. of the 28th Real-Time Systems Symposium.*

[Easwaran et al., 2006] Easwaran, A., Shin, I., Sokolsky, O., and Lee, I. (2006). Incremental Schedulability Analysis of Hierarchical Real-Time Components. In *Proc. of the 6th ACM Conference on Embedded Software*.

[Ebeling, 1997] Ebeling, C. (1997). *An Introduction to Reliability and Maintainability Engineering*. Waveland Press.

[Emberson et al., 2010] Emberson, P., Stafford, R., and Davis, R. (2010). Techniques for the Synthesis of Multiprocessor Tasksets. In *Proc. of the Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*.

[Ernst et al., 2002] Ernst, M. D., Badros, G. J., and Notkin, D. (2002). An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170.

[European Aviation Safety Agency, 2012] European Aviation Safety Agency (2012). Certification Memorandum - Software Aspects of Certification.

[Feld, 2004] Feld, J. (2004). PROFINET - Scalable Factory Communication for all Applications. In *Proc. of the IEEE International Workshop on Factory Communication Systems*.

[Feng and Mok, 2002] Feng, X. and Mok, A. (2002). A model of Hierarchical Real-Time Virtual Resources. In *Proc. of the 23rd IEEE Real-Time Systems Symposium*.

[Fidge, 1998] Fidge, C. J. (1998). Real-Time Schedulability Tests for Preemptive Multitasking. *Real-Time Systems*, 14:61–93.

[Filyner, 2003] Filyner, B. (2003). Open Systems Avionics Architectures Considerations. *Aerospace and Electronic Systems Magazine, IEEE*, 18(9):3–10.

[Fisher-Ogden, 2006] Fisher-Ogden, J. (2006). Hardware Support for Efficient Virtualization. Technical report, University of California, San Diego.

[Fohler, 1993] Fohler, G. (1993). Changing Operational Modes in the Context of Pre Run-time Scheduling. *IEIC Transactions on Information and Systems - Special Issue on Responsive Computer Systems*, pages 1333–40.

[Freedman et al., 1996] Freedman, P., Gaudreau, D., Boutaba, R., and Mehaoua, A. (1996). The Two Real-Time Solitudes: Computerized Control and Telecommunications. In *Proc. of the 2nd IEEE International Conference on Engineering of Complex Computer Systems*.

[Funk et al., 2001] Funk, S., Goossens, J., and Baruah, S. (2001). On-line Scheduling on Uniform Multiprocessors. In *Proc. of the 22nd IEEE Real-Time Systems Symposium*, pages 183–192.

[Fürst et al., 2009] Fürst, S., Mössinger, J., Bunzel, S., Weber, T., Kirschke-Biller, F., Heitkämper, P., Kinkelin, G., Nishikawa, K., and Lange, K. (2009). AUTOSAR–A Worldwide Standard is on the Road. In *Proc. of the 14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden.*

[Garey and Johnson, 1979] Garey, M. and Johnson, D. (1979). *Computers and Intractability.* W.H. Freman, New York.

[Gausemeier et al., 2014] Gausemeier, J., Rammig, F., Schäfer, W., and Sextro, W., editors (2014). *Dependability of Self-optimizing Mechatronic Systems.* Lecture Notes in Mechanical Engineering. Springer.

[Geisler, 2014] Geisler, J. (2014). *Selbstoptimierende Spurführung für ein neuartiges Schienenfahrzeug.* PhD thesis, University of Paderborn.

[Ghaisas et al., 2010] Ghaisas, S., Karmakar, G., Shenai, D., Tirodkar, S., and Ramamritham, K. (2010). SParK: Safety Partition Kernel for Integrated Real-Time Systems. In *Lecture Notes in Computer Science*, volume 6462, pages 159–174.

[Giannopoulou et al., 2014] Giannopoulou, G., Stoimenov, N., Huang, P., and Thiele, L. (2014). Mapping Mixed-Criticality Applications on Multi-Core Architectures. In *Proc. of Design, Automation & Test in Europe.*

[Gilles, 2012] Gilles, K. (2012). A Multi-Core Hypervisor for Embedded Real-Time Systems. Master's thesis, University of Paderborn.

[Gilles et al., 2013] Gilles, K., Groesbrink, S., Baldin, D., and Kerstan, T. (2013). Proteus Hypervisor: Full Virtualization and Paravirtualization for Multi-Core Embedded Systems. In *Proc. International Embedded Systems Symposium*, pages 293–305.

[Goyal et al., 1996] Goyal, P., Guo, X., and Vin, H. (1996). A hierarchical cpu scheduler for multimedia operating systems. In *Proc. of the Usenix Association 2nd Symposium on Operating Systems Design and Implementation.*

[Groesbrink, 2010] Groesbrink, S. (2010). Comparison of alternative hierarchical scheduling techniques for the virtualization of embedded real-time systems. Master's thesis, University of Paderborn.

[Groesbrink, 2014] Groesbrink, S. (2014). Virtual Machine Migration as a Fault Tolerance Technique for Embedded Real-Time Systems. In *Proc. of the 8th IEEE International Conference on Software Security and Reliability*.

[Groesbrink and Almeida, 2014] Groesbrink, S. and Almeida, L. (2014). A Criticality-aware Mapping of Real-time Virtual Machines to Multi-core Processors. In *Proc. of the 19th IEEE International Conference on Emerging Technologies and Factory Automation*.

[Groesbrink et al., 2014a] Groesbrink, S., Almeida, L., de Sousa, M., and Petters, S. (2014a). Towards Certifiable Adaptive Reservations for Hypervisor-based Virtualization. In *Proc. of the 20th Real-Time and Embedded Technology and Applications Symposium*.

[Groesbrink et al., 2014b] Groesbrink, S., Korrapati, S., Schmitz, A., and Schreckenberg, A. (2014b). Hypervisor-based Consolidation for Automated Teller Machines. In *Proc. of the Embedded World Conference*.

[Gu and Zhao, 2012] Gu, Z. and Zhao, Q. (2012). A State-of-the-Art Survey on Real-Time Issues in Embedded Systems Virtualization. *Journal of Software Engineering and Applications*, 5(4):277–290.

[Guan et al., 2011] Guan, N., Ekberg, P., Stigge, M., and Yi, W. (2011). Effective and Efficient Scheduling of Certifiable Mixed-Criticality Sporadic Task Systems. In *32nd IEEE Real-Time Systems Symposium*.

[Gupta et al., 2006] Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A. (2006). Enforcing Performance Isolation Across Virtual Machines in XEN. In *Proc. of the ACM/IFIP/USENIX 2006 International Conference on Middleware*.

[Gut et al., 2012] Gut, G., Allmann, C., Schurius, M., and Schmidt, K. (2012). Reduction of Electronic Control Units in Electric Vehicles Using Multicore Technology. In Pankratius, V. and Philippsen, M., editors, *Multicore Software Engineering, Performance, and Tools*, volume 7303 of *Lecture Notes in Computer Science*, pages 90–93. Springer Berlin Heidelberg.

[Hansen and Jul, 2004] Hansen, J. and Jul, E. (2004). Self-migration of Operating Systems. In *Proc. of the Workshop on ACM SIGOPS European Workshop*.

[Harbour and Palencia, 2003] Harbour, M. G. and Palencia, J. (2003). Response Time Analysis for Tasks Scheduled under EDF within Fixed Priorities. In *Proc. of the Real-Time Systems Symposium*.

[Hasan et al., 2013] Hasan, S. F., Siddique, N., and Chakraborty, S. (2013). *Intelligent Transport Systems*. Springer, New York, USA.

[Heiser, 2007] Heiser, G. (2007). Virtualization for Embedded Systems - Technology White Paper (Document No. OK 40036:2007). www.ok-labs.com.

[Heiser, 2009] Heiser, G. (2009). Hypervisors for consumer electronics. In *Proc. of the 6th IEEE Consumer Communications and Networking Conference*.

[Heiser and Leslie, 2010] Heiser, G. and Leslie, B. (2010). The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proc. of the ACM Asia-Pacific Workshop on Systems*.

[Henke et al., 2008] Henke, C., Tichy, M., Schneider, T., J, J. B., and Schäfer, W. (2008). System Architecture and Risk Management for Autonomous Railway Convoys. In *Proc. of the International Systems Conference*.

[Hergenhan and Heiser, 2008] Hergenhan, A. and Heiser, G. (2008). Operating Systems Technology for Converged ECUs. In *Proc. of the 6th Embedded Security in Cars Conference*.

[Herman et al., 2012] Herman, J., Kenna, C., Mollison, M., Anderson, J., and Johnson, D. (2012). RTOS Support for Multicore Mixed-Criticality Systems. In *Proc. of the Real-Time Technology and Applications Symposium*, pages 197–208.

[Hestermeyer et al., 2004] Hestermeyer, T., Oberschelp, O., and Giese, H. (2004). Structured Information Processing For Self-optimizing Mechatronic Systems. In *Proc. of the International Conference on Informatics in Control, Automation and Robotics*.

[Hines and Gopalan, 2009] Hines, M. and Gopalan, K. (2009). Post-copy Based Live Virtual Machine Migration Using Adaptive Pre-paging and Dynamic Self-ballooning. In *Proc. of the ACM Conference on Virtual Execution Environments*.

[Horn, 1974] Horn, W. (1974). Some Simple Scheduling Algorithms. In *Naval Research Logistics Quarterly*, volume 21, pages 177—185.

[Horner, 1989] Horner, D. R. (1989). *Operating Systems: Concepts and Applications*. Scott, Foresman and Company, Glenview, IL, USA.

[Huang et al., 2012] Huang, H., Gill, C., and Chenyang, L. (2012). Implementation and Evaluation of Mixed-Criticality Scheduling Approaches for Periodic Tasks.

In *Proc. of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium*.

[IBM, 2005] IBM (2005). PPC405Fx Embedded Processor Core User's Manual. Technical report, International Business Machines Corporation.

[IBM, 2006] IBM (2006). Power isa version 2.03. Technical report, International Business Machines Corporation.

[IBM, 2010] IBM (2010). PowerPC ISA 2.06 Revision B. Technical report, International Business Machines Corporation.

[IBM Research, 2012] IBM Research (2012). IBM PowerPC 4XX Instruction Set Simulator (ISS).

[IEC, 2010] IEC (2010). IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems. Technical report, International Electrotechnical Commission.

[IEEE, 1990] IEEE (1990). *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries.* Institute of Electrical and Electronics Engineers, New York, USA.

[Inam et al., 2011] Inam, R., Maeki-Turja, J., Sjoedin, M., Ashjaei, S. M. H., and Afshar, S. (2011). Support for Hierarchical Scheduling in FreeRTOS. In *Proc. of the 16th IEEE Conference on Emerging Technologies and Factory Automation.*

[ISO, 2011] ISO (2011). ISO 26262: Road vehicles — Functional safety. Technical report, International Organization for Standardization.

[Kaiser, 2008] Kaiser, R. (2008). Alternatives for Scheduling Virtual Machines in Real-time Embedded Systems. In *Proc. of the 1st Workshop on Isolation and Integration in Embedded Systems*, pages 5–10.

[Kalogeraki et al., 2008] Kalogeraki, V., Melliar-Smith, P., Moser, L., and Drougas, Y. (2008). Resource Management Using Multiple Feedback Loops in Soft Real-Time Distributed Object Systems. *Journal of Systems and Software*, 81(7):1144–1162.

[Keckler et al., 2009] Keckler, S. W., Olukotun, K., and Hofstee, H. P., editors (2009). *Multicore Processors and Systems.* Springer.

[Kelly et al., 2011] Kelly, O., Aydin, H., and Zhao, B. (2011). On Partitioned Scheduling of Fixed-Priority Mixed-Criticality Task Sets. In *Proc. of the Conference on Trust, Security and Privacy in Computing and Communications*.

[Kernighan and Ritchie, 1988] Kernighan, B. and Ritchie, D. (1988). *The C Programming Language*. Prentice Hall.

[Kerstan, 2011] Kerstan, T. (2011). *Towards Full Virtualization of Embedded Real-Time Systems*. PhD thesis, University of Paderborn.

[Khalilzad et al., 2012] Khalilzad, N., Behnam, M., Spampinato, G., and Nolte, T. (2012). Bandwidth Adaption in Hierarchical Scheduling Uzing Fuzzy Controllers. In *Proc. of the Symposium on Industrial Embedded Systems*, pages 148–157.

[Kim et al., 2009] Kim, D. S., Machida, F., and Trivedi, K. (2009). Availability Modeling and Analysis of a Virtualized System. In *Proc. of the IEEE Pacific Rim International Symposium on Dependable Computing*.

[King et al., 2003] King, S., Dunlap, G., and Chen, P. (2003). Operating System Support for Virtual Machines. In *Proc. of the USENIX Annual Technical Conference*.

[Kiszka, 2011] Kiszka, J. (2011). Towards Linux as a Real-Time Hypervisor. In *Proc. of the Real Time Linux Workshop*.

[Kivity et al., 2007] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. (2007). KVM: the Linux Virtual Machine Monitor. In *Proc. of the Linux Symposium*.

[Kopetz, 1997] Kopetz, H. (1997). *Real-time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic.

[Kotaba et al., 2013] Kotaba, O., Nowotsch, J., Paulitsch, M., Petters, S. M., and Theiling, H. (2013). Multicore in Real-Time Systems–Temporal Isolation Challenges due to Shared Resources. In *Proc. of the International Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems*.

[Kozuch and Satyanarayanan, 2002] Kozuch, M. and Satyanarayanan, M. (2002). Internet suspend/resume. In *Proc. of the IEEE Workshop on Mobile Computing Systems and Applications*.

[Kumar et al., 2007] Kumar, S., Raj, H., Schwan, K., and Ganev, I. (2007). Re-architecting VMMs for Multicore Systems: The Sidecore Approach. In *Proc. of the Workshop on Interaction between Operating Systems and Computer Architecture.*

[Kuo and Mok, 1991] Kuo, T. and Mok, A. K. (1991). Load Adjustment in Adaptive Real-Time Systems. In *Proc. of the IEEE Real-Time Systems Symposium.*

[Kuo and Li, 1998] Kuo, T.-W. and Li, C.-H. (1998). A Fixed Priority Driven Open Environment for Real-Time Applications. In *Proc. of the IEEE Real-Time Systems Symposium.*

[Kwak et al., 2001] Kwak, S., Choi, B., and Kim, B. (2001). An Optimal Checkpointing-Strategy for Real-Time Control Systems Under Transient Faults. *IEEE Transactions on Reliability*, 50(3).

[Lackorzynski et al., 2012] Lackorzynski, A., Warg, A., Völp, M., and Härtig, H. (2012). Flattening Hierarchical Scheduling. In *Proc. of the 10th ACM International Conference on Embedded Software*, pages 93–102.

[Lamport, 1974] Lamport, L. (1974). A new solution of Dijkstra's concurrent programming problem. *Communumications of the ACM*, 17:453–455.

[Land and Doig, 1960] Land, A. and Doig, A. (1960). An Automatic Method of Solving Discrete Programming Problems. *Econometrica*, 28(3):497–520.

[Lauzac et al., 1998] Lauzac, S., Melhem, R., and Mosse, D. (1998). An Efficient RMS Admission Control and its Application to Multiprocessor Scheduling. In *Proc. of the Symposium on Parallel Processing.*

[Lee et al., 1996] Lee, C., Rajkumar, R., and Mercer, C. (1996). Experiences with Processor Reservation and Dynamic QOS in Real-time Mach. In *Proc. of the Multimedia Japan.*

[Lee and Seshia, 2011] Lee, E. and Seshia, S. (2011). *Introduction to Embedded Systems - A Cyber-Physical Systems Approach.* http://LeeSeshia.org.

[Lee et al., 2011] Lee, J., Xi, S., Chen, S., Phan, L. T. X., Gill, C., Lee, I., Lu, C., and Sokolsky, O. (2011). Realizing Compositional Scheduling Through Virtualization. In *Proc. of the Real-Time and Embedded Technology and Applications Symposium*, pages 13–22.

[Lee et al., 2010] Lee, M., Krishnakumar, A., Krishnan, P., Singh, N., and Yajnik, S. (2010). Supporting Soft Real-Time Tasks in the Xen Hypervisor. In *Proc. of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*.

[Lehoczky et al., 1989] Lehoczky, J., Sha, L., and Ding, Y. (1989). The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proc. of the Real-Time Systems Symposium*.

[Lehoczky et al., 1987a] Lehoczky, J., Sha, L., and Strosnider, J. (1987a). Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *Proc. of the IEEE Real-Time Systems Symposium*.

[Lehoczky et al., 1987b] Lehoczky, J. P., Sha, L., and Ding, Y. (1987b). The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. Department of Statistics, Carnegie Mellon University.

[Leontyev and Anderson, 2009] Leontyev, H. and Anderson, J. (2009). A Hierarchical Multiprocessor Bandwidth Reservation Scheme with Timing Guarantees. *Real-Time Systems*, 43:60–92.

[Leung and Whitehead, 1982] Leung, J. and Whitehead, J. (1982). On the Complexity of Fixed Priority Scheduling of Periodic Real-Time Tasks. *Performance Evaluation*, 2(4).

[Leung and Zhao, 2005] Leung, J. and Zhao, H. (2005). Real-Time Scheduling Analysis. Technical report, U.S. Department of Transportation – Federal Aviation Administration.

[LeVasseur et al., 2008] LeVasseur, J., Uhlig, V., Chapman, M., Chubb, P., Leslie, B., and Heiser, G. (2008). Pre-virtualization: Soft Layering for Virtual Machines. In *Proc. of the 13th Asia-Pacific Computer Systems Architecture Conference*.

[Leveson, 1995] Leveson, N. (1995). *Safeware, System Safety and Computers*. Addison-Wesley, Boston.

[Li and Baruah, 2010] Li, H. and Baruah, S. (2010). An Algorithm for Scheduling Certifiable Mixed-Criticality Sporadic Task Systems. In *Proc. of the 31st IEEE Real-Time Systems Symposium*.

[Li et al., 2012a] Li, N., Kinebuchi, Y., Mitake, H., Shimada, H., Lin, T., and Nakajima, T. (2012a). A Light-Weighted Virtualization Layer for Multicore Processor-Based Rich Functional Embedded Systems. In *Proc. of the 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*.

[Li et al., 2012b] Li, N., Kinebuchi, Y., Mitake, H., Shimada, H., Lin, T.-H., and Nakajima, T. (2012b). A Light-Weighted Virtualization Layer for Multicore Processor-Based Rich Functional Embedded Systems. In *Proc. of the 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*.

[Liebetrau et al., 2012] Liebetrau, T., Kelling, U., Otter, T., and Hell, M. (2012). Energy Saving in Automotive E/E Architectures. Technical report, Infineon Technologies, www.infineon.com.

[Liebig et al., 2011] Liebig, J., Kästner, C., and Apel, S. (2011). Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proc. of the 10th International Conference on Aspect-oriented Software Development*.

[Lin et al., 2009] Lin, J., Lu, Q., Ding, X., Zhang, Z., Zhang, X., and Sadayappan, P. (2009). Enabling Software Management for Multicore Caches with a Lightweight Hardware Support. In *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pages 1–12.

[Lin et al., 2013] Lin, T., Mitake, H., and Nakajima, T. (2013). Improving GPOS Real-time Responsiveness using vCPU Migration in an Embedded Multicore Virtualization Platform. In *Proc. of the 16th IEEE International Conference on Computational Science and Engineering*.

[Lin et al., 2010] Lin, Y.-C., Yang, C.-Y., Chang, C.-W., Chang, Y.-H., Kuo, T.-W., and Shih, C.-S. (2010). Energy-Efficient Mapping Techniques for Virtual Cores. In *Proc. of the Euromicro Conference on Real-Time Systems*.

[Lipari and Baruah, 2000] Lipari, G. and Baruah, S. (2000). Greedy Reclamation of Unused Bandwidth in Constant Bandwidth Servers. In *Proc. of the Euromicro Conference on Real-Time Systems*, pages 193–200.

[Lipari and Baruah, 2001] Lipari, G. and Baruah, S. (2001). A Hierarchical Extension to the Constant Bandwidth Server Framework. In *Proc. of the 7th IEEE Real-Time Technology and Applications Symposium*.

[Lipari and Bini, 2003] Lipari, G. and Bini, E. (2003). Resource Partitioning Among Real-Time Applications. In *Proc. of the Euromicro Conference on Real-Time Systems*, pages 151–158.

[Lipari et al., 2000] Lipari, G., Carpenter, J., and Baruah, S. (2000). A Framework for Achieving Inter-Application Isolation in Multiprogrammed, Hard Real-time Environments. In *Proc. of the 21st Real-Time Systems Symposium*.

[Littlefield-Lawwill and Kinnan, 2008] Littlefield-Lawwill, J. and Kinnan, L. (2008). System Considerations For Robust Time And Space Partitioning In Integrated Modular Avionics. In *Proc. of the Digital Avionics Systems Conference*.

[Littlefield-Lawwill and Ramanathan, 2007] Littlefield-Lawwill, J. and Ramanathan, V. (2007). Advancing Open Standards In Integrated Modular Avionics: An Industry Analysis. In *Proc. of the 26th Digital Avionics Systems Conference*.

[Liu, 1969] Liu, C. (1969). Scheduling Algorithms for Multiprocessors in a Hard Real-Time Environment. In *JPL Space Programs Summary*, volume 37-60, pages 28–31.

[Liu and Layland, 1973a] Liu, C. and Layland, J. (1973a). Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Envionment. In *Journal of the Association for Computing Machinery*, volume 20(1).

[Liu and Layland, 1973b] Liu, C. and Layland, J. (1973b). Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20:44–61.

[Liu et al., 2006] Liu, J., Huang, W., Abali, B., and Panda, D. (2006). High Performance VMM-Bypass I/O in Virtual Machine. In *Proc. of the USENIX Annual Technical Conference*.

[Liu, 2000] Liu, J. W. S. (2000). *Real-Time Systems*. Prentice Hall.

[Liu et al., 2012] Liu, L., Cui, Z., Xing, M., Bao, Y., Chen, M., and Wu, C. (2012). A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems. In *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 367–376.

[Lopez et al., 2000] Lopez, J., Garcia, M., Diaz, J., and Garcia, D. (2000). Worst-case Utilization Bound for EDF Scheduling on Real-Time Multiprocessor Systems. In *Proc. of the 12th Euromicro Conference on Real-Time Systems.*

[Lopez et al., 2003] Lopez, J., Garcia, M., Diaz, J., and Garcia, D. (2003). Utilization bounds for Multiprocessor Rate-Monotonic Systems. *Real-Time Systems.*

[Lorente et al., 2006] Lorente, J., Lipari, G., and Bini, E. (2006). A Hierarchical Scheduling Model for Component-based Real-time Systems. In *Proc. of the 20th Parallel and Distributed Processing Symposium.*

[Lückel et al., 2008] Lückel, J., Grotstollen, H., Henke, M., Hestermeyer, T., and Liu-Henke, X. (2008). RailCab System: Engineering Aspects. *Dynamical Analysis of Vehicle Systems*, 497:237–281.

[Ma et al., 2013] Ma, R., Zhou, F., Zhu, E., and Guan, H. (2013). Performance Tuning Towards a KVM-based Embedded Real-Time Virtualization System. *Journal of Information Science and Engineering*, 29:1021–1035.

[Maggio et al., 2013] Maggio, M., Bini, E., Chasparis, G., and Arzen, K.-E. (2013). A Game-Theoretic Resource Manager for RT Applications. In *Proc. of the Euromicro Conference on Real-Time Systems*, pages 57–66.

[Marau et al., 2011] Marau, R., Lakshmanan, K., Pedreiras, P., Almeida, L., and Rajkumar, R. (2011). Efficient Elastic Resource Management for Dynamic Embedded Systems. In *Proc. of the Conference on Trust, Security and Privacy in Computing and Communications.*

[Marzario et al., 2004] Marzario, L., Lipari, G., Balbastre, P., and Crespo, A. (2004). IRIS: A New Reclaiming Algorithm for Server-based Real-time Systems. In *Proc. of the Real-time and Embedded Technology and Applications Symposium*, pages 211–218.

[Masmano et al., 2009] Masmano, M., Ripoll, I., and Crespo, A. (2009). XtratuM: a Hypervisor for Safety Critical Embedded Systems. In *Proc. of the 11th Real-Time Linux Workshop.*

[Masrur et al., 2010] Masrur, A., Drossler, S., Pfeuffer, T., and Chakraborty, S. (2010). VM-Based Real-Time Services for Automotive Control Applications. In *Proc. of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications.*

[Masrur et al., 2011] Masrur, A., Pfeuffer, T., Geier, M., Drössler, S., and Chakraborty, S. (2011). Designing VM Schedulers for Embedded Real-time Applications. In *Proc. of the 7th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 29–38.

[Matic and Henzinger, 2005] Matic, S. and Henzinger, T. (2005). Trading End-to-End Latency for Composability. In *Proc. of the IEEE Real-Time Systems Symposium.*

[Melo et al., 2013] Melo, M., Maciel, P., Araujo, J., Matos, R., and Araujo, C. (2013). Availability Study on Cloud Computing Environments: Live Migration as a Rejuvenation Mechanism. In *Proc. of the International Conference on Dependable Systems and Networks*, pages 1–6.

[Milojičić et al., 2000] Milojičić, D., Douglis, F., Paindaveine, Y., Wheeler, R., and Zhou, S. (2000). Process Migration. *ACM Computing Surveys*, 32(3):241–299.

[Mok, 1983] Mok, A. (1983). *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment.* PhD thesis, Massachusetts Institute of Technology.

[Mok et al., 2001] Mok, A., Feng, X., and Chen, D. (2001). Resource Partition for Real-Time Systems. In *Proc. of the Real-Time Technology and Applications Symposium.*

[Mollison et al., 2010] Mollison, M., Erickson, J., Anderson, J., Baruah, S., and Scoredos, J. (2010). Mixed-Criticality Real-Time Scheduling for Multicore Systems. In *Proc. of the International Conference on Computer and Information Technology.*

[Moyer, 2013] Moyer, B. (2013). *Real World Multicore Embedded Systems.* Newnes.

[Nagarajan et al., 2007] Nagarajan, A., Mueller, F., Engelmann, C., and Scott, S. (2007). Proactive Fault Tolerance for HPC with Xen Virtualization. In *Proc. of the International Conference on Supercomputing.*

[Nakajima et al., 2011] Nakajima, T., Kinebuchi, Y., Shimada, H., Courbot, A., and Lin, T. (2011). Temporal and Spatial Isolation in a Virtualization Layer for Multi-core Processor Based Information Appliances. In *Proc. of the 16th Asia and South Pacific Design Automation Conference.*

[Nanda and Chiueh, 2005] Nanda, S. and Chiueh, T. (2005). A Survey on Virtualization Technologies. Technical report, SUNY Stony Brook, Department of Computer Science.

[Natale and Sangiovanni-Vincentelli, 2010] Natale, M. D. and Sangiovanni-Vincentelli, A. (2010). Moving From Federated to Integrated Architectures in Automotive: The Role of Standards, Methods and Tools. *Proceedings of the IEEE*, 98(4):603–620.

[Navet et al., 2010] Navet, N., Monot, A., Bavoux, B., and Simonot-Lion, F. (2010). Multi-source and Multicore Automotive ECUs - OS Protection Mechanisms and Scheduling. In *Proc. of the International Symposium on Industrial Electronics*, pages 3734–3741.

[Negrean et al., 2009] Negrean, M., Schliecker, S., and Ernst, R. (2009). Response-time Analysis of Arbitrarily Activated Tasks in Multiprocessor Systems with Shared Resources. In *Proc. of the Design, Automation Test in Europe Conference*, pages 524–529.

[Nelson et al., 2005] Nelson, M., Lim, B., and Hutchins, G. (2005). Fast Transparent Migration for Virtual Machines. In *Proc. of the USENIX Annual Technical Conference*.

[Nogueira and Pinho, 2007] Nogueira, L. and Pinho, L. (2007). Capacity Sharing and Stealing in Dynamic Server-based Real-Time Systems. In *Proc. of the Parallel and Distributed Processing Symposium*, pages 1–8.

[Nowotsch and Paulitsch, 2012] Nowotsch, J. and Paulitsch, M. (2012). Leveraging multi-core computing architectures in avionics. In *Proc. of the 9th European Dependable Computing Conference*, pages 132–143.

[Nutt, 2000] Nutt, G. (2000). *Operating Systems: A Modern Perspective.* Addison-Wesley.

[Obermaisser et al., 2009] Obermaisser, R., El Salloum, C., Huber, B., and Kopetz, H. (2009). From a Federated to an Integrated Automotive Architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):956–965.

[Oberthür et al., 2010] Oberthür, S., Zaramba, L., and Lichte, H. (2010). Flexible Resource Management for Self-X Systems: An Evaluation. In *Proc. of the Workshop on Self-Organizing Real-Time Systems.*

[Oikawa et al., 2006] Oikawa, S., Ito, M., and Nakajima, T. (2006). Linux/RTOS Hybrid Operating Environment on Gandalf Virtual Machine Monitor. *Lecture Notes in Computer Science*, 4096:287–296.

[Olderog and Dierks, 2008] Olderog, E. and Dierks, H. (2008). *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press.

[Ongaro et al., 2008] Ongaro, D., Cox, A. L., and Rixner, S. (2008). Scheduling I/O in Virtual Machine Monitors. In *Proc. of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 1–10.

[Orsila et al., 2007] Orsila, H., Kangas, T., Salminen, E., Hämäläinen, T., and Hännikäinen, M. (2007). Automated Memory-aware Application Distribution for Multi-processor System-on-Chips. *Journal of Systems Architecture*, pages 795–815.

[Paolieri et al., 2009] Paolieri, M., Quiñones, E., Cazorla, F. J., Bernat, G., and Valero, M. (2009). Hardware Support for WCET Analysis of Hard Real-time Multicore Systems. *SIGARCH Comput. Archit. News*, 37(3):57–68.

[Papadopoulos et al., 2010] Papadopoulos, Y., Walker, M., Reiser, M.-O., Weber, M., Chen, D.-J., Törngren, M., Servat, D., Abele, A., Stappert, F., Lönn, H., Berntsson, L., Johansson, R., Tagliabo, F., Torchiaro, S., and Sandberg, A. (2010). Automatic Allocation of Safety Integrity Levels. In Fabre, J.-C., Guetta, O., and Trapp, M., editors, *Proc. of the 1st Workshop on Critical Automotive Applications: Robustness & Safety*, ACM International Conference Proceeding Series, pages 7–10. ACM.

[Paun et al., 2013] Paun, V.-A., Monsuez, B., and Baufreton, P. (2013). On the Determinism of Multi-core Processors. In Choppy, C. and Sun, J., editors, *Proc. of the 1st French Singaporean Workshop on Formal Methods and Applications*, volume 31 of *OpenAccess Series in Informatics (OASIcs)*, pages 32–46, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Peiró et al., 2010] Peiró, S., Crespo, A., Ripoll, I., and Masmano, M. (2010). Partitioned Embedded Architecture based on Hypervisor: the XtratuM Approach. In *Proc. of the European Dependable Computing Conference*, pages 67–72.

[Pelzl et al., 2008] Pelzl, J., Wolf, M., and T.Wollinger (2008). Virtualization Technologies for Cars – Solutions to Increase Safety and Security of Vehicular ECUs. In *Automotive – Safety & Security 2008*.

[Peng and Shin, 1997] Peng, D. and Shin, K. (1997). Assignment and Scheduling Communicating Periodic Tasks in Distributed Real-Time Systems. *IEEE Transactions on Software Engineering*.

[Petters et al., 2009] Petters, S., Lawitzy, M., Heffernan, R., and Elphinstone, K. (2009). Towards Real Multi-Criticality Scheduling. In *Proc. of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

[Phan et al., 2010] Phan, L., Lee, I., and Sokolsky, O. (2010). Compositional Analysis of Multi-Mode Systems. In *Proc. of the 22nd Euromicro Conference on Real-Time Systems*.

[Popek and Goldberg, 1974a] Popek, G. J. and Goldberg, R. P. (1974a). Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421.

[Popek and Goldberg, 1974b] Popek, G. J. and Goldberg, R. P. (1974b). Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421.

[Porrmann et al., 2009] Porrmann, M., Hagemeyer, J., Pohl, C., Romoth, J., and Strugholtz, M. (2009). RAPTOR: A Scalable Platform for Rapid Prototyping and FPGA-based Cluster Computing. In *Proc. of the International Conference on Parallel Computing*.

[Powell and Miller, 1983] Powell, M. and Miller, B. (1983). Process Migration in DEMOS/MP. In *Proc. of the ACM Symposium on Operating System Principles*.

[Prisaznuk, 2008a] Prisaznuk, P. (2008a). ARINC 653 Role in Integrated Modular Avionics (IMA). In *Proc. of the IEEE/AIAA 27th Digital Avionics Systems Conference*.

[Prisaznuk, 2008b] Prisaznuk, P. (2008b). ARINC 653 Role in Integrated Modular Avionics (IMA). In *Proc. of the 27th IEEEE Digital Avionics Systems Conference*.

[Pulido et al., 2006] Pulido, J., Uruena, S., Zamorano, J., Vardanega, T., and de la Puente, J. (2006). Hierarchical Scheduling with Ada 2005. In Pinho, L. and Harbour, M. G., editors, *Ada-Europe, Lecture Notes in Computer Science 4006*, pages 1 – 12. Springer.

[Punnekkat et al., 2001] Punnekkat, S., Burns, A., and Davis, R. (2001). Analysis of Checkpointing for Real-Time Systems. *The International Journal of Time-Critical Computing Systems*, 20:83–102.

[Ramasamy and Schunter, 2007] Ramasamy, H. and Schunter, M. (2007). Architecting Dependable Systems Using Virtualization. In *Proc. of the DSN Workshop on Architecting Dependable Systems.*

[Rasche and Polze, 2005] Rasche, A. and Polze, A. (2005). Dynamic Reconfiguration of Component-based Real-time Software. In *Proc. of the IEEE Workshop on Object-Oriented Real-Time Dependable Systems.*

[Real and Crespo, 2004] Real, J. and Crespo, A. (2004). Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal. *Real-Time Systems*, 26(2):161–197.

[Regehr and Stankovic, 2001] Regehr, J. and Stankovic, J. (2001). HLS: A Framework for Composing Soft Real-Time Schedulers. In *Proc. of the 22nd IEEE Real-Time Systems Symposium*, pages 3–14.

[Reinhardt and Kucera, 2013] Reinhardt, D. and Kucera, M. (2013). Domain Controlled Architecture – A New Approach for Large Scale Software Integrated Automotive Systems. In *Proc. of the 3rd International Conference on Pervasive and Embedded Computing and Communication Systems.*

[Robin and Irvine, 2000] Robin, J. and Irvine, C. (2000). Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proc. of the 9th USENIX Security Symposium.*

[Rosen et al., 2007] Rosen, J., Andrei, A., Eles, P., and Peng, Z. (2007). Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *Proc. of the 28th IEEE International Real-Time Systems Symposium*, pages 49–60.

[Rostedt, 2007] Rostedt, S. (2007). Internals of the RT Patch. In *Proc. of the Linux Symposium.*

[Roy et al., 2013] Roy, A., Ganesan, R., Dash, D., and Sarkar, S. (2013). Reducing Service Failures by Failure and Workload aware Load Balancing in SaaS Clouds. In *Proc. of the IEEE/IFIP Dependable Systems and Networks Workshop.*

[RTCA/DO, 2012] RTCA/DO (2012). DO-178C: Software Considerations in Airborne Systems and Equipment Certification. Technical report, Radio Technical Commission for Aeronautics.

[Rushby, 1981] Rushby, J. (1981). Design and Verification of Secure Systems. In *Proc. of the 8th ACM Symposium on Operating System Principles*.

[Rushby, 1999] Rushby, J. (1999). Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance. Technical Report NAS1-20334, FAA Technical Center and NASA Langley Research Center.

[Saewong et al., 2002] Saewong, S., Rajkumar, R., Lehoczky, L., and Klein, M. (2002). Analysis of Hierarchical Fixed-priority Scheduling. In *Proc. of the 14th IEEE Euromicro Conference on Real-Time Systems*.

[Sangorrin et al., 2012] Sangorrin, D., Honda, S., and Takada, H. (2012). Integrated Scheduling for a Reliable Dual-OS Monitor. *IPSJ Transactions on Advanced Computing Systems*, 5(2):99–110.

[Santos et al., 2011] Santos, R., Behnam, M., Nolte, T., Pedreiras, P., and Almeida, L. (2011). Multi-level Hierarchical Scheduling in Ethernet Switches. In *Proc. of the International Conference on Embedded Software*, pages 185–194.

[Santos et al., 2012] Santos, R., Lipari, G., Bini, E., and Cucinotta, T. (2012). Online Schedulability Tests for Adaptive Reservations in Fixed Priority Scheduling. *Real-Time Systems*, 48:601–634.

[Sapuntzakis et al., 2002] Sapuntzakis, C., Chandra, R., Pfaff, B., Chow, J., Lam, M., and Rosenblum, M. (2002). Optimizing the Migration of Virtual Computers. In *Proc. of the ACM Symposium on Operating Systems Design and Implementation*.

[Schoeberl, 2009] Schoeberl, M. (2009). Time-predictable Cache Organization. In *Software Technologies for Future Dependable Distributed Systems*, pages 11–16.

[Sha, 2004] Sha, L. (2004). Real-Time Virtual Machines for Avionics Software Porting and Development. In *Real-Time and Embedded Computing Systems and Applications*, pages 123–135. Springer Berlin Heidelberg.

[Sha et al., 1986] Sha, L., Lehoczky, J. P., and Rajkumar, R. (1986). Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. In *Proc. of the Real-Time Systems Symposium*, pages 181–191.

[Shin et al., 2008a] Shin, I., Behnam, M., Nolte, T., and Nolin, M. (2008a). Synthesis of Optimal Interfaces for Hierarchical Scheduling with Resources. In *Proc. of the IEEE Real-Time Systems Symposium.*

[Shin et al., 2008b] Shin, I., Easwaran, A., and Lee, I. (2008b). Hierarchical Scheduling Framework for Virtual Clustering Multiprocessors. In *Proc. of the 20th Euromicro Conference on Real-Time Systems.*

[Shin and Lee, 2003] Shin, I. and Lee, I. (2003). Periodic Resource Model for Compositional Real-Time Guarantees. In *Proc. of the 24th IEEE International Real-Time Systems Symposium.*

[Shin and Lee, 2004] Shin, I. and Lee, I. (2004). Compositional Real-Time Scheduling Framework. In *Proc. of the 25th IEEE Real-Time Systems Symposium.*

[Shin and Lee, 2008] Shin, I. and Lee, I. (2008). Compositional Real-Time Scheduling Framework with Periodic Model. *ACM Transactions on Embedded Computing Systems*, 7(3):30:1–30:39.

[Singh et al., 2013] Singh, A., Shafique, M., Kumar, A., and Henkel, J. (2013). Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends. In *Proc. of the Design Automation Conference.*

[Sites et al., 1993] Sites, R. L., Chernoff, A., Kirk, M. B., Marks, M. P., and Robinson, S. G. (1993). Binary Translation. *Commun. ACM*, 36(2):69–81.

[Smith and Nair, 2005a] Smith, J. and Nair, R. (2005a). *Virtual Machines.* Morgan Kaufmann.

[Smith and Nair, 2005b] Smith, J. E. and Nair, R. (2005b). The architecture of virtual machines. In *IEEE Computer*, volume 38(5), pages 32–38.

[Smith and Nair, 2005c] Smith, J. E. and Nair, R. (2005c). *The Architecture of Virtual Machines.* IEEE Computer.

[Sondermann-Wolke and Sextro, 2009] Sondermann-Wolke, C. and Sextro, W. (2009). Towards the Integration of Condition Monitoring in Self-Optimizing Function Modules. In *Proc. of Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns.*

[Stallings, 2005] Stallings, W. (2005). *Operating Systems: Internals and Design Principles.* Pearson Prentice Hall.

[Stankovic, 1988] Stankovic, J. (1988). Misconceptions about Real-Time Computing. *IEEE Transactions on Computers*, Oct.

[Stankovic and Ramamritham, 1989] Stankovic, J. and Ramamritham, K. (1989). The Spring Kernel: A New Paradigm for Real-time Operating Systems. *ACM Operating System Review*, 23(3):54–71.

[Stankovic et al., 1995] Stankovic, J., Spuri, M., Natale, M. D., and Buttazzo, G. (1995). Implications of Classical Scheduling Results for Real-Time Scheduling. In *IEEE Computer*, pages 16–25.

[Stankovic et al., 1998] Stankovic, J., Spuri, M., Ramaritham, K., and Buttazzo, G. (1998). *Deadline Scheduling for Real-Time Systems.* Kluwer Academic.

[Stappert et al., 2010] Stappert, F., Jonsson, J., Mottok, J., and Johansson, R. (2010). A Design Framework for End-To-End Timing Constrained Automotive Applications. *Proc. of the Conference on Embedded Real-Time Software and Systems.*

[Steinberg and Kauer, 2010] Steinberg, U. and Kauer, B. (2010). NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proc. of the European Conference on Computer systems.*

[Storey, 1996] Storey, N. (1996). *Safety-Critical Computer Systems.* Prentice Hall.

[Su et al., 2009] Su, D., Chen, W., Huang, W., Shan, H., and Jiang, Y. (2009). SmartVisor: Towards an Efficient and Compatible Virtualization Platform for Embedded System. In *Proc. of the 2nd Workshop on Isolation and Integration in Embedded Systems.*

[Su and Zhu, 2013] Su, H. and Zhu, D. (2013). An Elastic Mixed-Criticality Task Model and Its Scheduling Algorithm. In *Proc. of Design, Automation and Test in Europe*, pages 147–152.

[Tanenbaum and Goodman, 1998] Tanenbaum, A. S. and Goodman, J. R. (1998). *Structured Computer Organization.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition.

[Tanenbaum and Woodhull, 2006] Tanenbaum, A. S. and Woodhull, A. S. (2006). *Operating Systems: Design and Implementation.* Pearson Prentice Hall.

[Tavares et al., 2012] Tavares, A., Carvalho, A., Rodrigues, P., Garcia, P., Gomes, T., Cabral, J., Cardoso, P., Montenegro, S., and Ekpanyapong, M. (2012). A Customizable and ARINC 653 Quasi-compliant Hypervisor. In *Proc. of the IEEE International Conference on Industrial Technology.*

[Theimer et al., 1985] Theimer, M., Lantz, K., and Cheriton, D. (1985). Preemptable Remote Execution Facilities for the V-System. *SIGOPS Operating Systems Review*, 19(5):2–12.

[Thiebaut and Gerlach, 2012] Thiebaut, S. S. and Gerlach, M. (2012). Multicore and Virtualization in Automotive Environments. *EE Times europe automotive.*

[Uhlig et al., 2005] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C., Anderson, A. V., Bennett, S. M., Kagi, A., Leung, F. H., and L. Smith, . (2005). Intel Virtualization Technology. *Computer*, 38(5):48–56.

[Vera et al., 2003] Vera, X., Lisper, B., and Xue, J. (2003). Data Caches in Multitasking Hard Real-Time Systems. In *Proc. of the 24th IEEE Real-Time Systems Symposium*, pages 154–165.

[Vestal, 2007] Vestal, S. (2007). Preemptive Scheduling of Multi-Criticality Systems with Varying Degrees of Execution Time Assurance. In *Proc. of the Real-Time Systems Symposium.*

[Wang and Lin, 2000] Wang, Y.-C. and Lin, K.-J. (2000). The Implementation of Hierarchical Schedulers in the RED-Linux Scheduling Framework. In *Proc. of the 12th Euromicro Conference on Real-Time Systems*, pages 231–238.

[Watkins and Walter, 2007] Watkins, C. and Walter, R. (2007). Transitioning from Federated Avionics Architectures to Integrated Modular Avionics. In *Proc. of the 26th IEEE/AIAA Digital Avionics Systems Conference*, pages 2.A.1–1–2.A.1–10.

[Whitaker et al., 2002] Whitaker, A., Shaw, M., and Gribble, S. (2002). Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In *Proc. of the USENIX Annual Technical Conference.*

[Wilhelm et al., 2008] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. (2008). The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–47.

[Williston, 2009] Williston, K. (2009). Consolidating Hardware with Virtualization. In *Embedded Innovator*, volume Fall.

[Wilson and Preyssler, 2008] Wilson, A. and Preyssler, T. (2008). Incremental Certification and Integrated Modular Avionics. In *IEEE Digital Avionics Systems Conference*.

[Wirth, 1976] Wirth, N. (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall.

[Xi et al., 2011] Xi, S., Wilson, J., Lu, C., and Gill, C. (2011). RT-Xen: Towards Real-time Hypervisor Scheduling in Xen. In *Proc. of the International Conference on Embedded Software*.

[Xu and Parnas, 1993] Xu, J. and Parnas, D. (1993). On Satisfying Timing Constraints in Hard Real-Time Systems. In *IEEE Transactions on Software Engineering*, volume 19(1), pages 70–84.

[Yang et al., 2011] Yang, J., Kim, H., Park, S., Hong, C., and Shin, I. (2011). Implementation of Compositional Scheduling Framework on Virtualization. *ACM SIGBED Review*, 8(1):30–37.

[Yoo et al., 2008] Yoo, S., Liu, Y., Hong, C., Yoo, C., and Zhang, Y. (2008). MobiVMM: a Virtual Machine Monitor for Mobile Phones. In *Proc. of the First Workshop on Virtualization in Mobile Computing*.

[Yun et al., 2014] Yun, H., Mancuso, R., Wu, Z.-P., and Pellizzoni, R. (2014). PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. In *Proc. of the IEEE International Conference on Real-Time and Embedded Technology and Applications Symposium*.

[Zabos et al., 2009] Zabos, A., Davis, R. I., Burns, A., and Harbour, M. G. (2009). Spare Capacity Distribution Using Exact Response-time Analysis. In *Proc. of the International Conference on Real-time and Network Systems*, pages 97–106.

[Zhang and Burns, 2007] Zhang, F. and Burns, A. (2007). Analysis of Hierarchical EDF Pre-emptive Scheduling. In *Proc. of the 28th IEEE International Real-Time Systems Symposium*.

[Zhang et al., 2010] Zhang, J., Chen, K., Zuo, B., Ma, R., Dong, Y., and Guan, H. (2010). Performance Analysis Towards a KVM-Based Embedded Real-Time Virtu-

alization Architecture. In *Proc. of the 5th International Conference on Computer Sciences and Convergence Information Technology.*