ADVANCED INTEGRATION TECHNIQUES FOR HIGHLY RELIABLE DUAL-OS EMBEDDED SYSTEMS

DANIEL SANGORRIN LOPEZ

Title in Japanese

高信頼デュアルOS組込みシステムにおける統合技術。

Keywords

Reliability, Dual-OS systems, Virtualization, Embedded, Real-Time, ARM, TrustZone, Device sharing, Scheduling, Communications.

Preface

Except where indicated below, this dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration. This dissertation is not substantially the same as any that I have submitted for a degree or diploma or other qualification at any other University.

The work reported in § 2.5 (SafeG) was originally carried out at Nagoya University's center for embedded computing systems (NCES) before I started my research. During my time as a doctoral student I contributed to the maintenance, dissemination and extension of SafeG as a main developer.

On the subject of spelling, the author sides with the Oxford English Dictionary, who justify their consistent use of the termination -ize as opposed to -ise.

This dissertation contains 42 figures, 14 tables and approximately 32,819 words.

Daniel Sangorrín López
Embedded and Real-Time Systems Laboratory
Nagoya University (Japan)
27th July 2012

Acknowledgements

Associate Professor Shinya Honda has been an excellent supervisor, and I cannot thank him enough for his support, guidance, patience, and for being so generous with his time. His contributions both to this thesis and to my happiness at Nagoya these last few years are very gratefully acknowledged. I also thank my advisor Professor Hiroaki Takada for his time and support, and for giving me the opportunity to carry out my PhD in such a wonderful country.

Financial support from the Japanese Government (Monbukagakusho) scholarship is gratefully acknowledged.

I have met some wonderful people here in Nagoya. I am very grateful to my laboratory colleagues for their encouragement, and to my friends for making my time in Nagoya so happy and memorable.

This thesis is dedicated to my family, for loving me and supporting me in everything that I have wanted to do.

Abstract

This thesis considers dual-OS virtualization for consolidating a trusted real-time operating system (RTOS) and an untrusted general-purpose operating system (GPOS) onto the same hardware platform. Research on dual-OS systems is motivated by their smaller hardware cost—due to the fact that hardware is shared—and their ability to address the increasing complexity of modern embedded systems—by leveraging the GPOS advanced functionality—without affecting the timely behavior of the RTOS. The most fundamental requirement of a dual-OS system is guaranteeing the reliability and real-time performance of the RTOS against any misbehavior or malicious attack coming from the untrusted GPOS. For that reason, we use a dual-OS system (SafeG) that supports complete isolation of the memory and devices assigned to the RTOS; and gives higher priority to the execution of the RTOS. The SafeG dual-OS system is based on ARM TrustZone Security extensions, and its main component is the SafeG monitor, which is used to context-switch between both OSs.

Although the mere execution of the RTOS and the GPOS in isolation may satisfy the requirements of some systems, increasing the integration of the dual-OS system can lead to performance improvements, new collaborative applications with higher sophistication, and a further decrease of the hardware cost. The main three novel contributions to the reliable integration of a dual-OS system proposed in this thesis are: an integrated scheduling framework; efficient dual-OS communications; and repartition-based device sharing.

The integrated scheduling framework supports the interleaving of the execution priority levels of both OSs with high granularity, and uses execution-time reservations for guaranteeing the timeliness of the RTOS. The evaluation results show that the framework is suitable for enhancing the responsiveness of the GPOS time-sensitive activities without compromising the reliability and real-time performance of the RTOS.

Dual-OS communications allow RTOS and GPOS applications to collaborate in complex distributed applications. Traditional approaches are usually implemented by extending the virtualization layer with new communication primitives. We present a more efficient approach that minimizes the communication overhead caused by unnecessary copies and context switches; and satisfies the strict reliability requirements of the RTOS.

Finally, we consider mechanisms for sharing devices reliably in dual-OS systems. We note that previous approaches based on paravirtualization are not well suited to device sharing patterns where the GPOS share greatly exceeds that of the RTOS. For that reason, we propose two new approaches that are based on dynamically re-partitioning devices between the RTOS and the GPOS at runtime. The evaluation results show an interesting trade-off between overhead, functionality and device latency.

Nomenclature

- ARM: Advanced Risc Machine.
- ASP: Advanced Standard Profile.
- ATK: Automotive Kernel.
- CAD: Computer-Aided Design.
- CAM: Computer-Aided Manufacturing.
- CNC: Computer Numerical Control.
- CPSR: Current Program Status Register.
- CPU: Central Processing Unit.
- DMA: Direct Memory Access.
- Dualoscom: Dual-OS Communications.
- ECU: Electronic Computing Unit.
- FIQ: Fast Interrupt Request.
- GPOS: General-Purpose Operating System.
- HMI: Human-Machine Interface.
- I/O: Input/Output.
- IRQ: Interrupt Request.
- IS: Integrated Scheduling.
- NS: Non-Secure.
- OS: Operating System.
- PLC: Programmable Logic Controller.
- RPC: Remote Procedure Call.
- RTOS: Real-Time Operating System.
- SafeG: Safety Gate.
- SMC: Secure Monitor Call.
- SWI: Software Interrupt.
- TCB: Trusted Computing Base.
- TOPPERS: Toyohashi OPen Platform for Embedded Real-time Systems.
- TPM: Trusted Platform Module.
- TZIC: TrustZone Interrupt Controller.
- TZPC: TrustZone Protection Controller.
- UCB: Untrusted Computing Base.
- VCPU: Virtual CPU.

- V-Device: Virtual Device.
- VMM: Virtual Machine Monitor.
- VL: Virtualization Layer.

Contents

Contents XI

List of Figures XV

List of Tables XVII

1 Introduction 1
1.1 Motivation . . . 1
1.2 Overview . . . 2

2 Background 3
2.1 Real-time embedded systems . . . 3
2.2 Definition of a dual-OS system . . . 4
2.2.1 Dual-OS system example . . . 5
2.2.2 Dual-OS system requirements . . . 6
2.3 Dual-OS systems: state of the art . . . 8
2.3.1 Idle scheduling . . . 10
2.3.2 Comparison of existing approaches . . . 11
2.4 Overview of the ARM architecture . . . 12
2.4.1 Introduction . . . 12
2.4.2 Register file . . . 13
2.4.3 CPU modes . . . 13
2.4.4 Coprocesors . . . 14
2.4.5 Exceptions . . . 14
2.4.6 ARM TrustZone . . . 15
2.5 The SafeG dual-OS system . . . 16
2.5.1 TrustZone configuration in SafeG . . . 16
2.5.2 Execution flow model of SafeG . . . 17

3 Integrated scheduling **19**
3.1 Introduction 19
3.2 Motivational example 20
3.3 Assumptions and requirements 22
3.4 Integrated scheduling architecture 23
3.4.1 Overview and merits 24
3.4.2 Groups of GPOS activities 25
3.4.3 GPOS scheduling events 25
3.4.4 Tracking GPOS scheduling events of type 1 and 2 26
3.4.5 Tracking GPOS scheduling events of type 3 27
3.4.6 IS Manager 27
3.4.7 RTOS protection 30
3.4.8 Example 31
3.5 Implementation 33
3.5.1 Implementation platform 33
3.5.2 Linux kernel modifications 33
3.5.3 TOPPERS/ASP modifications 34
3.6 Evaluation 34
3.6.1 Requirement 3.1: GPOS Latency 34
3.6.2 Requirement 3.2: RTOS timeliness 35
3.6.3 Requirement 3.3: overhead 36
3.6.4 Requirement 3.4: GPOS modifications 37
3.6.5 Requirement 3.5: RTOS modifications 37
3.6.6 Requirement 3.6: SafeG modifications 38
3.6.7 Use case example 38
3.7 Related work 40
3.8 Conclusions 41

4 Dual-OS communications **43**
4.1 Introduction 43
4.2 Background 44
4.2.1 Dual-OS communications 44
4.2.2 Related work 45
4.3 Requirements and assumptions 46
4.3.1 Reliability requirements 46
4.3.2 Efficiency requirements 46

4.3.3 Assumptions 47
4.4 Communications architecture 47
4.4.1 Satisfying reliability requirements 48
4.4.2 Satisfying efficiency requirements 50
4.4.3 Communication channels 51
4.4.4 Dualoscom interface 52
4.4.5 Middleware 56
4.5 Implementation 59
4.5.1 Implementation platform 59
4.5.2 Code modifications 59
4.5.3 Initialization steps 60
4.6 Evaluation 60
4.6.1 Requirement 4.1: memory isolation 61
4.6.2 Requirement 4.2: shared control data 61
4.6.3 Requirement 4.3: real-time 62
4.6.4 Requirement 4.4: memory faults 63
4.6.5 Requirement 4.5: unbounded blocking 64
4.6.6 Requirement 4.6: code modifications 64
4.6.7 Requirement 4.7: throughput 64
4.6.8 Requirement 4.8: memory size 66
4.6.9 Requirement 4.9: interface 66
4.6.10 Discussion 66
4.7 Conclusions 66

5 Reliable device sharing **67**
5.1 Introduction 67
5.2 Motivation 68
5.3 Reliable device sharing 70
5.3.1 Requirements and assumptions 70
5.3.2 Suitability of existing device sharing approaches 70
5.3.3 Reliable device sharing through re-partitioning 73
5.4 Implementation 77
5.5 Evaluation 78
5.5.1 Overhead 78
5.5.2 Device latency 80
5.5.3 Code modifications 81

5.5.4 Discussion 81
5.6 Related work 81
5.7 Conclusions 83

6 Conclusions and future work **85**
6.1 Summary 85
6.2 Suggestions for future work 86

Bibliography **87**

List of publications by the author **95**

List of Figures

2.1 Overview of a real-time embedded system. 3
2.2 Architecture of a generic dual-OS system. 5
2.3 Motivational example for the research on dual-OS systems. 6
2.4 Hybrid kernel approach. 8
2.5 VMM/Hypervisor approach. 9
2.6 Partitioning approach. 10
2.7 The idle scheduling principle. 10
2.8 ARM general-purpose and program status registers. 12
2.9 SafeG architecture 16
2.10 SafeG idle scheduling. 17
2.11 Execution paths of the SafeG monitor. 18

3.1 Use case example. 21
3.2 Execution priority levels for the motivational example. 22
3.3 Integrated scheduling architecture. 24
3.4 Pseudocode of the GPOS scheduler hook function. 26
3.5 Tracking GPOS interrupts. 27
3.6 IS Manager architecture. 28
3.7 Pseudocode of the scheduling events FIQ handler. 28
3.8 Pseudocode of the Manager task. 29
3.9 Timeline: GPOS real-time task activation. 31
3.10 Evaluation of requirement 3.1. 35
3.11 Evaluation of requirement 3.2. 35
3.12 Frames per second in each scheduling approach. 39

4.1 Dual-OS communications example. 44
4.2 Previous communication approaches. 45
4.3 The proposed dual-OS communications architecture. 48

4.4 Elements of a dualoscom communication channel. 49
4.5 Behavior of the filtering functionality in both communication directions. . . 50
4.6 The dualoscom build process. 53
4.7 Pseudocode of RPC communication. 57
4.8 Support for sampling messages on top of the dualoscom interface. 58
4.9 Limiting mechanisms for dealing with message overload attacks. 62
4.10 Evaluation of message interrupt limiting. 63
4.11 Overhead comparison. 65
4.12 Execution flow comparison. 65

5.1 Motivational example for device sharing applied to an in-vehicle system. . . 69
5.2 Device sharing approaches (VL=Virtualization Layer, comm.=Dual-OS communications, V-Device=Virtual Device). 71
5.3 Architecture of the pure re-partitioning mechanism. 74
5.4 Pseudocode of the pure re-partitioning mechanism. 75
5.5 Architecture of the hybrid re-partitioning mechanism. 75
5.6 Pseudocode of the hybrid re-partitioning mechanism. 76
5.7 CPU performance for each mechanism. 79

List of Tables

2.1 Differences between real-time and general-purpose systems. 4
2.2 Comparison among dual-OS system approaches. 11
2.3 ARM exception vector table. 14

3.1 Overhead of the IS architecture. 36
3.2 Source code lines and binary size increase. 37
3.3 Tasks in the use case example. 38

4.1 Requirements Vs. Our design choices. 47
4.2 Overhead of the four steps algorithm. 61
4.3 Tasks for the evaluation of the message interrupt rate limiting functionality. 62
4.4 Size increase. 64

5.1 Qualitative comparison of device sharing approaches. 72
5.2 Execution time overhead per register access. 79
5.3 Device latency of each mechanism. 80
5.4 Number of source lines of code modified. 80

Chapter 1

Introduction

This chapter briefly introduces the contents of this thesis. The chapter starts with the motivation for our research, and then shows an overview of the main results obtained.

1.1 Motivation

Methods that consolidate a real-time control system and a highly functional system on a single platform for reducing hardware costs are gaining considerable interest from different embedded domains[13]. For example, the market for in-vehicle technology has experienced a rapid growth. New cars include sophisticated parking and driving aid systems, as well as less-critical functionality such as navigation, multimedia or Internet connectivity[19].

In order to develop highly functional applications—such as a web browser or a media player—efficiently, a general-purpose operating system (GPOS) with its advanced libraries is considered essential. However, most GPOSs are not able to satisfy the strict reliability requirements of real-time control systems due to their large scale[4, 21]. Instead, a target-specific real-time operating system (RTOS) running on a different hardware platform has been traditionally used. A promising approach to cope with such complexity at a lower hardware cost is the use of a dual-OS system[18, 14, 10] for consolidating an RTOS and a GPOS on the same platform thanks to the use of a virtualization layer (VL).

The goal of this work is to investigate mechanisms that allow for a tighter integration between the RTOS and the GPOS activities inside a dual-OS system. The motivation behind it is threefold: to address the latency bottleneck caused by RTOS background tasks on GPOS activities that require short response times; to enable the development of new collaborative applications with higher sophistication; and to further decrease the system's hardware cost through mechanisms that allow both OSs to share devices reliably.

1.2 Overview

The main three novel contributions to the reliable integration of a dual-OS system proposed in this thesis are as follows:

- An *integrated scheduling* framework which supports the interleaving of the execution priority levels from both OSs with high granularity. The framework uses execution-time reservations for guaranteeing the timeliness of the RTOS. The evaluation results show that the framework is suitable for enhancing the responsiveness of the GPOS time-sensitive activities without compromising the reliability and real-time performance of the RTOS.

- *Dual-OS communications* which allow the RTOS and GPOS to collaborate in complex distributed applications. In contrast to traditional approaches based on extending the virtualization layer with new communication primitives—which imposes a rather high overhead—we present a more efficient approach that minimizes the communication overhead caused by unnecessary copies and context switches; and satisfies the strict reliability requirements of the RTOS.

- Mechanisms for *sharing devices reliably* in dual-OS systems. We note that previous approaches based on paravirtualization are not well suited to device sharing patterns where the GPOS share greatly exceeds that of the RTOS. For that reason, we propose two new approaches that are based on dynamically re-partitioning devices between the RTOS and the GPOS at runtime. From the evaluation results, we observe a trade-off between the lower overhead and higher functionality of the re-partitioning approaches; and the shorter device latency of the paravirtualization approach.

The techniques mentioned above were all implemented on an real embedded platform running a highly reliable dual-OS system (SafeG) that was developed in previous work. From the evaluation results, we could confirm that the presented techniques are useful for improving the integration of dual-OS systems, without affecting the reliability of the RTOS.

The remainder of this thesis is organized as follows. Chapter 2 reviews the state of the art in dual-OS systems, and introduces details about the highly reliable dual-OS system (SafeG) that will be extended throughout the thesis. Chapters 3 to 5 are dedicated to the explanation of the three main contributions mentioned above. Finally, chapter 6 concludes this thesis and discusses future work.

Chapter 2

Background

This chapter reviews the state of the art in dual-OS real-time embedded system techniques, and summarizes the main aspects of SafeG—a highly reliable dual-OS system developed in previous work—whose understanding is necessary for the remainder of the thesis.

2.1 Real-time embedded systems

This thesis focuses on real-time embedded systems which result from the combination of typically resource-constrained hardware (i.e., processor and I/O) and special-purpose software designed to interact with a changing external environment within a larger system (see figure 2.1). In a real-time embedded system, the correctness of the system's behavior not only depends on the computed results, but also on the time at which they are generated.

Real-time applications typically consist of several concurrent *tasks* running on a special-purpose real-time operating system (RTOS). Tasks are triggered by the arrival of certain external *events* (e.g., a periodic clock event); and must provide a *response* (i.e., complete their work) before a time limit called *deadline*. The timespan between the instant when an event arrives and the instant when the corresponding task provides a response is known as *latency* or *response time*.

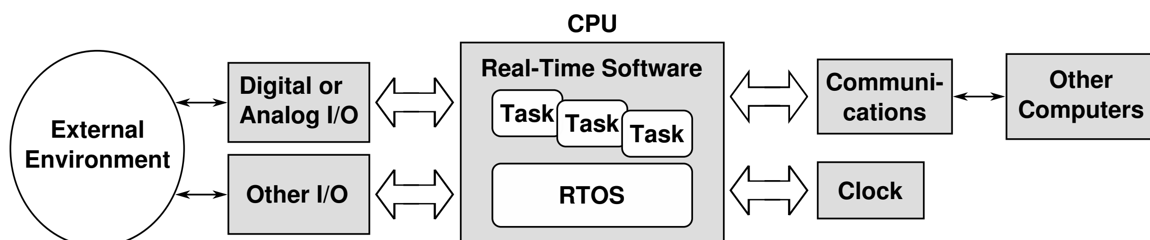


Figure 2.1: Overview of a real-time embedded system.

Table 2.1: Differences between real-time and general-purpose systems.

Property	General-purpose system	Real-time embedded system
Capacity	High throughput	Schedulability
Complexity	High functionality	High reliability
Responsiveness	Fast average response	Ensure worst-case latency
Overload	Fairness	Stability of critical part
Size	Large scale	Small scale

Table 2.1 shows the main differences between the properties of a real-time embedded system and a general-purpose (conventional) system. General-purpose systems are mainly concerned with increasing the average-case performance. Thus, their most important measures of merit are a high throughput, a high functionality—which typically comes at the cost of a larger size—a fast average response time, and fairness to all of its tasks. In contrast, real-time embedded systems are more concerned with ensuring that their worst-case behavior is predictable and acceptable. In particular, the reliability of the system and its ability to satisfy its timing requirements (i.e., *schedulability*) are paramount. For that reason, real-time embedded system engineers normally avoid including functionality that is not strictly necessary; and instead try to minimize the amount of trusted software. Real-time embedded systems also replace the notion of fairness by the notion of *stability*—note that if all timing requirements are met, then starvation or fairness are not an issue anymore. The notion of stability implies that a system that becomes unable to meet all of its timing requirements should at least ensure the deadlines of the most critical tasks, even at the cost of starving other less critical tasks.

During the following chapters, we will use the terms *real-time performance* and *timeliness* interchangeably for referring to the timing requirements of a real-time embedded system. Also, we define *hard* timing requirements as those whose dissatisfaction results in a system failure; and *soft* timing requirements as those whose dissatisfaction results in the degradation of the system performance.

2.2 Definition of a dual-OS system

A dual-OS system[12, 13, 18, 14] is a method for consolidating a real-time operating system (RTOS) and a general-purpose operating system (GPOS) onto the same embedded platform for reducing the total hardware cost (see figure 2.2). This is usually achieved through a virtualization layer (VL) that allows multiplexing the underlying hardware resources between both operating systems. The RTOS provides support for applications with strict reliability

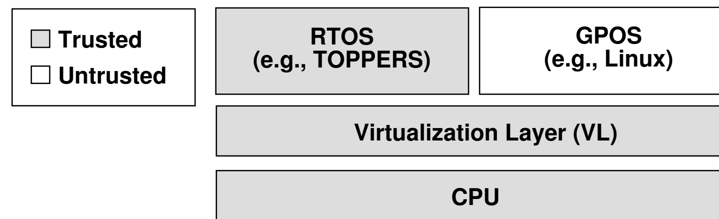


Figure 2.2: Architecture of a generic dual-OS system.

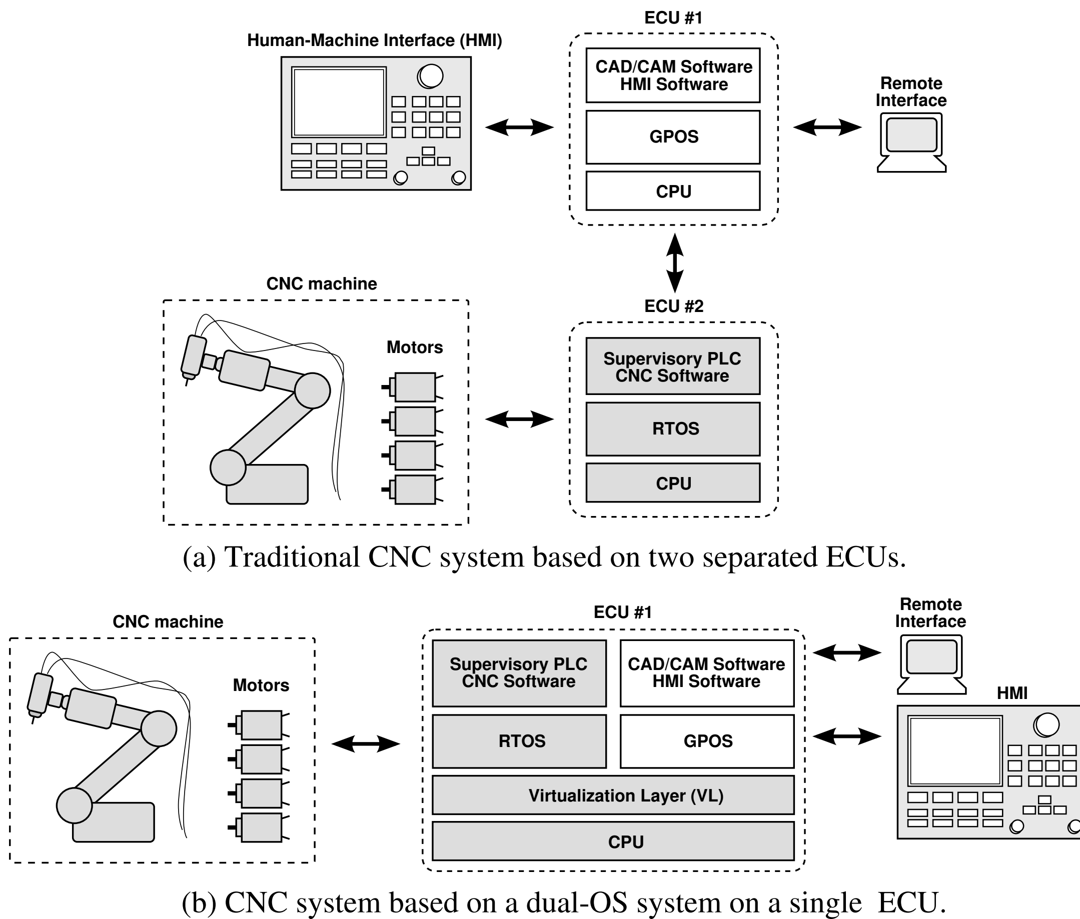
requirements (e.g., timeliness, isolation and security requirements). Both the RTOS and the VL are small scale and considered to belong to the trusted computing base (TCB). In contrast, the GPOS provides support for applications with high functionality requirements, and is considered to belong to the untrusted computing base (UCB) due to its large scale.

2.2.1 Dual-OS system example

Figure 2.3 illustrates a motivational example that applies the dual-OS system concept to a computer numerical control (CNC) system. CNC systems are often used in the manufacturing business for any process that can be described as a series of movements and operations (e.g., laser cutting, welding or picking and placing). Typical CNC machines have at least four motors driving motion axes that must move synchronously and with very high precision. The loss of a single control cycle causes the CNC machine to enter fault state, in order to prevent the machine from moving in such a way that can be harmful to the parts being machined or the machine tools themselves. The reliability and real-time requirements of the software controlling a CNC machine are very strict. For that reason, most CNC systems run their control software on top of a deterministic and highly reliable RTOS.

However, vendors need to increase the productivity of their CNC systems for being competitive. For example, having a good user interface, better usability or flexibility represents extra value to their customers. Most CNC systems accomplish that by leveraging the rich functionality of a GPOS to support advanced features such as human-machine interface (HMI) software, data logging, machine program development or remote interfaces.

Thus, the challenge for CNC vendors is to find an efficient method for combining all these applications with conflicting requirements into the same system. The traditional approach followed by most vendors is depicted in figure 2.3(a). It involves the use of two separate electronic computing units (ECUs) for running an RTOS and a GPOS. The main disadvantage of this approach is the extra hardware cost caused by the need of two separate ECUs. In contrast, figure 2.3(b) illustrates an alternative approach that uses a dual-OS system for consolidating both OSs on a single ECU in order to reduce the hardware cost.



(a) Traditional CNC system based on two separated ECUs.

(b) CNC system based on a dual-OS system on a single ECU.

Figure 2.3: Motivational example for the research on dual-OS systems.

2.2.2 Dual-OS system requirements

No matter what the implementation strategy is, any dual-OS system should satisfy a set of fundamental requirements[15]. The following set of requirements for dual-OS systems has been created by taking into account the needs of several embedded domains, such as CNC machines, car navigation systems, chemistry control systems and mobile phones.

REQUIREMENT 2.1 *Overhead*: minimize any unnecessary overhead.

REQUIREMENT 2.2 *Timeliness*: guarantee the real-time performance of the RTOS.

REQUIREMENT 2.3 *Isolation*: protect the RTOS memory and devices against any misbehavior or attack coming from the UCB.

REQUIREMENT 2.4 *Maintainability*: minimize modifications to the source code.

REQUIREMENT 2.5 *TCB Size*: minimize the size of the TCB.

REQUIREMENT 2.6 *Integration*: provide mechanisms for the deep integration of both operating systems without affecting the reliability of the TCB.

Requirement 2.1 is related to the decrease in performance caused by the interposition of a VL between the guest operating systems and the underlying hardware platform. Dual-OS systems must minimize this overhead.

Requirement 2.2 is fundamental for the timely behavior of RTOS applications. Therefore, dual-OS systems must guarantee that the RTOS has full control over the timing behavior of its tasks and interrupt handlers. As a corollary, it follows that the RTOS worst-case latency must be bounded regardless of any misbehavior by the GPOS.

Requirement 2.3 is defined because the large scale of GPOS software often leads to the appearance of defects (i.e., bugs) that can potentially compromise the reliability and security of the RTOS. Therefore, protecting the RTOS memory and devices against any misbehavior or malicious attack coming from the UCB is necessary to guarantee the correct execution of the RTOS. In particular, dual-OS systems must also protect the RTOS memory against DMA attacks—caused by devices with direct memory access that are maliciously controlled by the GPOS. Additionally, the RTOS in a dual-OS system is often used as a replacement for the trusted platform module (TPM[22]) in security applications. In such scenarios, the RTOS memory and device registers may contain sensitive data that must not be accessible by the GPOS. For that reason, memory and device isolation must be implemented not only for *write* but also for *read* GPOS accesses.

Requirement 2.4 refers to the fact that running two operating systems onto the same platform may involve changes (e.g., kernel patches) to their hardware abstraction layer. In particular, the maintenance of these changes on a large-scale and rapidly evolving GPOS has been proven to be an extremely hard job[1]. Therefore, dual-OS systems should minimize the amount of changes required to guest OSs.

Requirement 2.5 refers to the fact that in order to achieve a high degree of reliability the size and complexity of the trusted code must be minimized.

Requirement 2.6 has a wider meaning than the other previous requirements. It expresses the fact that running the two guest OSs in complete isolation can cause certain inefficiencies. In particular, a few problems that can benefit from a better integration of both OSs are the scheduling of GPOS soft real-time tasks and interrupts (see § 3.2); services that require a communications system between both OSs (see § 4.2.1); and device sharing (see § 5.2). Note that although a dual-OS system should contribute to the tighter integration of both OSs, this integration must not affect the satisfaction of the other set of requirements.

In § 2.3, we perform a comparison among several existing approaches to the implementation of dual-OS systems, and qualitatively evaluate their ability to satisfy the set of requirements defined above.

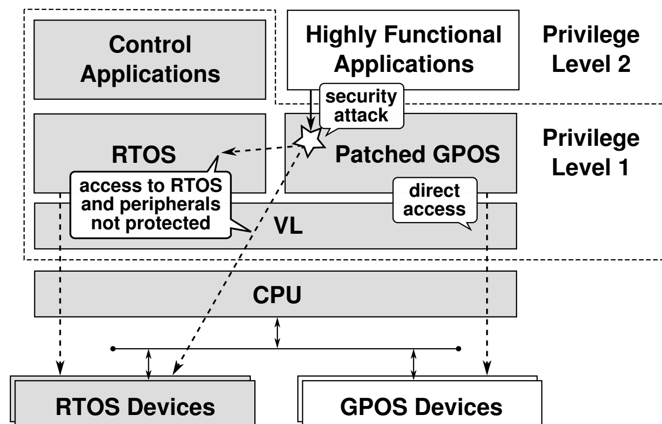


Figure 2.4: Hybrid kernel approach.

2.3 Dual-OS systems: state of the art

The majority of the existing approaches to consolidating an RTOS and a GPOS onto the same platform can be classified in the following categories:

- *Hybrid kernels*: this was one of the first successful approaches to the consolidation of an RTOS and a GPOS onto the same platform. Although details vary depending on the implementation[39, 10, 11], virtually all of them were based on patching the GPOS kernel code (usually Linux) with a software VL that controls the interrupt controller and gives priority to the execution of the RTOS. Figure 2.4 depicts the hybrid kernel approach. The main advantage of this approach is its low overhead thanks to the fact that both operating systems execute with the highest privilege level of the processor, and therefore have direct access to the hardware. However, this comes at the cost of not having enough memory isolation between both OSs. For that reason, the RTOS is vulnerable to malicious attacks or faults in the GPOS. For example, if the GPOS runs out of control and disables all interrupts, the RTOS would not be able to recover control. Furthermore, the GPOS may access memory assigned to the RTOS, causing its failure or stealing sensitive information.

- *Virtual Machine Monitor (VMM) or Hypervisor*: this approach consists of virtualizing the underlying hardware resources and multiplexing them between several guest operating systems. Although originally VMM approaches have focused on cloud computing systems[16] with many guest OSs, the same virtualization techniques can be applied for the implementation of dual-OS systems[4, 6]. There exist several methods to implement virtualization of resources. However, hypervisors for dual-OS systems usually expose a software interface (*paravirtualization*) and require the hardware ab-

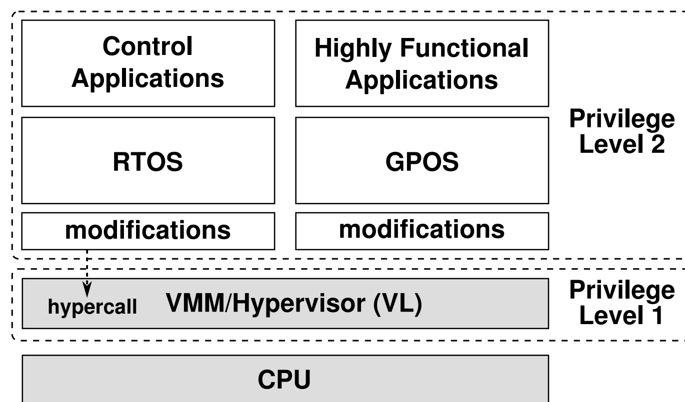
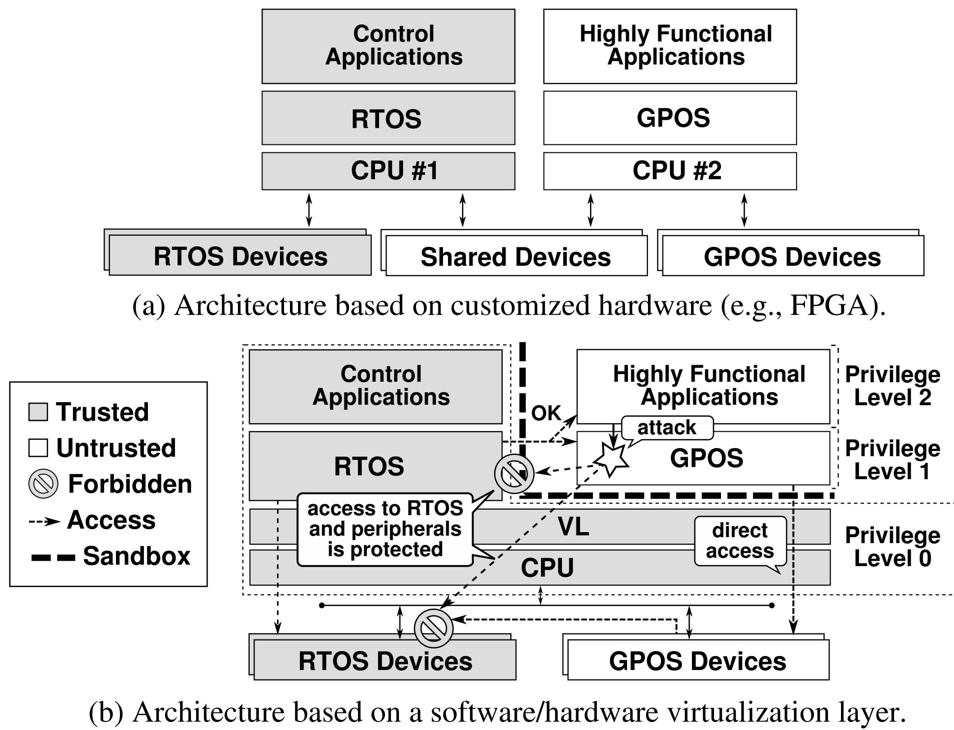


Figure 2.5: VMM/Hypervisor approach.

straction layer of the guest OSs to be modified (see figure 2.5). The main advantage of the VMM/Hypervisor approach over the hybrid kernel approach is a better isolation among the guest operating systems. This is achieved by executing them under a lower processor privilege level and enforcing their isolation through memory protection mechanisms. In this sense, we can say that hypervisors are close to the concept of microkernels[4] because they focus on minimizing the amount of software that belongs to the TCB. The main disadvantages of this approach are the execution overhead caused by the need to call the hypervisor (*hypercalls*) for the execution of privileged operations; and the amount of modifications to the guest operating systems[1] required. Additionally, many implementations are vulnerable to DMA attacks against the RTOS memory, caused by devices controlled by a compromised GPOS[68].

- *Hardware-aid resource partitioning*: this approach consists of partitioning all of the hardware resources in the system, and assigning those that are critical for the reliability of the system to the RTOS and the rest of them to the GPOS. The main advantages of this approach are its strong isolation—including protection against DMA-based attacks—and near native performance, because both OSs can access their corresponding devices directly. Another important advantage is the fact that guest OSs do not require extensive modifications. On the downside, this approach usually has strong dependencies on the underlying hardware; and lower flexibility compared to the VMM approach. Figure 2.6(a) depicts an architecture that implements the partitioning approach by using customized dual-core hardware as in [23]; and figure 2.6(b) depicts an alternative architecture that uses hardware virtualization extensions to accomplish the same goal. In § 2.5 we describe SafeG, an implementation of the partitioning approach based on the ARM TrustZone hardware virtualization extensions.



(a) Architecture based on customized hardware (e.g., FPGA).

(b) Architecture based on a software/hardware virtualization layer.

Figure 2.6: Partitioning approach.

2.3.1 Idle scheduling

Most existing single-core dual-OS system implementations[39, 10, 5, 6] use the so-called *idle scheduling* principle for multiplexing processing time. This principle, illustrated by figure 2.7, implies that the GPOS—including its soft real-time tasks and interrupt handlers—always executes with lower *global priority* than the RTOS. Thanks to this principle, dual-OS system architectures can guarantee that the real-time performance of the RTOS is independent of the execution of the GPOS, as required by requirement 2.2 (*timeliness*).

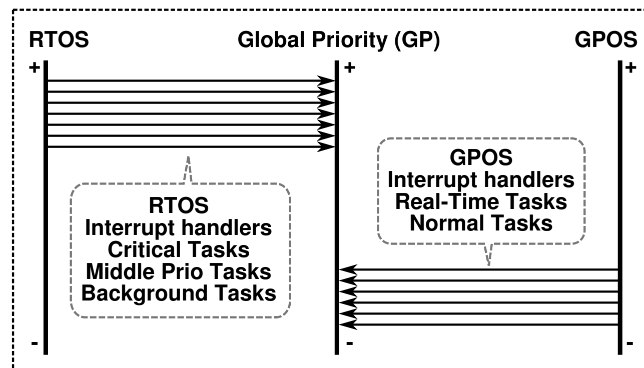


Figure 2.7: The idle scheduling principle.

Table 2.2: Comparison among dual-OS system approaches.

Requirement name	Number	Hybrid	VMM/Hypervisor	Partitioning
Overhead	2.1	✓	✗	✓
Timeliness	2.2	✓	✓	✓
Isolation	2.3	✗	✓	✓
Maintainability	2.4	✗	✗	✓
TCB Size	2.5	✗	✓	✓
Integration	2.6	✓	✓	✗

However, the idle scheduling principle—although useful for providing time isolation to the RTOS—is not suitable for the scheduling of GPOS activities with soft real-time requirements (e.g., a movie player). For example, § 3.2 illustrates some issues of the idle-scheduling principle through an example dual-OS system where the RTOS controls the movements of a robot, and logs them into a secure disk while the GPOS displays a movie.

In § 3.4, we propose an alternative scheduling mechanism based on the integration of the schedulers of both operating systems through CPU-time resource reservations.

2.3.2 Comparison of existing approaches

Table 2.2 compares the ability of the three approaches mentioned above (i.e., hybrid kernel, VMM/hypervisor and hardware-aid resource partitioning) to satisfy the dual-OS system requirements defined in § 2.2.2. From the point of view of a system engineer, there are several trade-offs that must be gauged when selecting a dual-OS system for a specific target application. If the hardware platform does not have virtualization support, then the only two possible choices are using a hybrid kernel or a hypervisor. Normally, the hypervisor approach should be selected—because it supports memory isolation and has a smaller TCB—unless the overhead incurred is not acceptable, in which case the hybrid kernel approach may be the only remaining option. If the hardware platform does support virtualization, the two major choices are either using a hypervisor or following a partitioning strategy. Although partitioning is superior to the hypervisor approach in most aspects—in particular it has a lower overhead and it is easier to maintain—there are certain target applications that require high flexibility where hypervisors may be the choice of preference.

In this thesis, we investigate advanced integration techniques that allow hardware-aid resource partitioning approaches to better satisfy the flexibility requirements of such applications. In particular, we put emphasis on achieving a tight integration between both guest OSs without affecting the best properties (i.e., low overhead, timeliness, memory isolation, maintainability and small TCB size) of the partitioning approach.

	System level views							
		Privileged modes						
			Exception modes					
Application level view	User mode	System mode	Supervisor mode	Monitor mode	Abort mode	Undefined mode	IRQ mode	FIQ mode
R0	R0_usr							
R1	R1_usr							
R2	R2_usr							
R3	R3_usr							
R4	R4_usr							
R5	R5_usr							
R6	R6_usr							
R7	R7_usr							
R8	R8_usr							R8_fiq
R9	R9_usr							R9_fiq
R10	R10_usr							R10_fiq
R11	R11_usr							R11_fiq
R12	R12_usr							R12_fiq
SP	SP_usr		SP_svc	SP_mon	SP_abt	SP_und	SP_irq	SP_fiq
LR	LR_usr		LR_svc	LR_mon	LR_abt	LR_und	LR_irq	LR_fiq
PC	PC							
APSR	CPSR							
			SPSR_svc	SPSR_mon	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Figure 2.8: ARM general-purpose and program status registers.

2.4 Overview of the ARM architecture

This thesis focuses on single-core *embedded* systems (e.g., resource-constrained systems embedded in a larger system) rather than on more powerful machines. For that reason, we implemented our techniques on a dual-OS system (SafeG) that runs on a popular ARM embedded single processor. This section briefly reviews the ARM architecture and SafeG.

2.4.1 Introduction

ARM[43] (Advanced RISC Machine) is a 32-bit (there is a special mode—called Thumb mode—for running 16-bit instructions too) instruction set architecture (ISA) based on the philosophy of reduced instruction set computers (RISC). In contrast to complex instruction set computer (CISC) architectures, ARM has (mostly) simple 1-cycle instructions, broken down pipelines that can execute in parallel, and a large general-purpose register set. ARM

instructions have a rather high density and may encode notable features such as conditional execution. Unlike CISC architectures data processing instructions operate exclusively with registers, and separate load and store instructions are used to move data between registers and external memory. The ARM architecture has evolved over time through several versions (e.g., ARMv1, ARMv2...ARMv7), and each architecture version includes several families or profiles (e.g., ARM7TDMI or ARM Cortex-A).

2.4.2 Register file

The ARM architecture contains a total of 16 32-bit general-purpose registers (see figure 2.8), and some of them are banked depending on the CPU mode (see § 2.4.3 for an explanation about CPU modes). The value contained in a general-purpose register can be either some data or an address. General-purpose registers are identified as r0 to r15, although registers r13 to r15 are typically aliased as sp (stack pointer), lr (link register) and pc (program counter). Additionally, there are two program status registers (PSR): the Current Program Status Register (CPSR) and the banked Saved Program Status Register (SPSR). PSRs are used for ARM cores to monitor and control internal operation (i.e., hold processor status and control information). The format of both PSR includes a set of condition code flags, exception mask bits and the current processor mode.

2.4.3 CPU modes

An ARM processor can be in any of the following modes at a given time—note that for simplicity, we have omitted the newest hypervisor mode released in the ARMv8 architecture:

- *User mode* (USR): the non-privileged mode for user applications.
- *System mode* (SYS): a privileged mode that can only be entered by modifying the mode bits of the CPSR.
- *Supervisor mode* (SVC): a privileged mode entered through the SWI (Software Interrupt) instruction and used typically for the kernel to execute system calls.
- *Abort mode* (ABT): a privileged mode that is entered when a prefetch or data abort exception occurs.
- *Undefined mode* (UND): a privileged mode that is entered when an undefined instruction exception occurs.
- *Interrupt mode* (IRQ): a privileged mode that is entered when an interrupt request (IRQ) arrives and the corresponding the CPSR mask bit is enabled.

Table 2.3: ARM exception vector table.

Exception	Mode	Memory offset
Reset	SVC	0x00
Undefined instruction	UND	0x04
Supervisor call	SVC	0x08
Prefetch abort	ABT	0x0C
Data abort	ABT	0x10
IRQ (interrupt)	IRQ	0x18
FIQ (fast interrupt)	FIQ	0x1C

- *Fast Interrupt mode* (FIQ): a privileged mode that is entered when a fast interrupt request (FIQ) arrives and the corresponding the CPSR mask bit is enabled.
- *Monitor mode* (MON): a privileged mode for the execution of the TrustZone monitor software (see § 2.5 for details).

2.4.4 Coprocesors

ARM processors use coprocessors to extend the architecture without having to add new complex instructions or registers. The ARM architecture allows up to 16 coprocessors. Coprocessor 15 (CP15) is reserved for typical control functions such as the overall system configuration, TLB and Cache management, system performance monitoring, or memory management.

2.4.5 Exceptions

An exception in the ARM architecture involves the occurrence of a certain condition that causes the normal execution of the processor to suspend, change mode and load a special address within an pre-registered exception vector table. Table 2.3 shows the name of the existing exceptions, the mode in which they are processed and the relative offset to the beginning of the vector table (typically placed at address 0 or 0xFFFF0000). Note that the Supervisor call is replaced by the Secure Monitor call (SMC) in the vector table used in monitor mode (see § 2.4.6).

ARM processors have two types of interrupt known as FIQ (Fast Interrupt Request) and IRQ (Interrupt Request). The main difference between them resides in the fact that FIQ interrupts have higher priority and more banked registers than IRQ interrupts. FIQ and IRQ interrupts can be disabled within a privileged mode by setting two flag bits (F and I respectively) of the CPSR.

2.4.6 ARM TrustZone

ARM TrustZone is a set of hardware extensions present in high-end ARM embedded processors such as ARM 1176[31] or the Cortex-A series. This section briefly introduces some concepts of ARM TrustZone that are necessary for understanding the rest of the thesis. For more information, refer to [29, 31, 5, 12].

- *Virtual CPUs*: an ARM TrustZone-enabled single processor provides two Virtual CPUs (VCPUs), the Secure VCPU and the Non-Secure VCPU. Each VCPU is equipped with its own memory management unit and exception vector table; and supports all ARM operation modes (i.e., User, FIQ, IRQ, Supervisor, Abort, System and Undefined modes). It is important to understand that these two VCPUs do not run in parallel but rather in a time-sliced fashion (i.e., in turns). In other words, only one of the VCPUs can be active at any given time.

- *The Monitor*: the Secure VCPU has an additional mode —called the monitor mode— which is used to context switch between both VCPUs. The software executing in monitor mode is commonly known as the secure Monitor. The entry to monitor mode is tightly controlled and can only be triggered by software executing the Secure Monitor Call (SMC) instruction or the occurrence of a hardware exception (i.e., IRQ, FIQ, Data abort and Prefetch abort exceptions) through an exception vector table in monitor mode. A VCPU context switch involves saving and restoring all ARM general purpose registers plus coprocessor registers that are shared by both VCPUs.

- *Address space partitioning*: when a bus master accesses memory or devices, the NS bit (Non-Secure bit) is propagated through the system bus indicating the privilege of that access (i.e., secure or non-secure). This allows the partitioning of the address space into two virtual worlds: the Secure and the Non-Secure world. This partitioning can be done statically by the hardware vendor or at runtime through the TrustZone Protection Controller (TZPC[36]). The Secure VCPU is allowed to access memory and devices from both worlds. However, hardware logic makes sure that Secure world memory and devices cannot be accessed by the Non-Secure VCPU nor other Non-Secure bus masters, such as Non-Secure DMA devices.

- *Device interrupts partitioning*: In a TrustZone-enabled processor, ARM recommends that Secure devices are configured to generate FIQ interrupts and Non-Secure devices are configured to generate IRQ interrupts. This configuration is carried out through the TrustZone Interrupt Controller (TZIC[32]) which is only accessible from

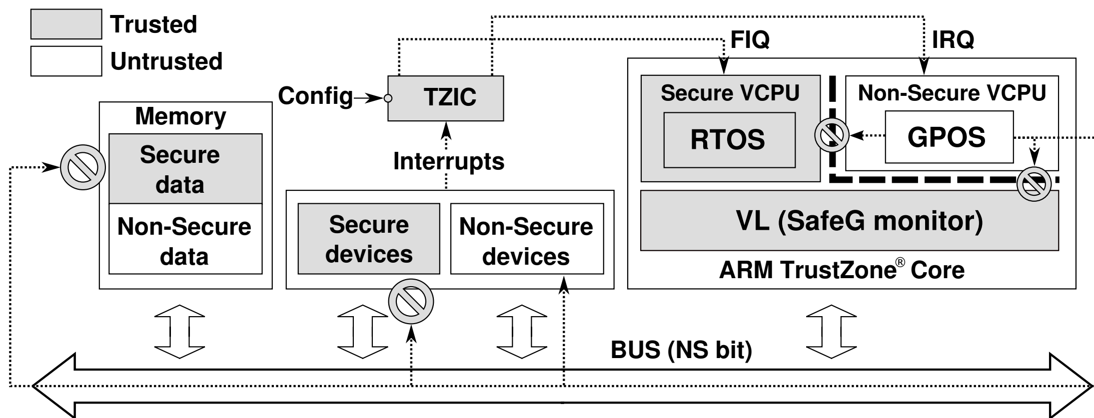


Figure 2.9: SafeG architecture

the Secure VCPU. To prevent Non-Secure software masking Secure device interrupts, TrustZone provides the FW (F flag Writable) bit which is only accessible by the Secure VCPU. When the FW bit is set to zero the F flag becomes non maskable for the Non-Secure world. That is, Non-Secure software cannot disable FIQ interrupts (i.e., interrupts generated by Secure devices).

2.5 The SafeG dual-OS system

SafeG (*Safety Gate*) is a reliable dual-OS system presented in previous work[18]. It was designed to support the concurrent execution of an RTOS and a GPOS on top of an ARM TrustZone-enabled[29] single core. SafeG guarantees that the RTOS memory is protected from the GPOS, which is considered by default unreliable. Moreover, SafeG provides time isolation for the RTOS to guarantee that hard real-time tasks always meet their deadlines. Both memory and time isolation are backed by the ARM TrustZone hardware extensions, which allows for a low-overhead implementation and minimal modifications to the GPOS.

2.5.1 TrustZone configuration in SafeG

Figure 2.9 depicts the overall organization of the SafeG architecture. The SafeG architecture takes advantage of ARM TrustZone hardware security extensions to concurrently execute an RTOS and a GPOS on top of the same single-core processor. The SafeG monitor—which is the main component of the SafeG architecture—is a specific implementation of the ARM TrustZone monitor. It focuses on guaranteeing the real-time performance requirements and memory isolation of the RTOS. The SafeG architecture leverages the ARM TrustZone hardware security extensions under the following configuration:

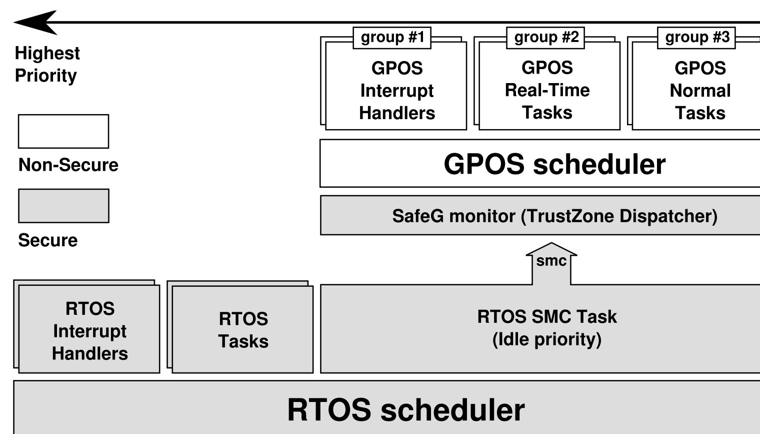


Figure 2.10: SafeG idle scheduling.

- *Virtual CPUs*: in the SafeG architecture the GPOS is assigned to the Non-Secure VCPU; and the RTOS and the SafeG monitor are assigned to the Secure VCPU.

- *The Monitor*: the *SafeG monitor* is the software component of the SafeG architecture that executes under monitor mode and handles the switching between the GPOS and the RTOS. The entry to SafeG monitor can only be triggered by software executing the SMC instruction or the occurrence of an FIQ while the Non-Secure VCPU is active. The SafeG monitor is small—around 2KB[18]—and executes with all interrupts disabled, which simplifies its verification. A VCPU context switch on an ARM1176[31] processor requires around 200 cycles[18].

- *Address space partitioning*: during initialization SafeG configures RTOS memory and devices as Secure world resources; and GPOS memory and devices as Non-Secure world resources. For that reason, the RTOS address space is protected from potentially malicious accesses by the GPOS.

- *Device interrupts partitioning*: SafeG architecture configures RTOS devices (i.e., Secure devices) to generate FIQ interrupts; and GPOS devices (i.e., Non-Secure devices) to generate IRQ interrupts. This is done through the TZIC, which cannot be accessed from the Non-Secure state.

2.5.2 Execution flow model of SafeG

The execution flow within the SafeG architecture is controlled by two fundamental principles that allow the RTOS to guarantee real-time performance requirements to its tasks and interrupt handlers.

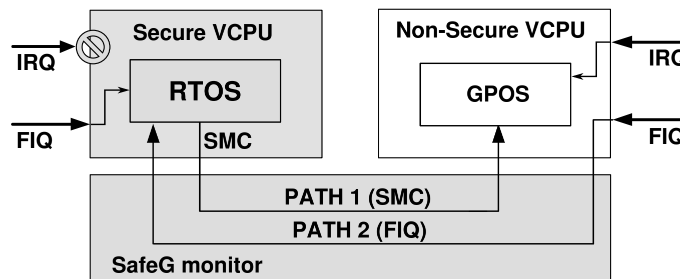


Figure 2.11: Execution paths of the SafeG monitor.

- *Idle scheduling*: the original SafeG architecture uses the idle scheduling principle explained in § 2.3.1, which implies that the GPOS is only allowed to execute during the RTOS idle time. The SafeG architecture can be seen as a two-level hierarchical scheduler (see figure 2.10) where the RTOS scheduler plays the role of a global priority scheduler; and the GPOS scheduler acts as a local scheduler. The whole GPOS is a black box represented in the RTOS scheduler by a task executed at idle priority. We call this task *RTOS SMC Task* because its body consists of a loop executing the SMC instruction to invoke the SafeG monitor. The SafeG monitor plays the role of a dispatcher that context switches to the GPOS whenever the RTOS becomes idle.

- *Processor control recovery*: this principle refers to the ability of the RTOS to recover control of the processor at any time through the use of an FIQ interrupt. When the RTOS is executing (i.e., Secure VCPU is active) IRQ interrupts are disabled (i.e., the I flag is one). This prevents GPOS devices interrupting the execution of RTOS tasks. On the other hand, when the GPOS is executing (i.e., Non-Secure VCPU is active) FIQ interrupts are always enabled (i.e., the F flag is zero). For that reason, the RTOS can recover the control of the processor at any time (e.g., by using a Secure timer device). The TrustZone FW configuration bit is set to zero to prevent the GPOS disabling FIQ interrupts (i.e., the GPOS cannot set the F flag to one).

Figure 2.11 illustrates the two main execution paths of the SafeG monitor. PATH 1 (SMC) is used by the RTOS SMC Task to context switch to the GPOS whenever the RTOS becomes idle. PATH 2 (FIQ) occurs when an FIQ interrupt arrives to the processor while the Non-Secure VCPU is active. The arrival of the FIQ interrupt forces the processor to enter Monitor mode, where SafeG FIQ vector handler switches back to the RTOS.

Chapter 3

Integrated scheduling

In this chapter, we replace the idle scheduling principle from the original SafeG architecture by an integrated scheduling architecture where the execution priority level of the GPOS and RTOS activities can be mixed with high granularity. The evaluation results show that the proposed approach is suitable for enhancing the responsiveness of the GPOS time-sensitive activities without compromising the reliability and real-time performance of the RTOS.

3.1 Introduction

The most extended approach to guaranteeing the real-time performance of the RTOS in a dual-OS system is the *idle scheduling* principle (see § 2.3.1), which consists of scheduling the GPOS with lower priority than the RTOS. The main advantages of this approach are its simplicity and its robustness against the presence of faulty or malicious GPOS software.

However, the idle scheduling principle is not suitable when the GPOS contains time-sensitive activities (e.g., multimedia tasks or device interrupt handlers). For example, the hardware buffer of a GPOS network card may get overwritten by the arrival of new packets if the execution of the corresponding GPOS interrupt handler is delayed for an excessive amount of time by the execution of the RTOS. This is particularly true if the RTOS contains compute-bound activities (e.g., security services) with a rather long execution time.

Porting GPOS time-sensitive applications and drivers to the RTOS is usually a flawed idea from the point of view of maintainability and TCB size. Similarly, moving RTOS compute-bound activities to the GPOS is often unsatisfactory. For example, security services (e.g., digital rights management) must run isolated from the UCB, and access special devices (e.g., a smart card with cryptographic keys) that are not accessible by the GPOS.

In § 3.2 and § 3.6.7, we describe a motivational example with more detail where the

RTOS in a dual-OS system based on SafeG controls a robot, and logs all of its operations into a secure disk while the GPOS displays a movie.

The goal of the work explained in this chapter is to fix the shortcomings of the idle scheduling approach. The most important contribution is an *integrated scheduling* (IS) architecture whose main features are:

- It supports mixing the global execution priority level of the GPOS and RTOS activities. This provides the system engineer with a means to configure the execution of a GPOS time-sensitive activity to have higher priority than the execution of an RTOS compute-bound task.

- The real-time performance and reliability of the RTOS is guaranteed even if the GPOS is faulty or misbehaves. More in detail, the execution of GPOS time-sensitive activities is controlled and limited through CPU-time resource reservations. This mechanism guarantees that RTOS tasks executing with lower priority will not suffer unbounded blocking nor starvation.

- The architecture does not require modifications to the source code of the dual-OS virtualization monitor nor to the RTOS kernel. This feature makes the integrated scheduling architecture easier to maintain and verify.

We have built the IS architecture on a physical platform, and evaluated it through several experiments and a realistic use case application. The evaluation results show that the architecture is suitable for enhancing the responsiveness of GPOS time-sensitive activities—by scheduling RTOS compute-bound tasks at a lower priority—and the overhead introduced is small enough for practical usage. This is accomplished without affecting the reliability and real-time performance of the RTOS.

The remainder of this chapter is organized as follows. § 3.2 explains the main issues present in idle scheduling that motivates this research. § 3.3 enumerates the main requirements for the IS architecture. § 3.4 constitutes the core of this chapter and explains the IS architecture. § 3.5 details the implementation of the IS architecture. § 3.6 evaluates it, and includes a use case example that shows its effectiveness in a real scenario. § 3.7 compares this research with previous work. Finally, the chapter is concluded in § 3.8.

3.2 Motivational example

Figure 3.1 illustrates an example that motivates the development of the proposed IS architecture. The hardware configuration of the system consists of the following elements:

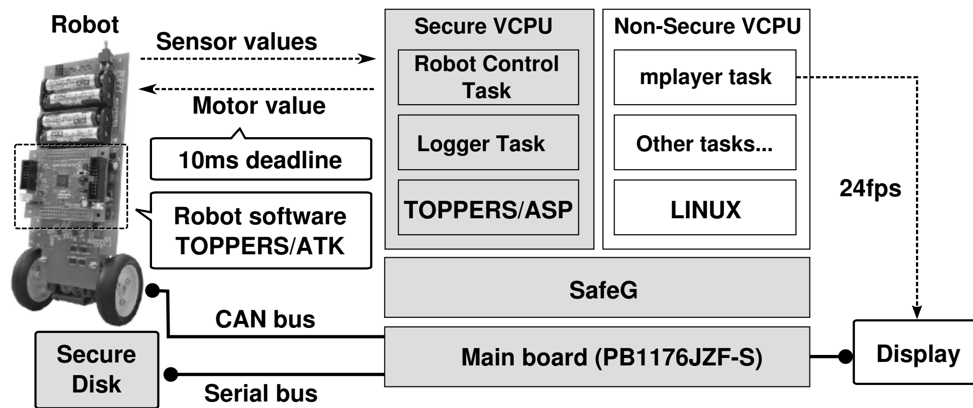
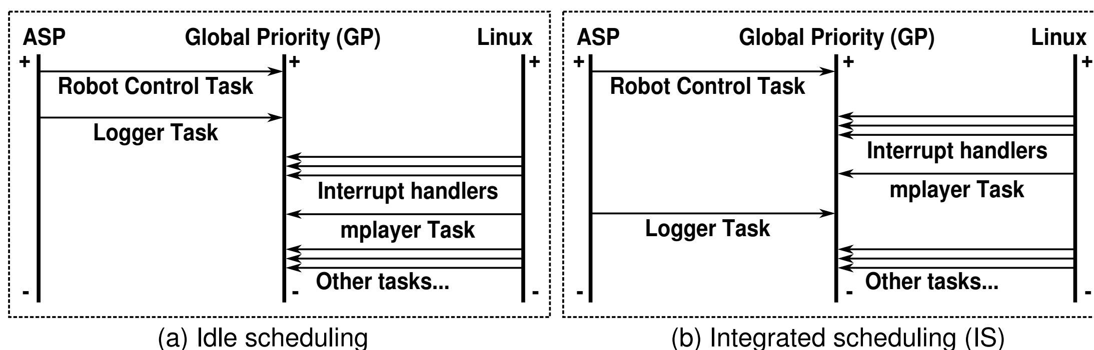


Figure 3.1: Use case example.

- *Main board*: a PB1176JZF-S[35] board, which contains an ARM1176 processor with TrustZone support.
- *Robot*: a *Puppy* robot[41] which contains a gyroscope and rotary encoder sensors. It is connected to the main board through a CAN bus, whose controller is assigned to the Secure VCPU.
- *Secure disk*: a disk to store secure data. It is accessed through a serial bus interface which is assigned to the Secure VCPU.
- *Display*: used by the Non-Secure VCPU to display a video.

The software configuration of the system is composed of the following elements:

- *Robot software*: the robot runs an application on top of TOPPERS/ATK[30]. The application sends the values of the gyroscope and rotary encoder sensors to the main board through the CAN bus with a period of 10ms. Then, it waits for an answer containing an appropriate motor value for the robot to keep balance. If this value arrives to the robot later than the next period (i.e., misses its deadline) the robot will lose balance and fall down.
- *Secure VCPU software*: the Secure VCPU contains two tasks running on top of ASP. The *robot control task* waits for messages coming from the Puppy robot—containing the sensor values—and sends replies back with the calculated values for the motor. The *logger task* is a compute-bound task whose main function is to encrypt and store the execution log generated by the *robot control task* onto the secure disk.
- *Non-Secure VCPU software*: the Non-Secure VCPU contains Linux with a minimal filesystem based on buildroot[34]. On top of that, a movie player application—called mplayer[42]—is used to show a 24fps MPEG4 video on the display through the Linux framebuffer device and executed with the maximum real-time priority in Linux.



(a) Idle scheduling (b) Integrated scheduling (IS)

Figure 3.2: Execution priority levels for the motivational example.

- *SafeG monitor*: as explained in § 2.5.1, the SafeG monitor executes under TrustZone monitor mode. It handles the switching between ASP and Linux following the idle scheduling principle, as illustrated in figure 3.2(a).

We built the system in figure 3.1 on a physical platform and observed that the robot performed correctly without any deadlines being missed. However, we noticed that the dynamic frame rate of the video experimented strong drops (see figure 3.12), to the extent that the user's watching experience became unacceptable. We realized that the strong drops in frame rate were caused by the blocking time imposed by the periodic execution of ASP's logger task on the mplayer application.

In this chapter, we propose a novel solution to this problem that replaces the idle scheduling principle by an integrated scheduling architecture capable of mixing the execution priority levels of both OSs. This allows the system engineer to modify the schedule of the motivational example from the one depicted in figure 3.2(a) to the one depicted in figure 3.2(b), which increases the execution priority of the Linux interrupt handlers and the mplayer task over the priority of the compute-bound ASP logger task. Furthermore, we leverage previous work on aperiodic servers[49] to limit (and enforce) the execution time of the Linux soft real-time activities, which is needed to ensure that ASP tasks executing with lower priority (e.g., the logger task) do not starve or miss a deadline.

3.3 Assumptions and requirements

We define the following set of initial assumptions. Relaxing these assumptions for wider usage is left for future work:

(a) The RTOS scheduler assigns each task to a fixed priority level. Tasks can be preempted by tasks with higher priority level (i.e., fixed-priority preemptive scheduling[26]).

(b) The GPOS scheduler must allow tasks with (soft) real-time requirements to take precedence over normal tasks by allocating a range of higher execution priority levels. Most popular GPOSs provide this feature.

(c) We need access to the source code of the GPOS scheduler. For that reason we will use an open source GPOS kernel.

Next, we define a list of requirements that the IS architecture must satisfy. Many of these requirements are common to other virtualization architectures [15]:

REQUIREMENT 3.1 *GPOS latency*: GPOS interrupt handlers and real-time tasks can be configured to take precedence over certain RTOS tasks with lower priority.

REQUIREMENT 3.2 *RTOS timeliness*: the worst-case response time of RTOS interrupt handlers and tasks must remain upper-bounded in all cases.

REQUIREMENT 3.3 *Overhead*: the execution time overhead introduced by the IS architecture must be small enough for practical usage.

REQUIREMENT 3.4 *GPOS modifications*: changes to the GPOS must be maintainable.

REQUIREMENT 3.5 *RTOS modifications*: the RTOS kernel must not be modified.

REQUIREMENT 3.6 *SafeG modifications*: the SafeG monitor must not be modified.

Requirement 3.1 refers to the ability to mix the priority levels of RTOS and GPOS activities from a global point of view. Notice that RTOS interrupt handlers still take precedence over any GPOS activity. Requirement 3.2 is necessary to preserve the real-time performance requirements of the RTOS activities even when malicious or defective software is executing in the Non-Secure VCPU. In particular, RTOS tasks with low priority must be protected from execution overruns by GPOS activities executing at a higher global priority. Requirement 3.3 means that the execution overhead introduced by the IS architecture must not affect the overall performance of the system to such an extent that it is no longer usable. Requirement 3.4 is necessary since a GPOS is large and usually evolves very rapidly, thus increasing the maintenance costs. Requirements 3.5 and 3.6 are defined because both the RTOS and SafeG belong to the trusted computing base. Leaving them unmodified simplifies its verification and smooths its maintainability.

3.4 Integrated scheduling architecture

This section describes the integrated scheduling (IS) architecture, whose goal is to support mixing the execution priority levels of RTOS and GPOS activities without compromising the reliability and real-time performance of the RTOS.

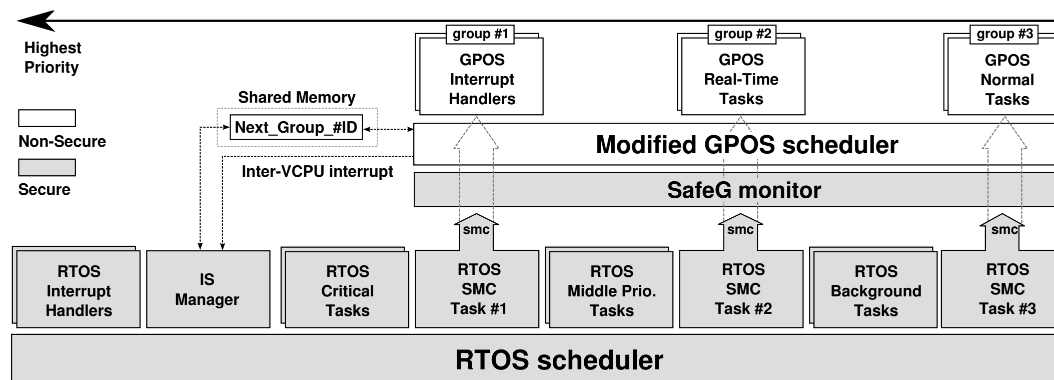


Figure 3.3: Integrated scheduling architecture.

3.4.1 Overview and merits

A straightforward method to implement the mixing of execution priority levels is to extend the SafeG monitor with a few hypercalls, a global scheduler, and support for CPU-time resource reservations. However, this approach has several problems that must be considered: the increase of the number of execution paths inside the SafeG monitor which must be certified; the negative effect on the latency of RTOS interrupts (because the SafeG monitor runs with all interrupts disabled); and the increase in complexity and size of the SafeG monitor.

Instead we take the challenge of designing a novel architecture based on the collaboration between the RTOS user-space and the GPOS kernel scheduler. Implementing the IS architecture at the RTOS user-space has the additional advantage of providing flexibility for the control of the GPOS execution. For example, an RTOS task could easily suspend or resume the execution of the GPOS by calling the existing RTOS application interface.

Figure 3.3 depicts the integrated scheduling (IS) architecture. The main idea is to subdivide GPOS activities into a configurable number of groups that will be scheduled by the RTOS scheduler. In order to accurately map each group of GPOS activities into a different RTOS scheduler's priority level, the IS manager needs to track changes in the GPOS scheduled group. For that reason, the GPOS scheduler is modified to notify the IS manager about such GPOS scheduling events. This collaboration is accomplished by means of an inter-VCPU interrupt—provided by the interrupt controller—and shared memory.

Finally, since the GPOS activities execute in a non-trusted open environment we need to make sure that any execution overrun will not affect the real-time performance of the RTOS tasks. To achieve that, the IS architecture runs each group of GPOS activities under the control of a CPU-time resource reservation.

3.4.2 Groups of GPOS activities

In the IS architecture, system engineers subdivide GPOS activities into several groups, according to their time requirements, and map each group to a different priority level on the RTOS scheduler—which act as the global scheduler. Unlike the idle scheduling approach that has a single RTOS SMC Task with idle priority, the IS architecture allows engineers to assign each group of GPOS activities to a different RTOS SMC Task, executed with a configurable priority. For example, in figure 3.3 GPOS activities are divided in these groups:

- Activities in group #1 (i.e., GPOS interrupt handlers) usually require a very short response time. Therefore they are represented by the RTOS SMC Task #1 which executes at a high priority.
- Activities in group #2 (i.e., GPOS real-time tasks) are usually I/O bound. They spend most of the time waiting for events to arrive, and require good responsiveness to attend to them. For that reason, they are represented by the RTOS SMC Task #2 which executes at a middle priority.
- Activities in group #3 (i.e., GPOS normal tasks) do not have special real-time requirements, and therefore they are represented by the RTOS SMC Task #3 at idle priority. In the case that only this group existed, the IS architecture would be equivalent to the idle scheduling approach.

The IS architecture supports each group of activities being further subdivided into smaller groups—up to a single activity per group—to provide a more fine-grained scheduled system. For the sake of clarity and without loss of generality, the following explanations will use the mentioned 3 groups.

3.4.3 GPOS scheduling events

We define a *GPOS scheduling event* as the instant when the currently running group of GPOS activities is about to be substituted by a different group. We can distinguish 3 types of scheduling events:

- Event type 1: GPOS task from group $\alpha \rightarrow$ GPOS task from group β. This scheduling event occurs when the GPOS switches tasks from different groups.
- Event type 2: GPOS interrupt handler $\rightarrow$ GPOS task. This scheduling event occurs when a GPOS interrupt handler ends and a GPOS task that belongs to a different group is resumed.

```
1  Next_Group_ID : Positive range 1..Max_Groups;
2  Pragma Volatile(Next_Group_ID);
3
4  procedure GPOS_Scheduler_Hook (Next_Task : in TCB_Type) is
5    Prev_Group_ID : Positive range 1..Max_Groups;
6  begin
7    Prev_Group_ID := Next_Group_ID;
8    Next_Group_ID := Get_Group (Next_Task);
9    if (Next_Group_ID /= Prev_Group_ID) then
10     Send_InterVCPU_Interrupt;
11   end if;
12 end GPOS_Scheduler_Hook;
```

Figure 3.4: Pseudocode of the GPOS scheduler hook function.

- Event type 3: GPOS task or RTOS task → GPOS interrupt handler. This scheduling event occurs when a GPOS interrupt handler interrupts the execution of a GPOS task that belongs to a different group or an RTOS task.

The IS architecture needs to keep track of all GPOS scheduling events for the IS manager in the RTOS to resume the RTOS SMC Task representing the next scheduled GPOS group. Notifications of GPOS scheduling events are sent to the RTOS through FIQ interrupts as we explain in the next two sections.

3.4.4 Tracking GPOS scheduling events of type 1 and 2

We inserted a hook function into the GPOS scheduler—which is called at every context switch—in order to notify the RTOS about the occurrence of GPOS scheduling events of type 1 and 2. Figure 3.4 shows the pseudocode of the GPOS scheduler hook function, which executes in GPOS kernel context and receives the control block of the next scheduled task (Next_Task) as a parameter. At every GPOS context switch, the hook function updates a variable named Next_Group_ID with the #ID of the GPOS group where Next_Task belongs to. Then, the hook function checks whether the next group differs from the previously active GPOS group or not (see line 9). In the case that the next GPOS group is different, the hook function sends an inter-VCPU interrupt to the RTOS (see line 10) as a way to notify the RTOS of the occurrence of a GPOS scheduling event of type 1 or 2. The variable Next_Group_ID is placed in inter-VCPU shared memory. It is used by the IS manager—after checking its range—for the management of the state (i.e., suspended or resumed) of the RTOS SMC Tasks. Even if malicious or faulty GPOS software intentionally set the variable Next_Group_ID to the GPOS group with the highest priority, the RTOS real-time performance is still protected by the corresponding CPU-time resource reservations.

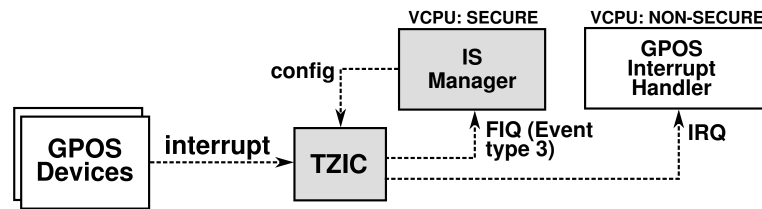


Figure 3.5: Tracking GPOS interrupts.

3.4.5 Tracking GPOS scheduling events of type 3

GPOS scheduling events of type 3 cannot be tracked from the GPOS scheduler since GPOS device interrupts may be raised asynchronously even while the RTOS is executing. In order to track the occurrence of GPOS device interrupts within the IS architecture, we made a subtle modification to the management of device interrupts:

- When a group of GPOS interrupt handlers (i.e., group #1) is not active, the corresponding GPOS device interrupts are configured as FIQ interrupts. Therefore, the interrupt flow from the original SafeG architecture (see § 2.5.2) is modified to notify the RTOS, through an FIQ handler, of the occurrence of GPOS interrupts.

- When the RTOS is notified of the occurrence of a GPOS interrupt, the corresponding GPOS group is activated and all device interrupts associated to that group are configured back as IRQ interrupts, as in the original SafeG architecture.

Figure 3.5 illustrates the management of GPOS device interrupts within the IS architecture. The presented approach allows the RTOS to track the activation of a group of GPOS interrupt handlers even while the RTOS is in execution (i.e., while IRQ interrupts are disabled by setting the I flag to 0). The configuration of the GPOS device interrupts is carried out through the TZIC[32] by a special software agent in the RTOS, called the *Manager task* (see § 3.4.6) and does not require modifications to the source code of the SafeG monitor. The overhead introduced to the RTOS is upper-bounded, but it can disturb the execution of RTOS tasks. Therefore, it must be taken into account when performing the schedulability analysis of the RTOS tasks and interrupt handlers.

3.4.6 IS Manager

The *IS Manager* is an RTOS software agent executed with higher priority (see figure 3.3) than the RTOS SMC Tasks. It is in charge of managing the RTOS SMC Tasks whenever a FIQ interrupt is raised due to the occurrence of a GPOS scheduling event. Figure 3.6 shows the internal architecture of the IS Manager, which is composed of the following elements:

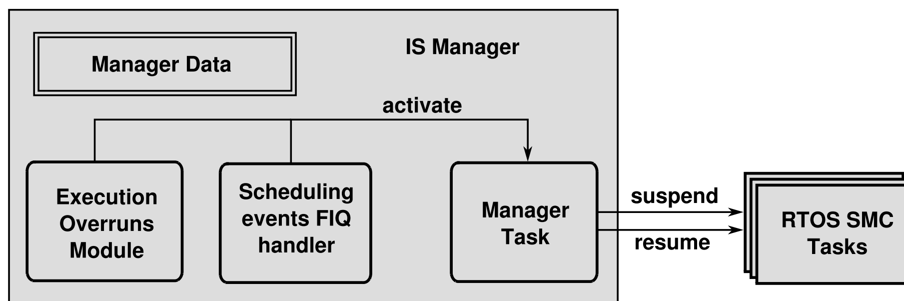


Figure 3.6: IS Manager architecture.

- *Scheduling events FIQ handler*: an RTOS interrupt handler (i.e., FIQ handler) which is raised whenever a GPOS scheduling event occurs. Figure 3.7 shows the pseudocode of the handler. When a GPOS scheduling event occurs, the handler activates the Manager task (which is described below). Then, in the case that the event was of type 3 the handler updates the shared variable Next_Group_ID (see line 5) and configures the GPOS interrupts belonging to that group as IRQ interrupts (see line 6).

- *Manager Task*: an RTOS task aimed at controlling the state of the RTOS SMC Tasks. Figure 3.8 shows the pseudocode of the Manager task. When activated by the Scheduling events FIQ handler, the Manager task calculates which RTOS SMC Task should be activated next (see line 6). Then, if that task is different to the previous one—it could be the same due to execution time overruns as we will see later—the Manager task resumes it and suspends the previous one. The RTOS SMC Task running at idle priority is a special case and remains always active since it cannot affect the real-time performance of the RTOS tasks.

- *Execution Overruns Module*: a software module aimed at limiting the execution time of each RTOS SMC Task in order to preserve the timeliness of the RTOS tasks that run with lower priority. Until now, we assumed that GPOS interrupt handlers and GPOS real-time tasks had a fixed worst-case execution time and inter-arrival period. How-

```
1 procedure Scheduling_Event_FIQ_Handler (The_Event : in Event) is
2 begin
3   Activate_Task (Manager_Task);
4   if (The_Event.Type = 3) then
5     Next_Group_ID := Get_Group (The_Event.Interrupt)
6     Config_Group_Interrupts_As_IRQ (Next_Group_ID);
7   end if;
8 end Scheduling_Event_FIQ_Handler;
```

Figure 3.7: Pseudocode of the scheduling events FIQ handler.

```
1  task body  Manager_Task is
2    Prev_SMC_Task : SMC_Task_Type;
3    Next_SMC_Task : SMC_Task_Type;
4  begin
5    Prev_SMC_Task := Get_Current_SMC_Task(Manager_Data);
6    Next_SMC_Task := Get_Next_SMC_Task(Manager_Data, Next_Group_ID);
7    if (Next_SMC_Task /= Prev_SCM_Task) then
8        Resume_Task (Next_SMC_Task);
9        Suspend_Task (Prev_SMC_Task);
10       Set_Current_SMC_Task (Manager_Data, Next_SMC_Task);
11   end if;
12 end Manager_Task;
```

Figure 3.8: Pseudocode of the Manager task.

ever, a fundamental principle of the SafeG architecture is to consider the GPOS as a non-trusted component. Therefore, we need to guarantee the real-time performance requirements of all RTOS activities even in the case that the GPOS misbehaves or is attacked by malicious software. For that reason, RTOS SMC Tasks are executed under the control of CPU-time resource reservations. A CPU-time resource reservation is a mechanism to limit—and at the same time guarantee—a certain amount of execution time (the reserved budget) within a certain period and at a certain execution priority level. CPU-time resource reservations have been successfully used in previous works[40] to guarantee the execution time of RTOS tasks and the bandwidth of the communication channels in real-time networks. To the best of our knowledge, this is the first time that CPU-time resource reservations have been applied to the integration of a dual-OS system schedule. Each RTOS SMC Task, except the one that runs with idle priority, is assigned to a CPU-time resource reservation which is implemented using the following RTOS functionality:

– *Overrun timer*: an execution time timer that is used to keep track of the execution time consumed by the task associated to a CPU-time resource reservation. When an RTOS SMC Task executes, the budget of its CPU-time resource reservation is consumed. If an RTOS SMC Task exhausts its associated budget, the overrun timer expires. Then, an overrun timer handler activates the Manager task which suspends the associated RTOS SMC Task and activates another lower priority RTOS SMC Task with available budget.
– *Replenishment timer*: a timer used by the Execution Overruns Module to replenish the budget of a CPU-time resource reservation. The way to replenish the budget is dependant on the implementation algorithm. Our implementation is based on the deferrable server algorithm, presented in previous work[49],

which replenishes budget in a periodic fashion. When an RTOS SMC task that was previously suspended after exhausting its budget receives a budget replenishment, the Manager task resumes the RTOS SMC task again if appropriate.

Additionally, the module also contains a set of *discrete* CPU-time resource reservations to bound the number of times that the *Scheduling events FIQ handler* can be raised in a certain period. The algorithm for these discrete CPU-time resource reservations uses a counter variable and a periodic timer. It has the following steps:

1. The counter, initialized to zero, is incremented each time the Scheduling events FIQ handler is raised.
2. When the counter reaches a configurable limit, the inter-VCPU interrupt is masked and GPOS device interrupts are configured as IRQ interrupts.
3. When the associated periodic timer expires, the budget of the discrete CPU-time resource reservation is replenished. To accomplish that, the counter variable is reset to zero; the inter-VCPU interrupt is enabled; and GPOS device interrupts are configured as FIQ interrupts again.

- *Manager Data*: contains information about the RTOS SMC Tasks and is shared among the elements inside the RTOS SMC Tasks manager. The information for each RTOS SMC Task includes its execution state (i.e., suspended or resumed), priority, budget, replenishment period and overrun status.

3.4.7 RTOS protection

Since the GPOS usually executes in an open environment and is prone to software defects due to the size and complexity of its source code, the RTOS must be protected against any GPOS fault or misbehavior. The IS architecture introduces three main components that can be subject to attacks coming from faulty or malicious software running on the GPOS side:

- *GPOS Execution overruns*: groups of GPOS activities may try to overrun the execution time budget assigned to them. RTOS activities are protected against that misbehavior through the Execution Overruns Module explained above.
- *Interrupt attacks*: the GPOS may try to starve RTOS activities by sending a continuous stream of inter-VCPU interrupt requests (events of type 1 and 2) to the RTOS. The GPOS may also try to starve RTOS activities by programming a device to continuously generate interrupts requests (events of type 3). To protect the real-time

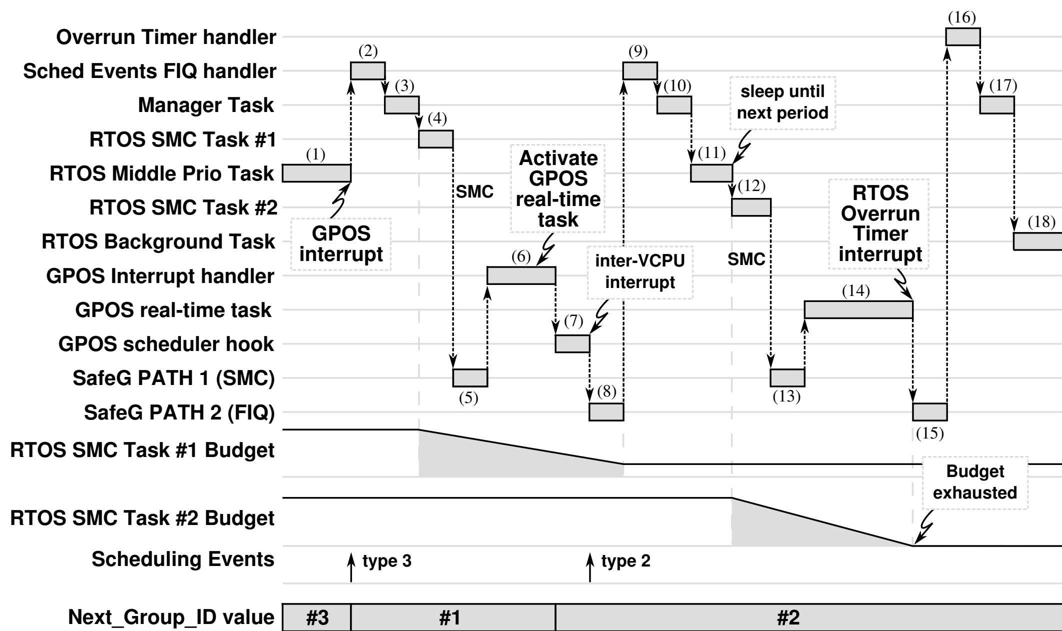


Figure 3.9: Timeline: GPOS real-time task activation.

performance of RTOS activities, the execution of the *scheduling events FIQ handler* in the IS manager must be limited. For that reason, the Execution Overruns Module contains a set of discrete CPU-time resource reservations to bound the number of times that a certain scheduling event interrupt—handled by the *scheduling events FIQ handler*—can occur in a certain period.

- *Shared Variable attacks*: the RTOS IS Manager must check the range of the variable Next_Group_ID each time because it exists in inter-VCPU shared memory, and therefore it could be modified by the GPOS to contain a out-of-range value (i.e., not in 1..Max_Groups). If the GPOS set the variable to the group with highest priority, the execution time of the corresponding GPOS activities will be limited by the Execution Overruns Module.

Finally, it should be noted that although the execution times of the Manager task, the overrun and the replenishment timers are upper-bounded, they must be taken into account for the schedulability analysis of the system.

3.4.8 Example

Figure 3.9 shows a simple example timeline to illustrate the execution flow of the IS architecture. The system is composed of a few tasks and interrupt handlers scheduled with the same priority order as in figure 3.3. A detailed explanation of each step is shown below:

- (1) An RTOS middle priority task executes.
- (2) A GPOS interrupt occurs (event type 3) and it is handled by the RTOS through the Scheduling events FIQ handler. The handler activates the Manager task, updates the Next_Group_ID variable to #1 (i.e., group of GPOS interrupt handlers) and configures GPOS device interrupts as IRQ interrupts.
- (3) The Manager task resumes the execution of the RTOS SMC Task #1.
- (4) Since the RTOS SMC Task #1 has a higher priority than the RTOS middle priority task it resumes its execution. The CPU-time resource reservation associated to the RTOS SMC Task #1 starts consuming its budget.
- (5) The RTOS SMC Task #1 calls SafeG through an SMC instruction. SafeG switches to the GPOS (see § 2.5.2), where the GPOS interrupt handler starts executing.
- (6) The GPOS interrupt handler activates a GPOS real-time task and ends.
- (7) After that, the GPOS scheduler executes and the GPOS scheduler hook function is called. The hook function updates the Next_Group_ID to #2 (i.e., GPOS real-time tasks) and sends a GPOS scheduling event of type 2 to the RTOS through an inter-VCPU FIQ interrupt.
- (8) SafeG context switches back to the RTOS as explained in § 2.5.2.
- (9) In the RTOS, the Scheduling Events FIQ handler receives the inter-VCPU interrupt and activates the Manager task.
- (10) The Manager task suspends the RTOS SMC Task #1 and resumes the RTOS SMC Task #2.
- (11) Since the RTOS middle priority task has precedence over the RTOS SMC Task #2, it resumes its execution.
- (12) When the RTOS middle priority task ends—it goes to sleep until the next period—the RTOS SMC Task #2 is scheduled and the budget associated to its CPU-time resource reservation starts being consumed.
- (13) The RTOS SMC Task #2 calls SafeG through the SMC instruction. SafeG context switches to the GPOS where the GPOS real-time starts executing.
- (14) The GPOS real-time task executes for an excessive amount of time which causes the associated RTOS overrun timer to expire.
- (15) SafeG traps the RTOS overrun timer FIQ and context switches back to the RTOS.
- (16) The interrupt is then handled by the overrun timer handler, inside the Execution Overruns Module. The handler activates the Manager task.
- (17) The Manager task suspends the GPOS RTOS Task #2. However, the Next_Group_ID variable is not modified. This only means that the group of GPOS real-time tasks will

be temporarily represented by a lower priority RTOS SMC Task with available budget (in this case, the one running at idle priority) until the RTOS SMC Task #2 has its budget replenished.

- (18) The RTOS background task resumes its execution since the GPOS real-time task already exhausted its budget.

3.5 Implementation

This section presents details about the implementation of the IS architecture on SafeG using a physical platform. The implementation is distributed together with SafeG through the TOPPERS open source license[30].

3.5.1 Implementation platform

The hardware and software platforms used to implement the IS architecture were as follows:

- RealView Platform Baseboard (PB1176JZF-S)
  - Processor: ARM1176JZF-S[31] development chip at 210MHz.
  - Cache: 32KB
  - Mobile DDR RAM: 128MB (used by the GPOS)
  - PSRAM: 8MB (used by SafeG and the RTOS)
  - TZIC[32] and TZPC[36]
- RTOS: TOPPERS/ASP 1.6 [ASP] with overrun handlers enabled.
- GPOS: Linux 2.6.33 [Linux] with a minimal buildroot[34] filesystem.

Additionally, a Non-Trust version of ASP was adapted to the proposed global scheduling architecture for testing purposes.

3.5.2 Linux kernel modifications

The Linux kernel has been extended with a new module that includes code for the initialization of the shared variable Next_Group_ID. The module also contains information to map groups of GPOS activities into RTOS SMC tasks. The Linux scheduler was modified by inserting a callback function, which can be activated or deactivated from user space (with root privileges) through a boolean value in the Linux debug filesystem (debugfs). The main goal of this callback function is to notify GPOS scheduling events of type 1 and 2; and to update the shared variable Next_Group_ID.

3.5.3 TOPPERS/ASP modifications

The TOPPERS project[30] is a non profit organization that aims at producing high quality open-source software for embedded systems. ASP (Advanced Standard Profile) is one of TOPPERS real-time kernels and follows an improved version of the μITRON4.0 [44] specification. The IS implementation was fully accomplished at application-level space by taking advantage of the following ASP system calls:

- **`ista_ovr`**: used by the budget replenishment handlers to replenish the budget of a certain RTOS SMC task.
- **`isig_sem`**: used to signal the IS Manager from a handler executing in interrupt context (e.g., the overrun handler).
- **`wai_sem`**: used by the IS Manager to wait for a signal.
- **`sus_tsk`**: used by the IS Manager to suspend the execution of an RTOS SMC task.
- **`rsm_tsk`**: used by the IS Manager to resume the execution of an RTOS SMC task.

Dynamic handling of Non-Trust interrupts (i.e., events of type 3) is implemented by taking advantage of the TrustZone Interrupt Controller (TZIC[32]) whose architecture is depicted in figure 3.5.

3.6 Evaluation

In this section, we evaluate the satisfaction of the requirements specified in § 3.3 by our implementation of the IS architecture. We also evaluate the effectiveness of our approach in the solution to the idle scheduling issues presented in the motivational example at § 3.2.

3.6.1 Requirement 3.1: GPOS Latency

We evaluate requirement 3.1 through an experiment in order to confirm that the IS architecture allows enhancing the latency of GPOS activities. The evaluation system is composed of a periodic RTOS task executed at low priority; and a periodic timer interrupt handler on the GPOS executed at high priority (group #1). The experiment consists of measuring the maximum latency of the GPOS timer interrupt handler for several payloads in the RTOS task. The period of the RTOS task is 2 times the payload in all measurements.

The lower part of figure 3.10 shows the experiment results for the IS architecture. We observe that the maximum latency of the GPOS interrupt handler (28μs) remains independent of the payload of the low-priority RTOS task. In contrast, the maximum latency of

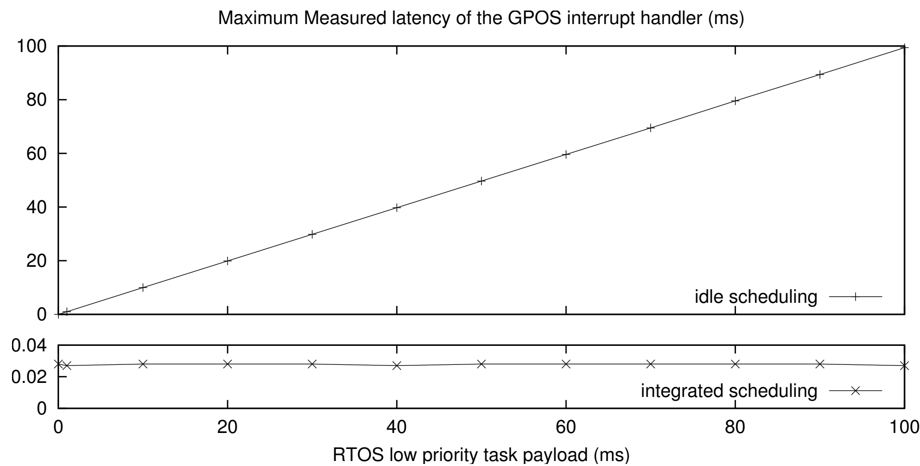


Figure 3.10: Evaluation of requirement 3.1.

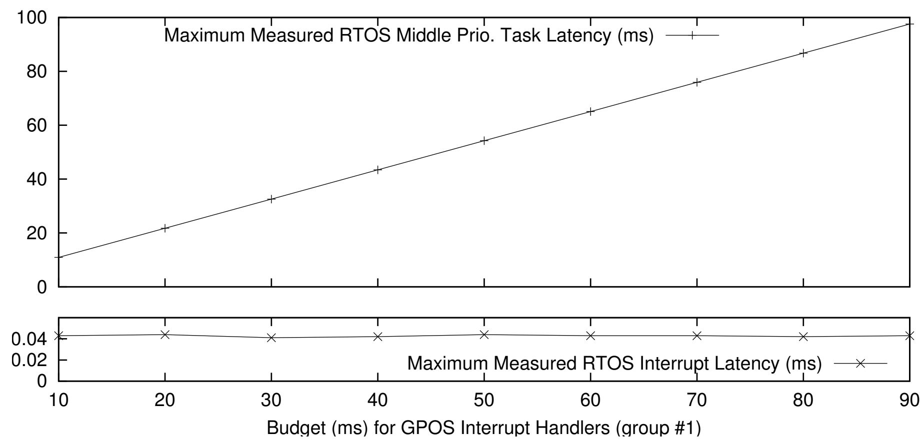


Figure 3.11: Evaluation of requirement 3.2.

the GPOS timer interrupt handler increases proportionally to the payload of the RTOS task for idle scheduling. These results are explained by the fact that the IS architecture allows GPOS interrupt handlers to execute with higher priority than low-priority RTOS tasks.

3.6.2 Requirement 3.2: RTOS timeliness

We carried out an experiment for evaluating the real-time performance of the RTOS (see requirement 3.2) under a worst-case situation where the GPOS misbehaves. The evaluation experiment contains a greedy GPOS interrupt handler—which belongs to the GPOS group #1 in figure 3.3—trying to monopolize the processor by executing an infinite loop. We measured the maximum latency of a periodic RTOS interrupt handler and an RTOS middle priority task for different values of the budget associated to CPU-time resource reservation that limits the execution of the GPOS group #1.

Table 3.1: Overhead of the IS architecture.

type	min(μs)	median(μs)	max(μs)
GPOS scheduling event type 1-2	7	15	28
GPOS scheduling event type 3	6	10	19
Budget replenishments	5	10	18
Budget overruns	5	9	20
SafeG switch time $\simeq 1.7\mu$s			
Linux system tick = 50ms (CONFIG_HZ=20)			

Figure 3.11 shows the results of the experiment. We observe that the maximum measured latency of the RTOS interrupt handler remains independent of the budget assigned to the group of GPOS interrupt handlers. This is coherent with the fact that RTOS interrupt handlers always execute with higher priority than the GPOS.

In contrast, the latency of the RTOS middle priority task experiments a variable delay caused by the blocking time imposed by the execution of the greedy GPOS interrupt handler, which executes with higher priority. However, as figure 3.11 shows, this delay is upper-bounded by the budget associated to the GPOS group #1, and therefore the real-time performance of the RTOS middle priority task is guaranteed.

3.6.3 Requirement 3.3: overhead

We measured the execution time overhead incurred by the IS architecture, as specified by requirement 3.3. Table 3.1 shows the evaluation results obtained by measuring the following 4 sources of overhead:

- *GPOS scheduling event type 1-2*: this is the execution time overhead by GPOS scheduling events of type 1 and 2. In the example depicted by figure 3.9 the worst-case measure would include steps (7) to (12).
- *GPOS scheduling event type 3*: execution time overhead caused by GPOS scheduling events of type 3. The worst-case measure includes steps (2) to (5) in figure 3.9 plus an additional SafeG context switch (in case the GPOS interrupt was raised while the GPOS was executing).
- *Budget replenishments*: execution time overhead caused by the Replenishment timer from the Execution Overruns Module. The worst-case measure includes 2 SafeG context switches, the execution of the replenishment timer handler and the execution of the Manager task.

Table 3.2: Source code lines and binary size increase.

type	new files	code increase	binary size increase
RTOS user	3	155 lines	2716 bytes
RTOS kernel	0	0	0
GPOS kernel	5	88 lines	372 bytes
GPOS user	0	0	0
SafeG monitor	0	0	0

- *Budget overruns*: execution time overhead caused by the overrun timers. The worst-case measure includes steps (15) to (17) in figure 3.9 plus an extra SafeG context switch to the GPOS.

The execution overheads shown in table 3.1 must be taken into account during the schedulability analysis of the RTOS. However, they are small enough for practical application, and therefore we can say that the implementation of the IS architecture is able to satisfy the requirement 3.3.

3.6.4 Requirement 3.4: GPOS modifications

During the implementation of the IS architecture, we extended the Linux kernel with a new driver which includes initialization code and stores information regarding the configuration of the GPOS groups. The module provides a callback function to notify GPOS scheduling events of type 1 and 2, which is inserted as a hook inside the Linux scheduler. The insertion of the hook is facilitated by the **`ftrace`**[50] kernel tracer hooks infrastructure, which reduces the necessary maintenance efforts. As table 3.2 shows, the implementation of the IS architecture on the Linux kernel required a total amount of 88 lines of C source code. 81 of those lines are independent from the Linux kernel while the remaining 7 lines correspond to the hook inserted in the scheduler.

3.6.5 Requirement 3.5: RTOS modifications

Table 3.2 shows that the size of the modifications to the RTOS kernel is zero. This is because we managed to implement the IS architecture completely on RTOS user space by using the TOPPERS/ASP[30] application interface which includes overrun handlers, cyclic handlers, semaphores and an interface to suspend and resume tasks. The implementation required a total of 155 C source code lines and the binary size was increased by 2716 bytes, most of them in the **`.bss`** (uninitialized data) section.

Table 3.3: Tasks in the use case example.

OS	task name	period(ms)	execution time(ms)
ASP	Robot Control task	10	5
ASP	Logger task	4000	500
Linux	mplayer task	41	12

3.6.6 Requirement 3.6: SafeG modifications

In this implementation the SafeG monitor was completely unmodified and therefore requirement 3.6 was satisfied. The main reason for which SafeG may need to be extended in a future implementation is that the chip did not support inter-VCPU interrupts. In that case, a new SMC call should be added for SafeG to emulate inter-VCPU interrupts.

3.6.7 Use case example

In order to prove the practical applicability and effectiveness of the proposed architecture we built the system previously described in § 3.2, and measured its performance under the idle scheduling and the integrated scheduling approach. The main difference between both approaches is the fact that on the IS approach, Linux interrupts and the mplayer task are scheduled with higher priority than the ASP logger task; whereas in the idle scheduling approach they are scheduled with lower priority.

The period and execution time of each task in the system are shown in table 3.3 (deadlines equal to periods). Additionally, a CPU-time resource reservation with 12ms of budget and 41ms of replenishment period was used to prevent the ASP logger task from starving on the IS approach. The theoretical worst-case response time of the mplayer task under each scheduling approach can be calculated by using real-time response analysis[47, 26]:

$$W_i(t) = C_i + \left\lceil \frac{t}{T_{i-1}} \right\rceil \cdot C_{i-1} + ... + \left\lceil \frac{t}{T_1} \right\rceil \cdot C_1 \tag{3.1}$$

$$WCRT_{mplayer_{idle}} = 12 + \left\lceil \frac{1027}{10} \right\rceil \cdot 5 + \left\lceil \frac{1027}{4000} \right\rceil \cdot 500 = 1027ms \tag{3.2}$$

$$WCRT_{mplayer_{is}} = 12 + \left\lceil \frac{27}{10} \right\rceil \cdot 5 = 27ms \tag{3.3}$$

Where $W_i(t)$ is the amount of work of priority P_i or higher started before time t; C_i is the execution time for task i; and T_i is the period of task i. The equation is solved by iterating the value of t until two successive steps yield the same result for $W_i(t)$, which equals

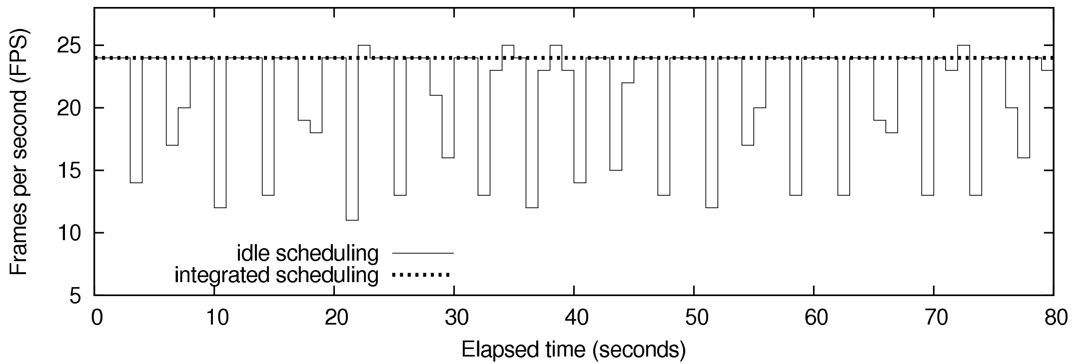


Figure 3.12: Frames per second in each scheduling approach.

to the worst-case response time for task i (i.e., $WCRT_i$). The mplayer task is schedulable under the IS architecture because its worst-case response time (27ms) is smaller than its deadline (41ms). In contrast, its worst-case response time under the idle scheduling approach (1027ms) is greater than its deadline (41ms), and therefore not schedulable. We can also see that the logger task remains schedulable under the IS architecture:

$$WCRT_{logger_{is}} = 500 + \left\lceil \frac{2418}{10} \right\rceil \cdot 5 + \left\lceil \frac{2418}{41} \right\rceil \cdot 12 = 2418ms < 4000ms \tag{3.4}$$

Figure 3.12 shows the dynamic frame rate at which the video is played under both scheduling approaches. We measured the dynamic frame rate of the video by using the *movbench* utilities available at [48]. We observe that the frame rate under the idle scheduling approach experiments strong drops approximately every 4000ms (i.e., the period of the logger task) which are caused by the blocking time imposed by the execution of the ASP logger task. The average frame rate is reduced from 24 to 20 frames-per-second, and most importantly the user's video watching experience gets dramatically worsened.

In contrast, we observe that under the IS architecture the video is played smoothly at a constant rate of 24fps. We also observed that this was accomplished without causing any deadline miss to the ASP robot control task (i.e., the robot never fell down) nor starvation to the ASP logger task. From these observations, we shall conclude that the proposed IS architecture was effective at enhancing the responsiveness of the mplayer task without damaging the real-time performance of the ASP activities.

3.7 Related work

There is a large amount of literature proposing different dual-OS mechanisms to enable the concurrent execution of a GPOS and an RTOS under time and memory isolation conditions. However, research on dual-OS scheduling is rather scarce and most of it can be classified in the following groups:

- *Idle scheduling*: dual-OS monitors that schedule the GPOS as the RTOS idle task belong to this group. A few examples are RTAI[39], Linux on ITRON[10] and more recently, MobiVMM[6]. [51] proposes a combination of a time-driven, priority-based and proportionally shared scheduler. However, in their proposal the RTOS is always assigned a static high priority, and therefore providing responsiveness to the GPOS is rather difficult.

- *Compositional scheduling*: this includes dual-OS monitors that schedule the RTOS and GPOS using time-based compositional scheduling frameworks. An example is the XTRATUM[11] hypervisor whose scheduler is based on ARINC-653[24]. This approach allows for a great degree of time isolation between the guest OSs. However, it is not suitable for event-driven processing, such as interrupts, that require very short latencies.

- *Fair scheduling*: a typical example is the XEN[2, 16] credit scheduler which enables sharing the CPU proportionally between the guest OS. XEN has several mechanisms [45] for improving the responsiveness of I/O bound guest OSs but it is not able to cope with the hard real-time requirements of an RTOS.

The idle-scheduling problem described in § 3.2 was also identified by [46], where a task grain scheduling algorithm for a virtualized embedded system was presented. The architecture proposed in [46] uses the L4-embedded microkernel as a hypervisor running para-virtualized versions of Linux (Wombat) and TOPPERS/JSP (L4/TOPPERS). In order to implement task grain scheduling, each of the guest operating systems notifies the priority of the running task to a global scheduler. However, their approach does not protect RTOS tasks real-time properties from GPOS misbehavior and is not capable of mixing the priority of GPOS interrupt handlers with RTOS tasks. Also, the architecture proposed in that approach requires an extra global scheduler while in the IS architecture the RTOS scheduler plays that role.

3.8 Conclusions

In this chapter, we replaced the idle scheduling principle from the original SafeG architecture by an integrated scheduling architecture that allows mixing the execution priority levels of RTOS and GPOS activities. The architecture is based on the collaboration of an RTOS user-space library and the GPOS scheduler. We showed that the IS architecture is useful for enhancing the responsiveness of GPOS activities with soft real-time requirements (e.g., interrupt handlers or multimedia applications). At the same time, the real-time performance requirements of the RTOS are guaranteed thanks to the use of overrun control mechanisms. We implemented the architecture on a physical platform and evaluated it. The results of the evaluation showed that all the requirements defined in § 3.3 were satisfied. Additionally, we built a use case example that proved the practical applicability and effectiveness of the proposed approach in a real-world application. The source code of the IS architecture and usage examples are distributed along with SafeG through the TOPPERS open source license[30].

Chapter 4

Dual-OS communications

Dual-OS communications allow a real-time operating system (RTOS) and a general-purpose operating system (GPOS)—sharing the same processor through virtualization—to collaborate in complex distributed applications. However, they also introduce new threats to the reliability of the RTOS that must be considered. In this chapter, we propose a novel dual-OS communications approach able to accomplish efficient communications without compromising the reliability of the RTOS.

4.1 Introduction

Although the mere execution of the RTOS and the GPOS in isolation may satisfy the requirements of some systems, support for communication between both OSs opens the door for new applications with higher sophistication. However, dual-OS communications also introduce new threats to the reliability of the RTOS that must be considered. Some existing dual-OS communication systems[14, 24, 11, 25] are able to cope with most of the aforementioned reliability threats. However, they all essentially follow the same traditional approach (see figure 4.2(a)) based on extending the dual-OS virtualization layer with additional communication primitives. Although the traditional approach simplifies synchronizing both OSs and protecting communication structures, it has several efficiency drawbacks to consider, such as the use of unnecessary data copies and context switches.

The main contribution of this work is a new dual-OS communications architecture able to provide efficient communications without compromising the reliability of the RTOS. We implemented it on a physical platform using the highly reliable SafeG dual-OS system (see § 2.5). Our implementation leverages ARM TrustZone hardware security extensions[29] to guarantee the memory protection and timeliness of the RTOS at a rather low overhead.

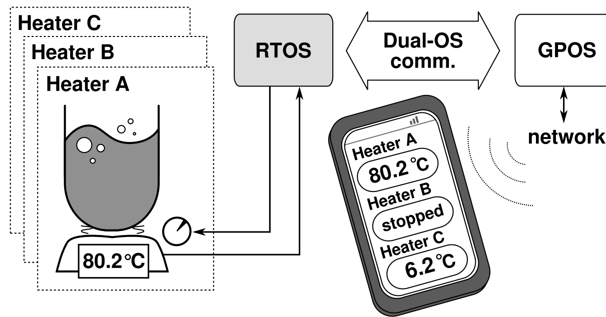


Figure 4.1: Dual-OS communications example.

We evaluated our approach through several experiments; and compared the results with the traditional approach used in previous dual-OS systems. From the evaluation results, we observed that the proposed architecture is indeed effective at minimizing the communication overhead while satisfying the strict reliability requirements of the RTOS.

The remainder of this chapter is structured as follows. § 4.2 reviews background knowledge for better understanding the contents of this chapter. We also introduce related work and detail the efficiency and reliability problems that we encountered in previous approaches to dual-OS communications. § 4.3 provides a list of requirements and assumptions for the design of our dual-OS communications architecture. § 4.4 constitutes the core of this chapter and describes our approach to dual-OS communications. § 4.5 details the implementation of our architecture and § 4.6 presents the results of the evaluation and discusses the satisfaction of the requirements listed in § 4.3. Finally, § 4.7 concludes the chapter.

4.2 Background

4.2.1 Dual-OS communications

A dual-OS communications system is a set of methods for the exchange of information between RTOS and GPOS applications—running on a dual-OS system—which opens the possibilities for new applications with higher sophistication:

- RTOS ⇒ GPOS communications are typically used for the RTOS to leverage the rich functionality of the GPOS. Figure 4.1 shows an example where dual-OS communications are used for the RTOS to report the status of a group of heating devices to the GPOS, which is in turn connected to a remote user through a network.

- GPOS ⇒ RTOS communications are typically used for the GPOS to request a service that only the RTOS can provide. For example, the RTOS may provide secu-

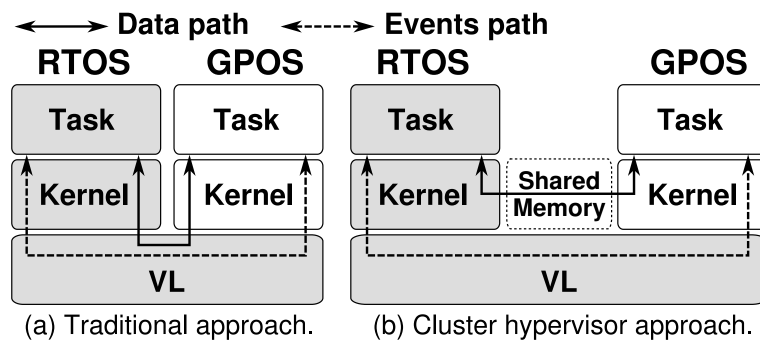


Figure 4.2: Previous communication approaches.

rity services[12, 29] (e.g., digital rights or authentication services) that use cryptographic keys stored on secure memory that are not accessible by the GPOS. Another application is device sharing[57, 58] which cannot be accomplished through GPOS devices—even if both OSs have access to them—because otherwise the GPOS could access sensible data belonging to the RTOS; or change the device's configuration.

4.2.2 Related work

In spite of its benefits, dual-OS communications also introduce new threats to the reliability of the RTOS: message overload attacks[52]; user and control data corruption attacks[53]; memory faults caused by shared pages being removed[59]; or unbounded waits caused by the non-cooperation of the GPOS are just a few examples.

There already exist several dual-OS communication systems[14, 24, 11, 25] capable of addressing most of the aforementioned reliability threats. However, they all essentially follow the same rather conservative traditional approach (see figure 4.2(a)) that requires the communications data path to traverse the VL, which needs to be extended with additional communication primitives. This approach simplifies the synchronization between both OSs—typically by disabling interrupts inside the VL—in the access to the control data used for communications; and is useful to protect it against GPOS attacks. However, it also has several efficiency drawbacks that must be considered, such as unnecessary data copies and context switches[66, 61].

In parallel, several works[54, 55, 56] have addressed the demanding efficiency requirements of inter-OS communications for multiple-guest hypervisors (e.g., XEN[16]) used in enterprise cluster computing. Most of them exploit the use of kernel shared memory for reducing the number of data copies (see figure 4.2(b)). Unfortunately, none of these hypervisors are able to guarantee the high reliability requirements of a dual-OS system.

4.3 Requirements and assumptions

This section presents a list of requirements and assumptions for the design of a reliable and efficient dual-OS communications system. The satisfaction of these requirements will be discussed in § 4.6.

4.3.1 Reliability requirements

REQUIREMENT 4.1 *Memory isolation*: TCB memory must be protected against any access—accidental or malicious—by the GPOS. Shared memory used for communications must only be accessible by the RTOS and GPOS privileged software, which should pass a software quality control for increasing its trustworthiness.

REQUIREMENT 4.2 *Shared control data*: control data shared by both OSs must be validated (e.g., using range checking) by the RTOS, and protected against further malicious modifications by the GPOS.

REQUIREMENT 4.3 *Real-time*: the timeliness of RTOS interrupt handlers and tasks must be guaranteed. In particular, it must be protected against message overload attacks coming from the GPOS.

REQUIREMENT 4.4 *Memory faults*: the architecture must guarantee that the RTOS will not access non-existent memory—causing the corresponding memory faults—even if GPOS tasks are swapped out to virtual storage or shared memory used for communications is unmapped by the GPOS.

REQUIREMENT 4.5 *Unbounded blocking*: the architecture must guarantee that RTOS tasks will not suffer unbounded waiting times caused by the non-cooperation of the GPOS.

REQUIREMENT 4.6 *Code modifications*: the architecture must not impose modifications to the RTOS kernel or the VL to avoid reissuing a new verification process, and for improving the maintainability of the architecture. The GPOS kernel can be extended with a driver (e.g. to handle communication events) but its core code must not be modified due to maintenance reasons.

4.3.2 Efficiency requirements

REQUIREMENT 4.7 *Throughput*: the architecture must minimize the overhead caused by unnecessary data copies and context switches; and the use of costly protocol stacks.

REQUIREMENT 4.8 *Memory size*: the amount of memory used for dual-OS communications must be minimized.

Table 4.1: Requirements Vs. Our design choices.

Type	Requirement name	Number	Evaluation	Our design choices
Reliability	Memory isolation	4.1	QL	Three trustworthiness levels.
	Shared control data	4.2	QL/QN	Four steps update algorithm.
	Real-time	4.3	QN	Message rate limiting.
	Memory faults	4.4	QL	Reserve shared memory.
	Unbounded blocking	4.5	QL	Timeouts and non-blocking.
	Code modifications	4.6	QL/QN	User-level library.
Efficiency	Throughput	4.7	QN	Shared memory; Filters.
	Memory size	4.8	QN	Static configuration interface.
	Interface	4.9	QL	Shared memory & events; RPCs; sampling messages.

QL=Qualitative QN=Quantitative

REQUIREMENT 4.9 *Interface*: the communications interface must be suitable for implementing communication patterns—such as remote procedure calls (RPC) or sampling communication—typically present in common embedded systems.

4.3.3 Assumptions

(a) *Static*: all RTOS communication resources can be statically allocated for reliability reasons.

(b) *Transparency*: the user interface does not require being transparent to the user.

(c) *Events driver*: the GPOS allows implementing a driver to send or receive software interrupts.

(d) *Raw user data*: validating the raw contents of user messages is out of the scope of this chapter.

(e) *Verified TCB*: we assume that software belonging to the TCB has been correctly verified and does not have any defects.

4.4 Communications architecture

Table 4.1 summarizes the list of requirements presented in § 4.3, and introduces the corresponding design choices followed by the proposed dual-OS communications architecture (hereafter *dualoscom* architecture). This section describes the design of the dualoscom architecture, which will be evaluated in § 4.6 against the mentioned requirements and compared against the traditional approach depicted in figure 4.2(a).

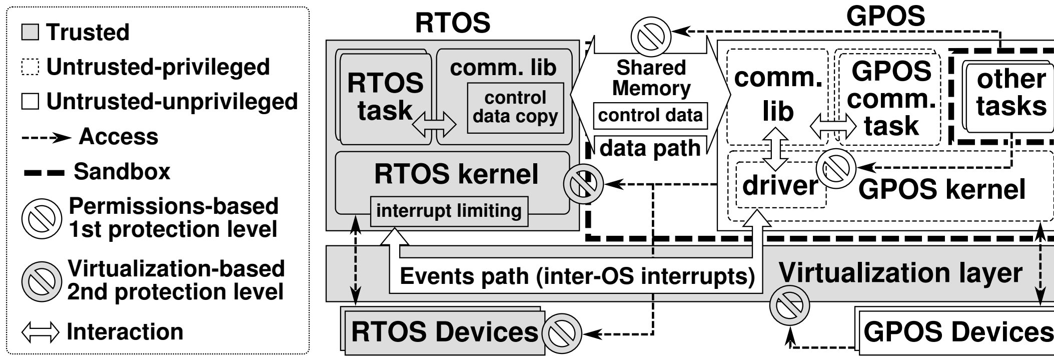


Figure 4.3: The proposed dual-OS communications architecture.

4.4.1 Satisfying reliability requirements

- *Memory isolation* (requirement 4.1): the dualoscom architecture relies on user-level shared memory for implementing dual-OS communications efficiently (see figure 4.3). In contrast to the traditional approach (see figure 4.2(a)) dualoscom control data structures are not protected by the VL. Instead, we divide GPOS tasks into two groups: GPOS communicating tasks with the privilege of accessing the shared memory region; and other GPOS tasks without such privilege. GPOS communicating tasks are created and thoroughly tested by the dual-OS system engineer during the development phase. In contrast, the other GPOS tasks may include malicious or buggy applications installed by the user during the lifetime of the system, and are expected to be less trustworthy. This reasoning leads to the existence of three trustworthiness levels (trusted, untrusted-privileged and untrusted-unprivileged), which are separated by the two protection sandboxes illustrated by figure 4.3. TCB memory (i.e., trusted memory for the RTOS and the VL) is protected against any GPOS access by the VL protection level; and communications shared memory is protected against untrustworthy GPOS tasks by the permissions-based protection level.

- *Shared control data* (requirement 4.2): if the permissions-based protection level is broken (e.g., by exploiting a GPOS kernel bug), untrusted-unprivileged GPOS tasks can attack the second virtualization-based protection level by maliciously modifying the shared control data. In order to protect the RTOS against such modifications—for example, to avoid dereferencing a null pointer—all updates by the RTOS to the shared control data are made in four steps: copy the required control data to the RTOS memory; validate it by range-checking; update it according to the current operation (e.g., enqueue); and finally, copy the modified control data back to shared memory. Validating the raw contents of user messages is out of the scope of this chapter.

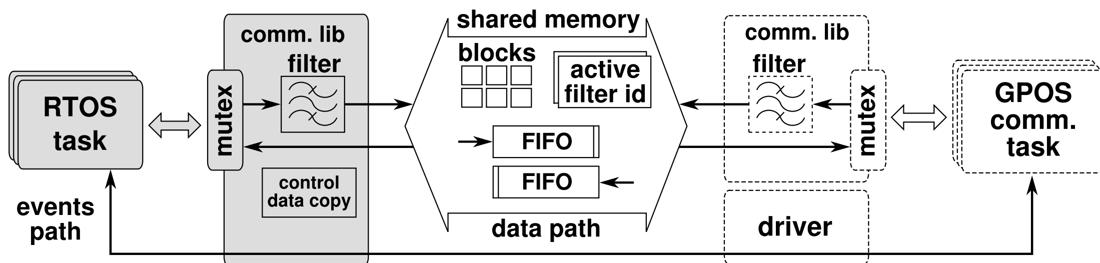


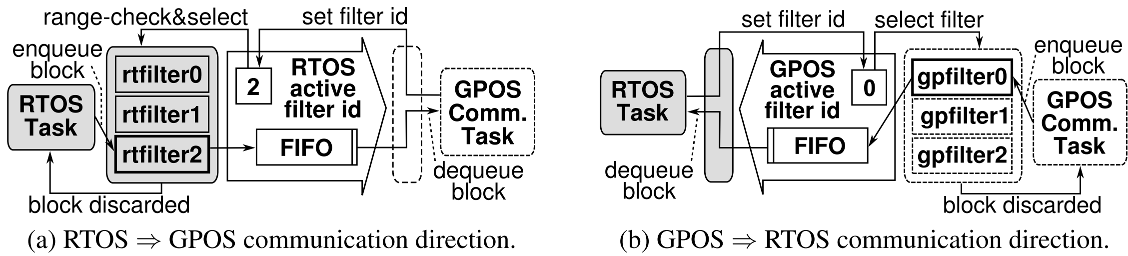
Figure 4.4: Elements of a dualoscom communication channel.

- *Real-Time* (requirement 4.3): another way for a malicious GPOS application to attempt breaking the second protection level is by sending an excessive amount of messages to the RTOS. The dualoscom architecture splits the transmission of messages in two parts: the data path and the events path. The data path involves non-blocking operations to enqueue or dequeue blocks of data; and is implemented through lock-free bidirectional FIFOs that exist in shared memory (see figure 4.4). The events path involves asynchronous notifications (implemented through inter-OS interrupts) and wait-event operations that may block the calling task until a timeout expires. This separation allows tasks to communicate using both polling or event-driven communication patterns. However, in order to protect the timeliness of RTOS activities the rate of GPOS $\Rightarrow$ RTOS message interrupts must be limited. The dualoscom architecture supports two message interrupt limiting algorithms presented in previous work[52]: the *strict* message interrupt limiter which enforces the minimum inter-arrival time between interrupts; and the *bursty* message interrupt rate limiter which enforces a maximum burst size and a maximum arrival rate. Finally, it is the responsibility of RTOS applications not to poll for new GPOS messages in an endless loop.

- *Memory faults* (requirement 4.4): to guarantee that the RTOS will never try to access non-existent or unmapped memory, the shared memory region used for communications is statically allocated at configuration time.

- *Unbounded blocking* (requirement 4.5): to avoid a situation in which RTOS tasks could wait for a GPOS message for an unbounded amount of time, a timeout can be specified in all blocking operations of the events path. Non-blocking operations never perform retries, and return an error code instead when there is contention.

- *Code modifications* (requirement 4.6): to avoid modifying the RTOS kernel or the VL, the data path is carried out by the dualoscom communications library (comm. lib) at user level. The events path and the message interrupt rate limiting mechanisms are



(a) RTOS ⇒ GPOS communication direction. (b) GPOS ⇒ RTOS communication direction.

Figure 4.5: Behavior of the filtering functionality in both communication directions.

implemented through the RTOS application interface (API). In contrast, to implement event operations (e.g., waiting or sending an event), the GPOS kernel requires being extended with a communications driver.

4.4.2 Satisfying efficiency requirements

- *Throughput* (requirement 4.7): to minimize the overhead caused by unnecessary data copies and context switches, all data path communications occur at user level through shared memory. To reduce the overhead caused by the events path, applications can choose to use a single event to notify the transmission of several messages, thus reducing the number of context switches per message. This is supported by splitting the communications interface between the transmission of data and events. There are two more mechanisms for reducing the overhead caused by unnecessary context switches: *filters* and *non-synchronized accesses*. Filters are functions that execute on the sender side of a channel (see figure 4.4 and figure 4.5) and are used for discarding the transmission of messages when they are not needed by the receiver (e.g., if a variable has not changed since the last time it was received). The access to a channel can be configured to be synchronized (e.g., through a priority ceiling mutex) or non-synchronized. This allows avoiding the execution time overhead associated to access synchronization when only a single task is supposed to access the channel.

- *Memory size* (requirement 4.8): to minimize the amount of memory used by dual-OS communications, all channel parameters can be configured. The configuration parameters of a channel include: the number of blocks and their size; the use of synchronized accesses; and its associated filters.

- *Interface* (requirement 4.9): the runtime interface to the dualoscom architecture supports shared memory blocks and asynchronous event notifications. By combining

them it is possible to build more complex communication patterns such as RPCs or sampling messages[24, 27]. See § 4.4.4 and § 4.4.5 for details.

4.4.3 Communication channels

A channel is a communication entity by means of which RTOS and GPOS untrusted-privileged tasks can exchange information. Figure 4.4 depicts the main structures of a communication channel, which is composed of the following elements:

- *Blocks*: a *block* is a piece of shared memory used to send data. Each channel contains a pool of a configurable number of blocks. All the blocks in a channel have a fixed size, which is also configurable. Blocks must be explicitly allocated before being used. They can be sent in both directions (i.e., RTOS⇔GPOS) and they can be released back to the channel's pool either by the sender or the receiver.

- FIFOs: a FIFO (First-In-First-Out) queue is a data structure used to deliver blocks in the same order they were enqueued. Each channel contains exactly two FIFOs, one for each communication direction. A FIFO has a number of elements equal to the number of blocks in the channel. Each enqueued element consists of a block identifier that was previously allocated and enqueued by a sender. A FIFO queue can be easily implemented using a lock-free algorithm if all of its operations are serialized. For that reason, the GPOS does not need to disable RTOS interrupts (e.g., for synchronization purposes) which is forbidden by the VL.

- *Filters*: a *filter* is a function that receives a block's buffer and size, and returns a boolean to indicate whether the corresponding block should be sent or not. Filters are used for discarding the transmission of a block (i.e., before it is enqueued) depending on its contents. They are used to avoid unnecessary communication overhead[27]. Figure 4.5 depicts the dualoscom filtering functionality. Each channel contains two active filter functions (e.g., **`rtfilter2`** and **`gpfilter0`**), one for each communication direction. Filters used in the RTOS ⇒ GPOS communication direction (e.g., **`rtfilter#`**) execute on the RTOS, and therefore must follow the same formal verification process as other components in the TCB. In contrast, GPOS ⇒ RTOS filters (e.g., **`gpfilter#`**) execute on the GPOS untrusted-privileged user space. Compared to untrusted-unprivileged software, GPOS filters must follow a software quality control. However, they are allowed to assume that data sent by the RTOS is valid. The source code of RTOS and GPOS filters is statically provided by the dual-OS system

engineer during the build process (see figure 4.6), and their contents cannot be modified during the execution of the system. Instead, each channel contains two variables (RTOS and GPOS *active filter id*), which are identifier numbers for indicating the currently active filter on each communication direction (e.g., 2 in figure 4.5(a) and 0 in figure 4.5(b)). While filter functions are located in the same memory region as the operating system where they execute, active filter id variables are located in shared memory. Receiver tasks can select the active filter at runtime by using a filter identifier—or a null value if no filtering is required—as illustrated by figure 4.5. Filter identifiers are automatically allocated by the dualoscom configurator tool during the configuration phase (see figure 4.6), and are internally represented as natural integers. Before a block is enqueued to a channel, the dualoscom library reads its *active filter id* number from shared memory, and executes the associated active filter function (e.g., **`rtfilter2`** and **`gpfilter0`**) on it. If the filter function returns *true*, the block is enqueued to the FIFO; otherwise an error code is returned to the user, indicating that the block was discarded. Note that in figure 4.5(a), a malicious GPOS task can potentially set a corrupted the active filter id (47). The RTOS library must always validate the range of the active filter id variable before using it to select a filter function for execution. Note that any value inside the range is valid, and therefore filters must be prepared to handle any block that is sent through that channel direction.

- *Events*: an *event* is a method for sending asynchronous notifications between the RTOS and the GPOS. Events can be sent in both directions and they are not queued, meaning that they must be acknowledged by the receiver before a new event can be sent. Events are sent independently to the process of enqueueing blocks. This allows senders to enqueue several blocks before notifying the receivers.

- *Mutexes*: a *mutex* is a mechanism used for serializing the access of tasks to a channel within the same OS. Channels can have up to two mutexes, one for each communication direction. Each mutex can be removed at configuration time—for minimizing the synchronization overhead—if access contention is not expected.

4.4.4 Dualoscom interface

The dualoscom build process

Figure 4.6 illustrates the dualoscom build process through the heating devices example from figure 4.1. As it is common practice in most RTOSs[30], dualoscom provides a configuration interface which allows all of its structures to be allocated statically. This is necessary

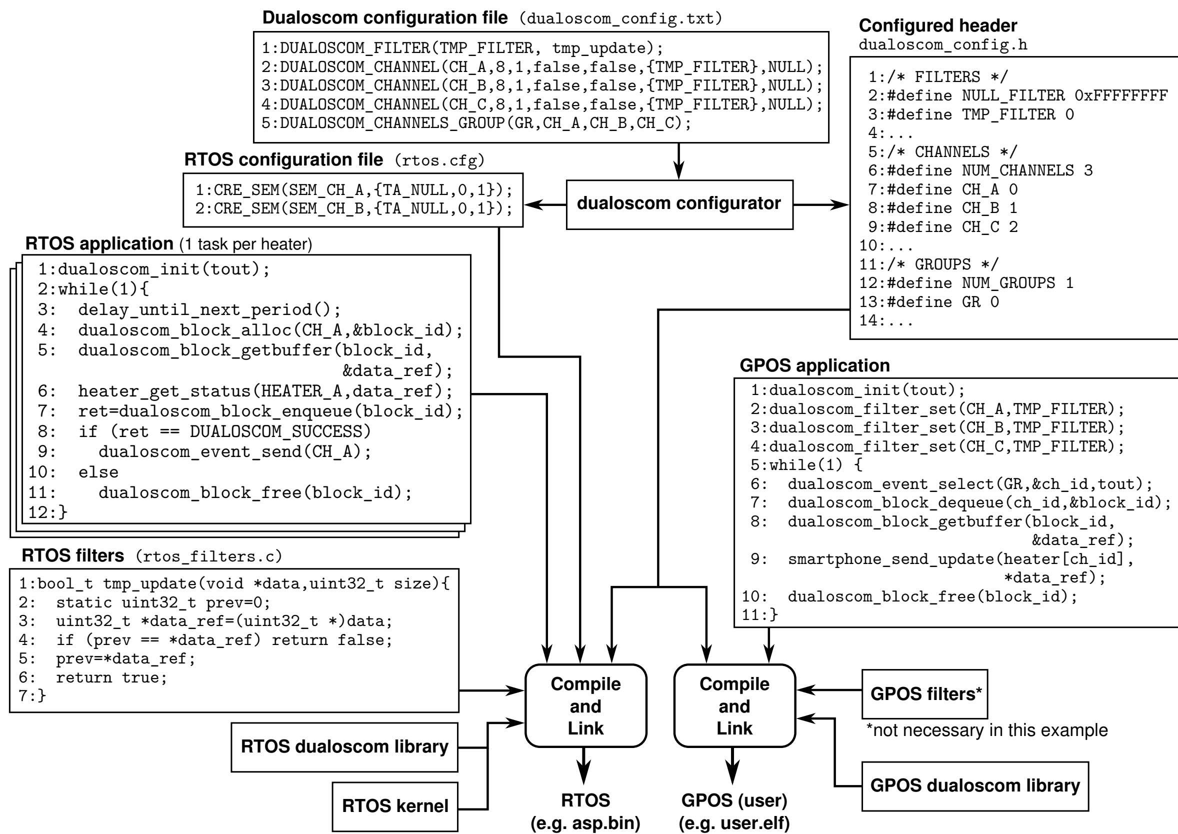


Figure 4.6: The dualoscom build process.

for guaranteeing the reliability of the TCB and it allows minimizing its memory and execution time overhead. First, the dual-OS system engineer provides a configuration file (e.g., **`dualoscom_config.txt`**) containing a channel declaration for each heating device. Then, the configuration file is parsed by the *dualoscom configurator* tool, which generates a configured header file (e.g., **`dualoscom_config.h`**) with constant definitions (e.g., identifiers); and an RTOS configuration file (e.g., **`rtos.cfg`**) with static declarations—note that RTOS resources are typically allocated statically for reliability reasons[26]. Finally, the dual-OS system engineer provides the RTOS and GPOS communicating applications, and the filter functions if necessary. These applications use a *runtime interface* whose syntax, together with the syntax of the configuration interface, is described below.

The build process ends with the generation of two binaries: the RTOS bare-metal binary (e.g., **`asp.bin`**), which contains the RTOS kernel, dualoscom library and application (the RTOS kernel is typically linked to the user application for performance reasons); and the GPOS user application (e.g., **`user.elf`**) which is linked to the dualoscom library. The GPOS kernel is patched with a dualoscom driver, which receives all configuration parameters from user-space at initialization, and therefore it only needs to be built once.

Configuration interface

The configuration interface uses the next syntax:

DUALOSCOM_FILTER(): used to declare a filter. It accepts the following parameters:

- **FILTER_NAME**: a name for the filter. The dualoscom configurator generates a constant with the same name (e.g., **TMP_FILTER** in figure 4.6) which is used as an identifier for the receiving tasks to select the active filter at runtime.
- **filter**: the name of a function which takes a block's buffer and size as input parameters, and returns a boolean value (see the **tmp_update** function in figure 4.6 for an example). If the return value is *true*, the block will be enqueued; otherwise it will be discarded. The body of the function is written and tested (i.e., it must follow the same software quality controls as any other software executed in the same trustworthiness level) by the dual-OS system engineer before the build process starts.

DUALOSCOM_CHANNEL(): used to declare a channel. It accepts these parameters:

- **CHANNEL_NAME**: a name for the channel. After configuration, the same name (e.g., **CH_A**) can be used to identify the channel.
- **num_blocks**: the number of blocks.
- **block_size**: the block size in memory words.
- **mutexes**: two booleans to indicate if mutual exclusion is used at each end.
- **rtos_filters**: list of **FILTER_NAME** values to declare which filters can be selected on the RTOS end of this channel. The value **NULL** can be used to declare no filter. By default, there is no active filter at initialization.
- **gpos_filters**: list of **FILTER_NAME** values to declare which filters can be selected on the GPOS end of this channel. The value **NULL** can be used to declare no filter. By default, there is no active filter at initialization.

DUALOSCOM_CHANNELS_GROUP(): used to declare a group of channels, to allow waiting for events on several channels at the same time.

- **GROUP_NAME**: a name for the group of channels. After configuration, the same name can be used to identify the group.
- **channels**: a list of **CHANNEL_NAME** values that must match the values used during the declaration of channels.

Runtime interface

The runtime interface is a set of functions for the RTOS and GPOS applications to communicate between each other at runtime. All functions return **DUALOSCOM_SUCCESS** upon success and one of the following errors upon failure:

ERROR 1 **DUALOSCOM_NOPERM**: not enough permissions.
ERROR 2 **DUALOSCOM_NOINIT**: the communications system is not initialized yet.
ERROR 3 **DUALOSCOM_PARAM**: incorrect parameter.
ERROR 4 **DUALOSCOM_FULL**: there are no free blocks.
ERROR 5 **DUALOSCOM_ENQ**: the block is enqueued.
ERROR 6 **DUALOSCOM_FILTER**: the block was discarded.
ERROR 7 **DUALOSCOM_EMPTY**: no block is enqueued.
ERROR 8 **DUALOSCOM_ALLOC**: the block is not allocated.
ERROR 9 **DUALOSCOM_TIMEOUT**: a timeout occurred.

The runtime interface is composed of the following list of functions. Note that the prefix **dualoscom_** has been omitted from each function for the sake of shortness.

Initialization functions

- **init(timeout)**: initializes the dualoscom system. The initialization protocol and the timeout units are implementation-dependent. May return errors 1, 3, and 9.

Block management functions

- **block_alloc(chan_id, &block_id)**: it allocates a block from a channel's pool. This function never blocks the calling task. May return errors 1, 2, 3, and 4.
- **block_free(chan_id, block_id)**: releases a block back to the channel's pool where it belongs. May return errors 1, 2, 3, and 8.
- **block_getbuffer(chan_id, block_id, &buffer_p, &size)**: to obtain a pointer to the beginning of the memory region of a block. May return errors 1, 2, 3, and 8.
- **block_enqueue(chan_id, block_id)**: enqueues a block to a channel's FIFO. May return errors 1, 2, 3, 6, and 8.
- **block_dequeue(chan_id, &block_id)**: dequeues a block from a channel's FIFO. This function never blocks the calling task. May return errors 1, 2, 3, 7, and 8.

Event management functions

- `event_send(chan_id)`: sends a channel event notification. If a notification had already been sent but not acknowledged by the receiver, then the function returns `DUALOSCOM_SUCCESS`. Otherwise it may return errors 1, 2, and 3.
- `event_wait(chan_id, timeout)`: this function makes the calling task wait for an event notification on a channel. If an event was pending, the function acknowledges it and returns immediately. Otherwise, the calling task is put in waiting state until an event arrives or a timeout occurs. The timeout units are implementation-dependent. May return errors 1, 2, 3, and 9.
- `event_select(group_id, &chan_id, timeout)`: this function makes the calling task wait for an event notification on a specific group of channels at the same time. If an event on one of the channels was pending, the function acknowledges it and returns immediately. Otherwise, the calling task is put in waiting state until an event arrives or a timeout occurs. The timeout units are implementation-dependent. May return errors 1, 2, 3, and 9.

Filter management functions

- `filter_set(chan_id, filter_id)`: used by receiver tasks to select one of the filter functions available at the sending side of a channel through a filter identifier. The filter identifier can be `NULL_FILTER` if no filtering is desired. May return errors 1, 2, and 3.

4.4.5 Middleware

This section describes an example implementation of *remote procedure calls* (RPCs) and *sampling messages*[24, 27] using the dualoscom interface. The goal of this section is just to show the potential of the basic interface, rather than providing a complete middleware framework.

RPC communication

Dual-OS RPC communications allow an RTOS client to request the execution of a subroutine by a GPOS server (or vice versa) in the same manner as if the subroutine was local. Figure 4.7 outlines the pseudocode of a simple algorithm—error checking is not shown—for accomplishing RPC communications on top of the basic dualoscom interface.

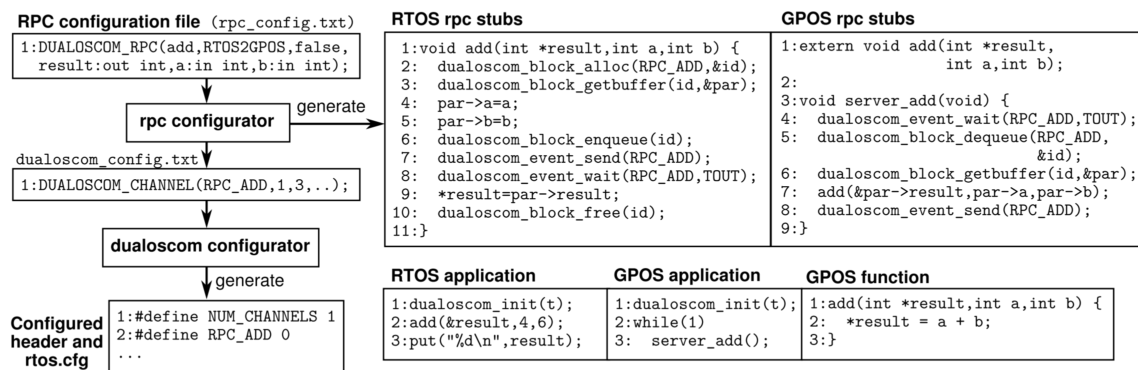


Figure 4.7: Pseudocode of RPC communication.

First, the system engineer must declare the RPC parameters on a configuration file (e.g., **rpc_config.txt**) which has the following syntax:

DUALOSCOM_RPC(): used to declare an RPC.

- **function**: the name of the function.
- **direction**: indicates the communication direction (RPCs are unidirectional).
- **mutex**: indicates if mutual exclusion is required at the client end.
- **params**: a list of parameters with the next format: **param : [in] [out] type**.

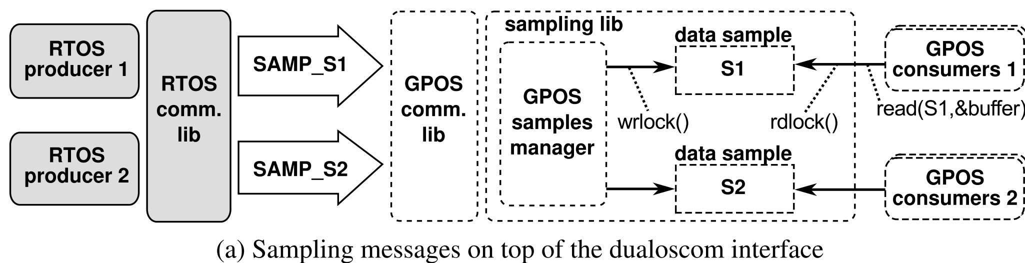
The RPC configuration file is parsed by the RPC configurator tool, which generates a dualoscom configuration file (e.g., **dualoscom_config.txt**) and the necessary stub functions on each OS that must also be linked into the final binaries.

RPCs are internally implemented through client request messages sent over dualoscom channels (e.g., **RPC_ADD**). Each request message contains the input parameters (e.g., **a** and **b**) for the subroutine, and memory space for the RPC server to store the output parameters (e.g., **result**). If the RPC is synchronous, the client is put into waiting state while the server processes the request message.

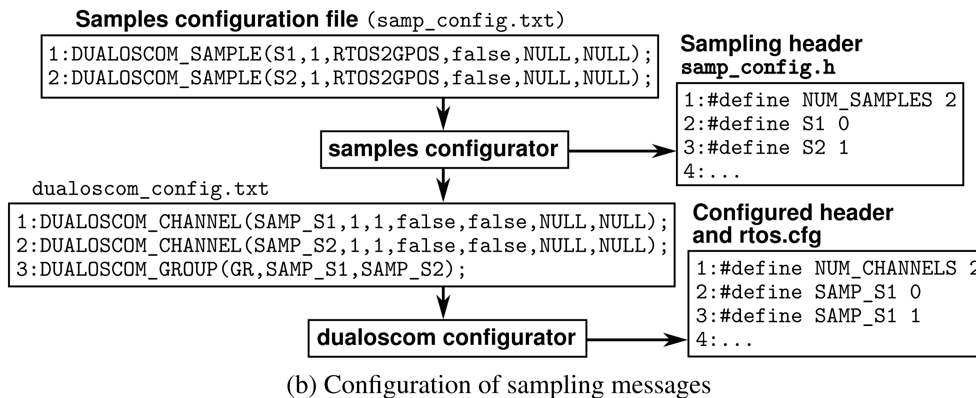
The main advantage of this interface is that, from the client application's point of view, the fact that the subroutine (e.g., **add**) executes remotely becomes completely transparent.

Sampling messages

Sampling messages—also known as *unqueued* messages—are part of several standards for real-time systems, such as ARINC653[24] or OSEK[27]. They are a useful method for the RTOS tasks to share data samples in a loosely-coupled fashion with the GPOS tasks. A data sample consists of a typically small region of memory containing a value that is updated periodically by a *producer* task. This value is read periodically by a loosely-coupled set of



(a) Sampling messages on top of the dualoscom interface



(b) Configuration of sampling messages

Figure 4.8: Support for sampling messages on top of the dualoscom interface.

consumer tasks. Sampling messages are useful for situations in which only the last value of some data (e.g., sensor data) is relevant to the application. Figure 4.8 illustrates a simple method to implement sampling messages on top of the dualoscom architecture. The example consists of two data samples sent in the RTOS ⇒ GPOS communication direction. Data samples (e.g., S1 and S2) are declared in a configuration file (e.g., **`samp_config.txt`**) using the following syntax:

`DUALOSCOM_SAMPLE()`: used to declare a data sample. The parameters are:

- **`SAMPLE_NAME`**: the name of the data sample. After configuration, the same name can be used to identify the data sample.
- **`mx_size`**: the maximum size of the sample.
- **`direction`**: indicates the communication direction (e.g., RTOS⇒GPOS).
- **`mutex`**: indicates if mutual exclusion is required to allow having multiple producers for the same data sample.
- **`filter`**: default filter.
- **`init`**: data sample initialization function.

The samples configuration file is parsed by the *samples configurator* tool (see figure 4.8(b)). This tool generates a sampling header (e.g., **`samp_config.h`**) that contains

the definition of several constants (e.g., the sample identifiers); and the dualoscom configuration file (e.g., **dualoscom_config.txt**), which contains a channel declaration per data sample (e.g., **SAMP_S1** and **SAMP_S2**) and a group. RTOS producer tasks periodically send new sample values through these channels. On the GPOS side, there is a sampling library that contains a samples manager agent. When a new sample value arrives, the samples manager updates the corresponding data sample in local memory (e.g., **S1** and **S2** in figure 4.8(a)). The access to these local data samples is protected through a readers-writer lock which allows several consumers to access the same data sample concurrently.

4.5 Implementation

This section gives details about the implementation of the dualoscom architecture on top of the SafeG dual-OS system.

4.5.1 Implementation platform

The hardware and software environment used to implement and evaluate the dualoscom architecture consisted of the following elements:

- PB1176JZF-S baseboard[35].
  - ARM1176JZF-S[31] core at 210MHz.
  - 32KB Cache
  - DRAM: 128MB (Non-Secure memory)
  - PSRAM: 8MB (Secure memory)
- RTOS: TOPPERS/ASP v1.6[30].
- GPOS: Linux v2.6.33 with buildroot[34].
- VL: TOPPERS/SafeG v0.2.

4.5.2 Code modifications

The implementation of the dualoscom architecture on ASP was completely accomplished at application level by leveraging ASP's application interface[30]. Therefore, neither the ASP kernel nor the SafeG monitor required any modifications. The most relevant parts of ASP's interface for the implementation of the dualoscom library were semaphore operations (e.g., **isig_sem** or **wai_sem**); and the static configuration interface of interrupt handlers, which were used for the transmission of asynchronous events through inter-OS interrupts.

In contrast, the Linux kernel was extended with a module that exports a character device file interface (e.g., `/dev/dualoscom`) to the user-space dualoscom library. The main two operations that the kernel module implements are the memory map (`mmap`) operation and the I/O control (`ioctl`) operation. The former is used by the function `dualoscom_init` to map a region of Non-Secure memory that is used as inter-OS shared memory. The later operation is used mainly for the transmission of events between both operating systems.

4.5.3 Initialization steps

The initialization of dualoscom comprises the following steps:

- Before Linux starts, an ASP task invokes the `dualoscom_init` function. This function initializes the shared memory region—which is hardwired into the code—and OS structures needed for the channel events (e.g., semaphores and interrupt handlers). Then the function resets a shared variable called `initialized` to false, and waits on a semaphore for the Linux side to initialize.
- When the RTOS SMC Task is scheduled, Linux starts its initialization. The values for the shared memory region are passed to the Linux kernel using the kernel parameters.
- When the dualoscom module gets loaded, it maps the shared memory region into the Linux kernel virtual address space; it creates a device file; and it associates an interrupt handler routine to the inter-OS interrupt number.
- Eventually a GPOS application will call the `dualoscom_init` function. This function maps the shared memory region into the application address space; initializes any application-level internal structures; and performs a system call to the kernel module. This call sets the initialized variable to true, and sends an event to ASP.
- Finally, ASP receives the event and wakes up the task that was waiting on a semaphore concluding the initialization process.

Note that if the GPOS never sends an event, then the task will still be woken up when the specified timeout value expires. Also, it is important to note that the library is prepared against malicious modifications of the shared memory region by the GPOS.

4.6 Evaluation

This section evaluates the satisfaction of the qualitative and quantitative requirements defined in table 4.1 by our implementation of the dualoscom architecture.

Table 4.2: Overhead of the four steps algorithm.

function	copied bytes	overhead
`block_alloc`	0	0μs
`block_free`	0	0μs
`block_getbuffer`	0	0μs
`block_enqueue`	12 bytes	2.15μs
`block_dequeue`	4 bytes	0.75μs
`event_send`	0	0μs
`event_wait`	0	0μs
`event_select`	0	0μs
`filter_set`	0	0μs

4.6.1 Requirement 4.1: memory isolation

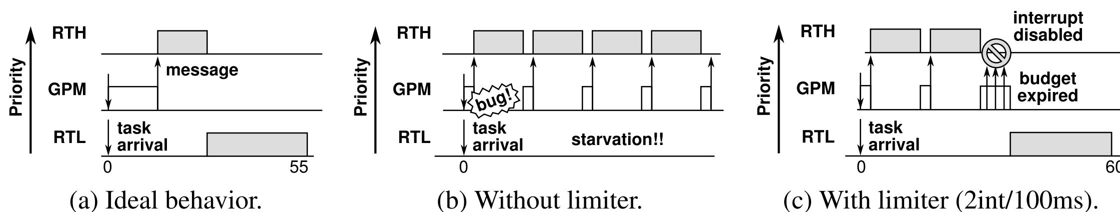
Memory isolation is satisfied by the dualoscom architecture through the use of two protection levels. In our implementation, the first level of protection is implemented through the permissions associated to a Linux device file (e.g., **/dev/dualoscom**) which are read/write for user tasks belonging to a configurable Linux group, and null for the rest of tasks. This device file is managed by the dualoscom Linux driver, and can be used for mapping the shared memory region into user space; or for the transmission of channel events. Protection with channel granularity can also be implemented at the cost of an increased memory usage due to the need of separated memory pages for each channel. The second level of protection is guaranteed by the TrustZone configuration as in the original architecture of SafeG.

4.6.2 Requirement 4.2: shared control data

Traditional approaches (see figure 4.2) store all control data in trusted memory, where it is not accessible by the GPOS. In contrast, dualoscom stores control data in shared memory—which is accessible by untrusted-privileged software—and uses a four steps algorithm (see § 4.4.1) involving a copy of the control data to RTOS memory. Updates to control data are done using variables validated on RTOS memory. Therefore the GPOS cannot attack the execution of the RTOS and requirement 4.2 is satisfied. Table 4.2 shows the amount of control data copied by each function and its validation overhead. We observed that the highest overhead occurs in the **block_enqueue** function, because it needs to copy and validate the active filter id and the FIFO's read/write cursors of the channel. The function **block_dequeue** only needs to copy and validate the FIFO's read cursor. The rest of functions do not require any copy since they are all based on simple atomic operations.

Table 4.3: Tasks for the evaluation of the message interrupt rate limiting functionality.

Task name	OS	Global Priority	C	T
RTH	RTOS	High	5ms	100ms
GPM	GPOS	Medium	10ms	100ms
RTL	RTOS	Low	40ms	100ms



(a) Ideal behavior. (b) Without limiter. (c) With limiter (2int/100ms).

Figure 4.9: Limiting mechanisms for dealing with message overload attacks.

4.6.3 Requirement 4.3: real-time

The dualoscom architecture supports message interrupt rate limiting mechanisms for protecting the real-time performance of RTOS activities against message overload attacks coming from the GPOS (see § 4.4.1). We evaluated the effectiveness of message interrupt limiting on a dual-OS system with two RTOS tasks and one GPOS task. The parameters of these tasks are shown in table 4.3, where C and T represent the execution time and period (deadlines are equal to periods) for RTOS tasks, or the execution time budget and replenishment period for the GPOS task. The dualoscom system is configured with a single communication channel for the GPM (GPOS Medium priority) task to send messages to the RTH (RTOS High priority) task every 100 ms. The RTL (RTOS Low priority) task does not use the dualoscom system at all. Instead, it is activated every 100 ms, and at each activation it consumes 40 ms of execution time with a global priority lower than the priority of the GPM and RTH tasks. Under an ideal scenario (see figure 4.9(a)) the worst-case response time (WCRT) of the RTL task (i.e., the length of the longest interval from the task's release till its completion) should always be shorter than $\sum C_i = 55ms$. However, if the GPM task misbehaves—due to a bug or a malicious user—it can potentially cause the RTL task enter starvation by generating a message overload attack against the RTH task (see figure 4.9(b)). Note that the time for the GPM task to send a single message is an order of magnitude lower than its execution time budget. Figure 4.9(c) illustrates how the dualoscom message interrupt rate limiting functionality can address this issue, by disabling the reception of message interrupts (e.g., using the RTOS interrupt controller) whenever the message interrupt rate limiting value is overrun.

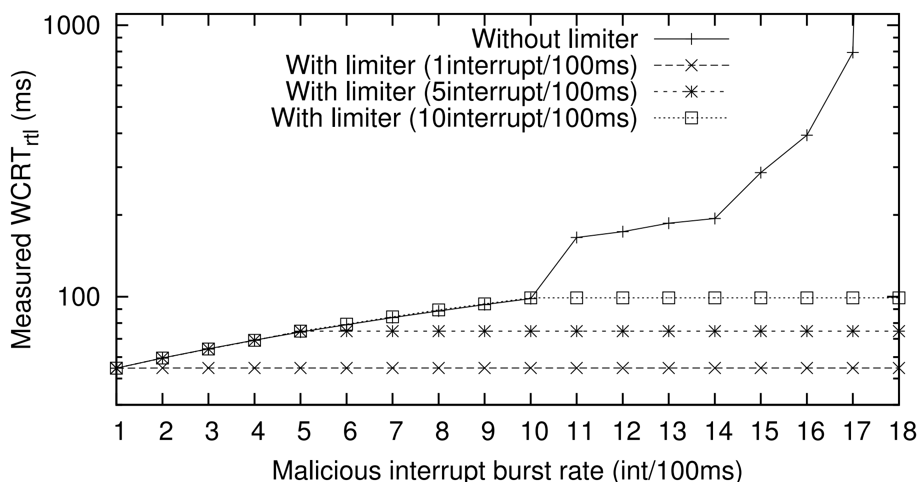


Figure 4.10: Evaluation of message interrupt limiting.

We inserted malicious code into the GPM task for generating message overload attacks against the RTH task. Then, we measured the WCRT of the RTL task with and without the use of a bursty message interrupt rate limiter (note that the strict limiter is a particular case of the bursty limiter). Figure 4.10 displays the measured WCRT of the RTL task for different malicious interrupt burst rates; and three different limiting values (i.e., 1, 5 and 10 interrupts/100ms). We observe that without a limiter, the WCRT of the RTL task increases proportionally to the malicious burst rate until it cannot meet its own deadlines for burst rates over 10. Then, it continues increasing with values close to the theoretical WCRT of the RTL task, which is defined by the smallest $x \in \mathbb{R}^+$ that satisfies the following formula[47] where N represents the malicious burst rate:

$$x = C_{rtl} + \left\lceil \frac{x}{T_{gpm}} \right\rceil C_{gpm} + \left\lceil \frac{x}{T_{rth}} \right\rceil C_{rth} \cdot N \quad (4.1)$$

For malicious burst rates above 17 interrupts per 100ms, the RTL task enters starvation. This occurs because the utilization of the RTH and GPM tasks reaches 100% for burst rates of 18 or more (e.g., $\frac{5 \cdot 18 + 10}{100} = 1$). In contrast, when limiters are used the WCRT of the RTL task is always upper-bounded by substituting N in Eq. 4.1 with the corresponding limiting value (i.e., 1, 5 and 10 interrupts/100ms). For that reason, we can say that the dualoscom architecture satisfies requirement 4.3.

4.6.4 Requirement 4.4: memory faults

This requirement is satisfied because the region of shared memory used for communications is allocated statically during configuration time. In our implementation, we reserve the

Table 4.4: Size increase.

OS	Level	SLOC	Binary increase
ASP	user	530 lines	5980 bytes
ASP	kernel	0 lines	0 bytes
Linux	user	483 lines	5456 bytes
Linux	kernel	279 lines	280 bytes
SafeG	monitor	0	0

region of shared memory by using a Linux kernel boot parameter (**mem**); and then create the necessary kernel page tables—in the dualoscom driver—using functionality for remapping I/O memory (**ioremap**[28]) from the Linux interface.

4.6.5 Requirement 4.5: unbounded blocking

The dualoscom architecture has two functions that can block the calling task: **event_wait** and **event_select** (see § 4.4.4). Both of them have a timeout parameter to protect RTOS tasks against the potential non-cooperation of the GPOS. The remainder interface uses non-blocking algorithms, and therefore we can say that requirement 4.5 is satisfied.

4.6.6 Requirement 4.6: code modifications

Table 4.4 presents the number of added source lines of code (SLOC) and the binary size increase for our reference implementation. Requirement 4.6 is satisfied because neither the RTOS kernel nor the SafeG monitor required any modifications. The Linux kernel was extended with a simple character driver. This driver can be easily maintained because its dependencies on Linux—fundamentally functions related to memory mapping and file operations—are small and unlikely to experience dramatic changes in the future.

4.6.7 Requirement 4.7: throughput

We implemented the traditional approach in figure 4.2(a) on top of SafeG, and compared its performance against the dualoscom approach. Figure 4.11 shows the results of the communications overhead of each approach (all measures are averaged). We observed that the dualoscom approach provides a constant overhead independent of the number of bytes being transmitted. This is a natural consequence of using zero-copy shared memory communications. In contrast, the overhead observed for the traditional approach is proportional to the amount of bytes being transmitted. The reason is that the main source of overhead

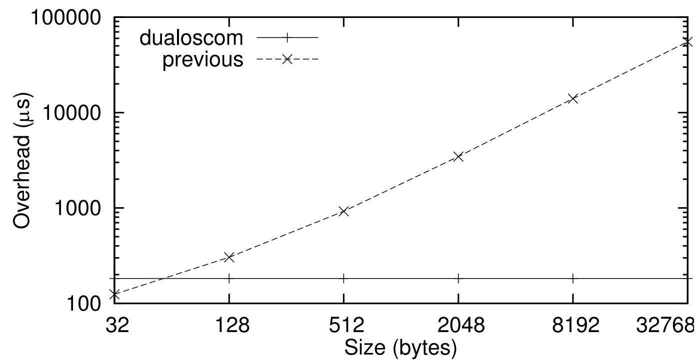


Figure 4.11: Overhead comparison.

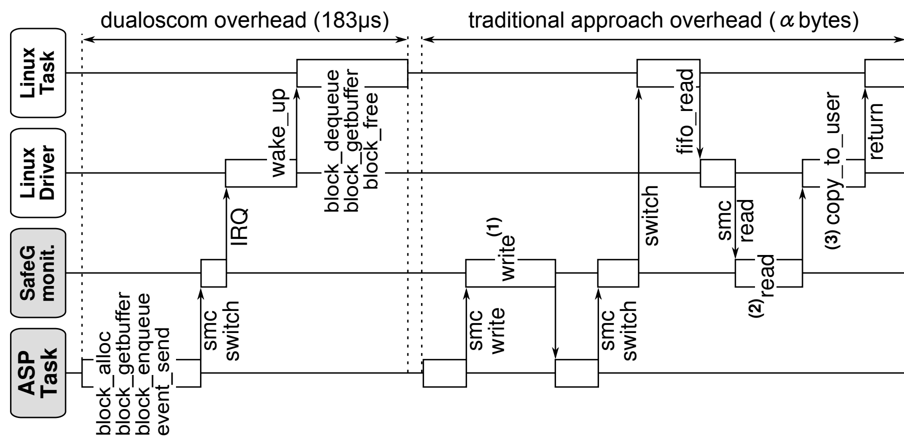


Figure 4.12: Execution flow comparison.

in the traditional approach is caused by the number of unnecessary data copies—(1), (2) and (3) in figure 4.12—and context switches. We also observed that differences in the communication overhead are smaller when the amount of data being transmitted is below 128 bytes. In particular, we observed that the traditional approach performed better for 32 bytes blocks. However, the main reason for that is that the dualoscom measurements include synchronization overhead (e.g., event notifications), while in the traditional approach the Linux task just polls (i.e., busy-wait) for the arrival of new messages at a **SCHED_FIFO** priority (this is the typical implementation of the traditional approach[11]). If the dualoscom approach is used in polling mode, the overhead for the transmission of a 32 bytes block decreases to 30μs. This is smaller than the 47μs required by the traditional approach to transmit a single byte. From these observations and other performance improvements—such as the filtering functionality or the ability to configure the synchronization needs of each individual channel—we can say that requirement 4.7 is satisfied.

4.6.8 Requirement 4.8: memory size

The memory size overhead per channel caused by the shared control data used for communications is only $40 + 4 \cdot NB$ bytes, where NB is the number of blocks in a channel. The memory overhead caused by the four steps algorithm (see § 4.4.1) is 16 bytes/channel. Also, the architecture provides a configuration interface for minimizing the size of the shared memory required in a particular application. For all these reasons, we can say that requirement 4.8 is satisfied.

4.6.9 Requirement 4.9: interface

§ 4.4.5 showed that the dualoscom interface is able to satisfy this requirement by supporting higher-level communication patterns—such as RPCs or sampling messages—that are often used in embedded systems.

4.6.10 Discussion

It seems that the dualoscom architecture has two fundamental advantages over the traditional approach: better real-time performance and lower overhead. The former is accomplished by using lock-free structures together with a novel 4-steps algorithm that avoids the need of using the VL—which runs with interrupts disabled—to transmit messages. The later is accomplished by using user-level shared memory which avoids unnecessary context switches or data copies, and makes it possible to send messages in constant time (see § 4.6.7) which is specially useful for time-sensitive applications. In contrast, the main advantage of the traditional approach over the dualoscom architecture is its simplicity.

4.7 Conclusions

We proposed a new dual-OS communications architecture designed by taking into account the reliability and efficiency requirements of a dual-OS system. We implemented it and the traditional approach on a physical platform using a reliable dual-OS system (SafeG) that leverages ARM hardware security extensions for guaranteeing the RTOS reliability. The implementation is distributed together with SafeG through the TOPPERS open source license[30]. Then, we evaluated both implementations through several experiments and observed that our architecture is more effective at minimizing the communications overhead and able to satisfy the reliability requirements of the RTOS.

Chapter 5

Reliable device sharing

This chapter investigates and compares several mechanisms for sharing devices reliably in a dual-OS system. In particular, we observe that device sharing mechanisms currently used for cloud virtualization are not necessarily appropriate for dual-OS systems. We propose two new mechanisms based on the dynamic re-partition of devices; and evaluate them on a physical platform to show the advantages and drawbacks of each approach.

5.1 Introduction

In dual-OS systems, devices are usually duplicated and assigned exclusively to each guest OS: devices that are critical for the reliability of the system are assigned to the RTOS; and the remainder devices are assigned to the GPOS. The dual-OS VL must guarantee that neither the GPOS nor GPOS devices—particularly devices with Direct Memory Access (DMA)—are allowed to access the memory and devices assigned to the RTOS.

Device duplication is useful for ensuring the reliability of the RTOS, and maximizing the system performance. However, it also adds a significant increase in the total hardware cost. Several device sharing mechanisms have been proposed in the context of cloud virtualization[16, 17] to reduce this cost. Most of them use a model in which device drivers are paravirtualized, splitting them between a front-end driver in the guest OSs, and the real back-end driver running on a trusted domain. The most typical application is sharing high-bandwidth network and storage devices that are concurrently accessed by numerous guest OSs. Despite its benefits, this approach is not suitable for dual-OS systems because of its rather high overhead; issues on the real-time performance of the trusted domain; and a significant increase in the complexity of the TCB.

This chapter investigates and compares several mechanisms for sharing devices reliably and efficiently in a dual-OS system. The main contributions are:

- A study on the suitability of existing device sharing mechanisms for dual-OS systems. We observe that—in contrast to cloud virtualization—highly concurrent device sharing is not usually required in dual-OS systems. Instead, a more common pattern is to use devices in turns, where the GPOS usage percentage greatly exceeds the RTOS's.

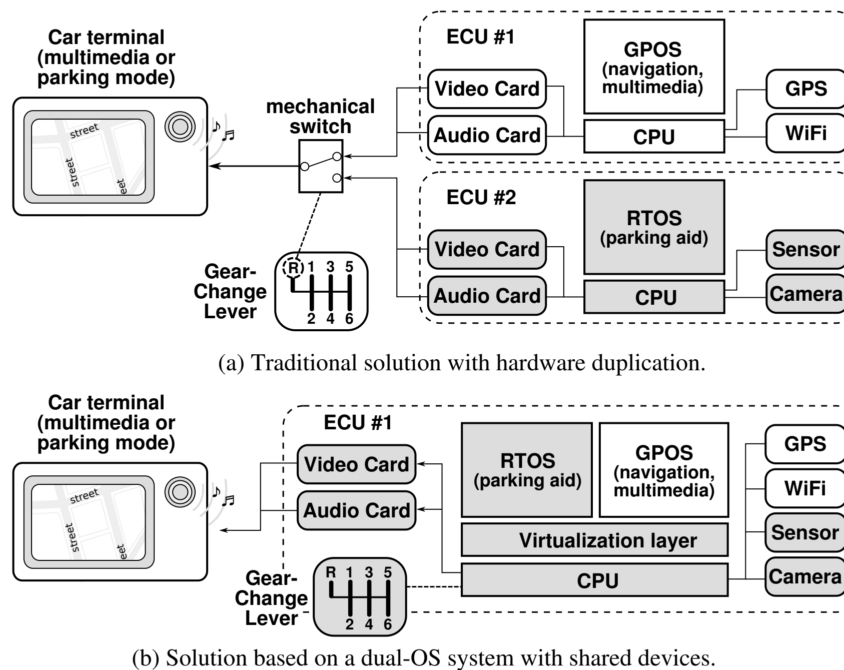
- We propose two new mechanisms based on the dynamic re-partition of devices to operating systems at runtime. The difference is a trade-off between execution overhead and the latency to access a shared device.

We implemented both mechanisms and the paravirtualization approach on a physical platform using the SafeG dual-OS system. From the results of the evaluation, we observed that each mechanism is best suited to a particular set of conditions and assumptions. In particular, our two new mechanisms seem more suitable for device sharing patterns commonly found in dual-OS systems than the paravirtualization approach.

The chapter is organized as follows. § 5.2 reviews previous approaches to device sharing and presents a motivational example for this work. § 5.3 is the core of the chapter and explains several device sharing mechanisms for dual-OS systems. § 5.4 details the implementation of our two new mechanisms and the paravirtualization approach on SafeG. § 5.5 evaluates the overhead, latency and code modifications of each implementation. § 5.6 compares this research with previous work. Finally, the chapter is concluded in § 5.7.

5.2 Motivation

Figure 5.1 illustrates a motivational example for reliable device sharing inside an in-vehicle car terminal[19] that operates in two modes: multimedia and parking mode. In *multimedia mode*, the terminal is used for GPS navigation, video playback or Internet access. This mode requires highly functional libraries such as video codecs or network stacks. For that reason, the most suitable way to implement it is by using a GPOS. In *parking mode*, the system fetches data from a camera and a distance sensor placed on the rear of the car. The camera data is displayed on the terminal to assist the driver during parking maneuver, and the distance to nearby obstacles is indicated through a repetitive sound. This mode requires high reliability and time determinism to avoid a potential car accident. For that reason, the most suitable way to implement it is by using an RTOS.



(a) Traditional solution with hardware duplication.

(b) Solution based on a dual-OS system with shared devices.

Figure 5.1: Motivational example for device sharing applied to an in-vehicle system.

The traditional approach to implement this system is illustrated by figure 5.1(a) and consists of two separated computing units (ECUs). One ECU contains a GPOS with rich libraries to handle the multimedia mode; and the other one contains a reliable RTOS to handle the parking mode. Parking mode is activated through a mechanical switch (i.e., a gear-change lever) whenever the car is driven backwards. Although this approach can satisfy the requirements of the system, duplication of the hardware increases the total cost.

In contrast, figure 5.1(b) illustrates a solution based on a dual-OS system with device sharing capabilities. Thanks to the use of a virtualization layer and device sharing, it is possible to consolidate both operating systems onto the same platform and avoid duplicating hardware. An important difference with device sharing in enterprise cloud virtualization is that devices (e.g., the video and sound card) are shared with low concurrency or rather in turns. For example, the car terminal is expected to operate in multimedia mode during most of the time; and only switch to parking mode occasionally. For that reason, existing device sharing mechanisms designed for highly concurrent systems—such as paravirtualization[16]—are not suited to this situation. Ideally, in a dual-OS system the GPOS should have direct access to devices for maximizing performance; and use its own feature-rich drivers instead of relying on a more complex TCB. Additionally, the worst-case amount of time that the RTOS has to wait for a shared device to be usable with reliability guarantees must be upper-bounded.

5.3 Reliable device sharing

5.3.1 Requirements and assumptions

Based on the motivational example above, we define the following set of requirements for the design of a reliable device sharing mechanism.

REQUIREMENT 5.1 *Completion*: device sharing mechanisms must guarantee that the TCB has full control over the successful completion of operations on shared devices.

REQUIREMENT 5.2 *Memory isolation*: TCB resources must be protected against any access—accidental or malicious—coming from UCB (including devices with DMA).

REQUIREMENT 5.3 *Real-time*: the timeliness of the RTOS must be guaranteed. In particular, malicious GPOS software must not be able to prevent or delay further use of a shared device (i.e., *device latency*) for an unbounded amount of time.

REQUIREMENT 5.4 *Software-only*: device sharing must be implemented in software. Customized hardware implementations are out of the scope of this chapter.

REQUIREMENT 5.5 *Performance*: the overhead caused by a device sharing mechanism (e.g., due to unnecessary data copies or context switches) must be minimized. Ideally, a device should be operated with native performance.

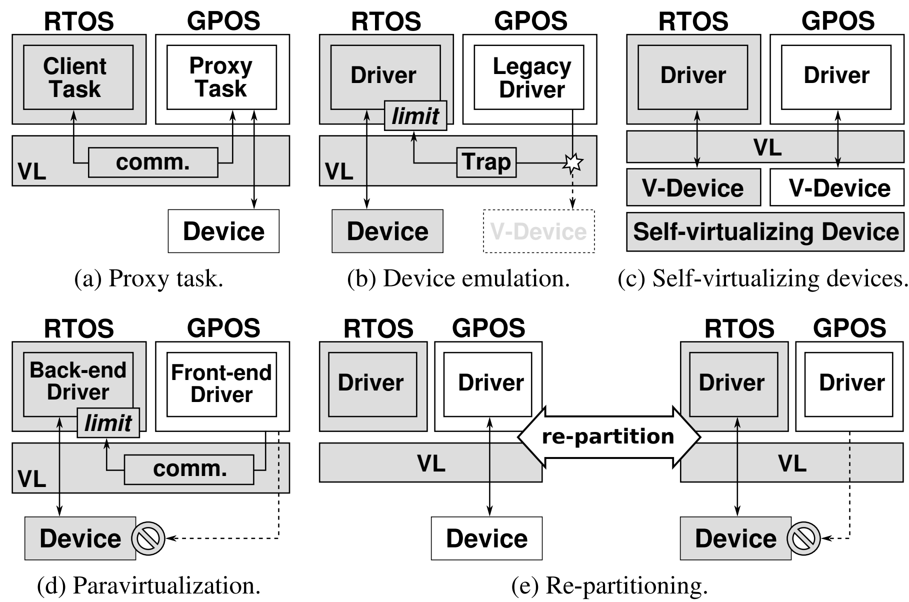
REQUIREMENT 5.6 *Code modifications*: modifications to the TCB software must be minimized. In particular, complex modifications to the VL must be avoided because they can increase the latency of RTOS interrupts. In contrast, the GPOS kernel can be extended with drivers. Nonetheless, GPOS applications and libraries should not require modifications for the sake of reusability.

We also make the following assumptions: software that belongs to the UCB does not have defects; the RTOS and GPOS drivers can be modified; the hardware reset time of a shared device is upper-bounded; the processor has a single core; and finally, we assume that the GPOS cannot damage a shared device.

5.3.2 Suitability of existing device sharing approaches

Figure 5.2(a)–(d) illustrate several existing approaches to device sharing, adapted to the context of a dual-OS system. Below we analyze each approach.

(a) *Proxy task*: in this approach, RTOS client tasks send requests to a proxy task in the GPOS—through a dual-OS communications system usually provided by the VL—with the intention of leveraging the richness of GPOS libraries and drivers. Requests can be



(a) Proxy task. (b) Device emulation. (c) Self-virtualizing devices.

(d) Paravirtualization. (e) Re-partitioning.

Figure 5.2: Device sharing approaches (VL=Virtualization Layer, comm.=Dual-OS communications, V-Device=Virtual Device).

sent with a high level of abstraction (e.g., play this sound), and therefore the overhead incurred is rather low. Despite all these benefits, the proxy task approach cannot be used for *reliable* device sharing because GPOS software is untrusted and it may misbehave or ignore RTOS requests which goes against requirement 5.1.

(b) *Device emulation*: this approach follows the classical Popek and Goldberg's trap-and-emulate model for machine virtualization[20]. The GPOS is tricked to think that there is a legacy device in the board. GPOS accesses to this virtual device are trapped by the VL and forwarded to the RTOS, where a driver handles the real device. This approach brings platform independence and flexibility to the GPOS. However, it has a significant execution overhead, and requires complex extensions to the TCB (see requirement 5.6) in order to implement the trap mechanism. Additionally, traps are typically delivered to the RTOS as software interrupts. To guarantee the real-time performance of the RTOS (see requirement 5.3), the TCB must limit the rate of these software interrupts, which may become a performance bottleneck if the GPOS needs to access device registers very frequently.

(c) *Self-virtualizing devices*: A self-virtualizing device with built-in support for real-time reservations could be shared seamlessly by the RTOS and the GPOS through separated interfaces, achieving near-native performance. Hardware virtualization support was recently introduced to some devices[33]. Unfortunately, the current availability of such

Table 5.1: Qualitative comparison of device sharing approaches.

Property	Existing approaches				Re-partitioning	
	Proxy	Emula-tion	Self-virt	Paravirt.	Pure	Hybrid
(1) Real-time	✗	✓	✓	✓	✓	✓
(2) Functionality	✓	✗	✓	✗	✓	✓
(3) Device Latency	✗	✓	✓	✓	✗	✓
(4) Overhead	✓	✗	✓	✗	✓	✗
(5) Concurrency	✓	✓	✓	✓	✗	✗
(6) Hardware Cost	✓	✓	✗	✓	✓	✓

devices is limited in practice to high bandwidth network and storage interfaces for enterprise cloud computing. The design of customized self-virtualizing hardware with support for real-time reservations is out of the scope of this chapter (see 5.4).

(d) *Paravirtualization*: in this approach, the GPOS is extended with a paravirtual driver—typically known as the front-end driver in XEN[16] split-driver terminology—that uses dual-OS communications for sending requests to the RTOS back-end driver. Paravirtualization helps raising the level of abstraction from bus operations to device-level operations in order to reduce the overhead, though its performance is still far from native. Similar to the emulation approach, the rate of device operation requests must be limited not to affect the real-time performance of the RTOS. The major drawback of this approach is the fact that the GPOS is limited to the functionality supported by the RTOS driver. RTOS drivers do not necessarily provide support for all of the functionality available in a certain device. For instance, a sound card may have audio capture features that are not needed by the RTOS. Implementing this extra functionality on the RTOS would complicate unnecessarily the TCB (see requirement 5.6).

The left part of table 5.1 summarizes qualitatively the properties of each approach. Property (1) refers to the ability to guarantee the timeliness of the RTOS. Property (2) indicates whether the GPOS uses its own fully functional drivers or not. Property (3) shows the adequacy of each approach to minimize the device latency. Property (4) refers to the overhead introduced by each approach. Property (5) expresses the suitability of each approach for a highly concurrent scenario. Finally, property (6) refers to the hardware cost of each approach.

We discard the proxy, device emulation and self-virtualizing approaches (i.e., approaches (a), (b) and (c)) because they cannot satisfy requirements 5.1, 5.6 and 5.4 respectively. Paravirtualization (approach (d)) can satisfy all of the requirements enumerated in § 5.3.1, at

the cost of reduced functionality and moderate overhead. However, it is not suitable for the type of device sharing patterns described in § 5.2, where the GPOS usage percentage of the shared device greatly exceeds the RTOS usage percentage. For that reason, in § 5.3.3 we explore a new approach based on dynamically re-partitioning devices between the RTOS and the GPOS at runtime.

5.3.3 Reliable device sharing through re-partitioning

The re-partitioning approach (see figure 5.2(e)) consists of dynamically modifying the assignment of devices to each OS at runtime. Re-partitioning is always initiated by the RTOS after a trigger condition (e.g., car going into backwards mode) and has several benefits:

- Devices can be accessed directly by both OSs which minimizes overhead.
- If a device is assigned to the GPOS, its interrupts (IRQ) are handled by the GPOS itself, which runs with the lowest RTOS priority. For that reason, the timeliness of RTOS tasks and interrupt handlers can be guaranteed.
- The VL does not require complex or any modifications at all.
- Any device can be used (e.g., not restricted to self-virtualizing devices).
- The GPOS can leverage its own feature-rich drivers, while the RTOS restricts itself to offer the minimum support in order to keep the TCB small.

We propose two mechanisms for implementing device sharing using the re-partitioning approach: a *pure* re-partitioning mechanism and a *hybrid* one. The main difference between them is a trade-off between the higher performance of pure re-partitioning; and the lower device latency of the hybrid mechanism.

Pure re-partitioning

Illustrated by figure 5.3, the architecture uses the concept of *hotplugging*—typically found in buses such as USB—and applies it to the dynamic re-partitioning of a device between the RTOS and the GPOS. Device sharing is managed by the so-called *Re-partition Manager* agents at each OS. The pseudocode of both agents is shown in figure 5.4. When a condition triggers the re-partitioning process, the RTOS re-partition manager is activated. The RTOS re-partition manager needs to handle two scenarios:

- If the device must be re-partitioned to the TCB, the RTOS re-partition manager will send an UNPLUG event to the GPOS counterpart. The RTOS re-partition manager is not dependent on the state of its GPOS counterpart. This is necessary for ensuring

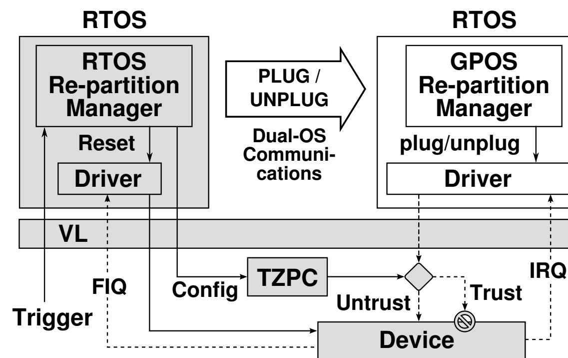


Figure 5.3: Architecture of the pure re-partitioning mechanism.

that even if the GPOS misbehaved, the RTOS would still be able to use the shared device with reliability guarantees. For that reason, once the UNPLUG event is sent, the RTOS re-partition manager needs to fully reset the device into a predefined state. This operation may involve disabling the device's interrupt, canceling current operations or waking the device from low-power mode. Immediately after resetting the device—and without the GPOS being able to execute—the RTOS re-partition manager configures the device as part of the TCB. Note that the opposite order would be insecure if the device was in the middle of a DMA operation. The method to configure a shared device as part of the TCB is dependent on the VL implementation. Once the re-partition process finishes, the RTOS can respond to the trigger condition and use the device reliably. When the GPOS is scheduled to execute by the VL (e.g., when the RTOS becomes idle) the GPOS re-partition manager must handle the UNPLUG event. The way to handle it may differ depending on the implementation but typically requires killing or suspending tasks that were using the device; and unloading or disabling the corresponding device driver.

- If the device must be re-partitioned to the UCB (e.g., when the RTOS does not need it), the RTOS re-partition manager must flush any sensitive data from the shared device; configure it as part of the UCB; and send a PLUG event to the GPOS. The GPOS re-partition manager will handle the PLUG event, which typically involves re-enabling or loading the corresponding device driver; and sending a notification to user space for registered processes to resume applications that were previously stopped.

The pure re-partitioning mechanism provides both OSs with direct access to devices for maximizing performance. However, fully resetting devices before re-partitioning can boost device latency to tens of milliseconds (see § 5.5), which may be considered excessive depending on the real-time application.

```
1  task RTOS_Repartition_Manager is
2  begin
3     loop
4        accept Repartition(Device, Trigger) do
5           case Trigger is:
6              when 'Set_Trust' =>
7                 Send_Event(UNPLUG)
8                 Reset(Device)
9                 Config(Device, TRUST)
10             when 'Set_Untrust' =>
11                Flush(Device)
12                Config(Device, UNTRUST)
13                Send_Event(PLUG)
14          end case
15       end Repartition
16    end loop
17 end task
```

```
1  task GPOS_Repartition_Manager is
2  begin
3     loop
4        Wait(Event, Device)
5        case Event is:
6           when 'UNPLUG' =>
7              Unplug(Device)
8           when 'PLUG' =>
9              Plug(Device)
10       end case
11    end loop
12 end task
```

Figure 5.4: Pseudocode of the pure re-partitioning mechanism.

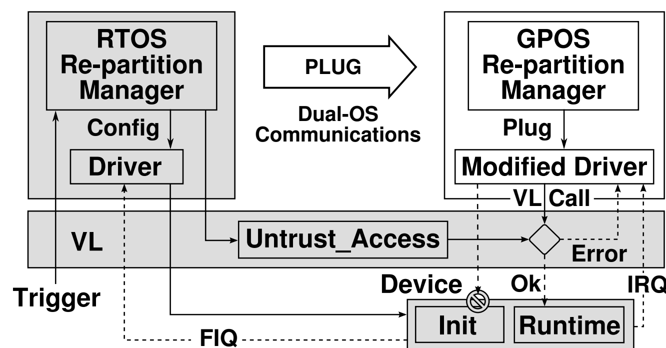


Figure 5.5: Architecture of the hybrid re-partitioning mechanism.

Hybrid re-partitioning

In order to reduce the device latency, we modified the pure re-partitioning mechanism with some concepts inspired by the paravirtualization approach, ergo the name of hybrid re-partitioning. The mechanism is derived from the observation that most part of the time spent on resetting a device is consumed on operations that are only performed at the initialization of the system but not at runtime. In the hybrid mechanism, the interface of a shared device is logically divided between bits that are required at initialization (*Init* interface); and those required during runtime (*Runtime* interface). The Init interface can only be accessed by the RTOS. For that reason, the RTOS can guarantee that certain conditions (e.g., that the device is powered on) are satisfied at all times, and thus reduce the time for resetting a device. In contrast, the Runtime interface can be re-partitioned to the RTOS or the GPOS. A software-only method to implement the hybrid approach (see figure 5.5) consists of configuring the

```
1  task RTOS_Repartition_Manager is
2  begin
3     Init(Device)
4     loop
5        accept Repartition(Device, Trigger) do
6           case Trigger is:
7              when 'Set_Trust' =>
8                 Reset_Runtime(Device)
9                 Config(Device, TRUST)
10             when 'Set_Untrust' =>
11                Flush(Device)
12                Config(Device, UNTRUST)
13                Send_Event(PLUG)
14          end case
15       end Repartition
16    end loop
17 end task
```

```
1  task GPOS_Repartition_Manager is
2  begin
3     loop
4        Wait_Event(PLUG, Device)
5        Plug(Device)
6     end loop
7  end task
```

```
1  procedure Write(Reg : in, Value : in) is
2  begin
3     Ret = VL_call(Reg,Value)
4     if Ret == Error then
5        Unplug(This)
6        Exit
7     end if
8  end procedure
```

Figure 5.6: Pseudocode of the hybrid re-partitioning mechanism.

device as part of the TCB, and extending the VL with a simple VL call for the GPOS to access the Runtime interface. Access permissions to the Runtime interface are controlled by the RTOS re-partition manager and the VL through a boolean variable (Untrust_Access) in trusted memory. Figure 5.6 shows the pseudocode of the hybrid mechanism which differs from the one in figure 5.4 in the following aspects:

- Devices do not require a complete reset when re-partitioned to the TCB because only the Runtime interface could have been altered by the UCB.
- In a software-only implementation, RTOS UNPLUG events can be replaced by a *lazy* algorithm. If the GPOS attempts calling the VL while the Runtime interface is assigned to the TCB, the VL will return an error code. The GPOS device driver is modified to handle this error code as an UNPLUG event. Note that the handling of PLUG and UNPLUG events must be serialized to avoid race conditions.

The right part of table 5.1 summarizes the properties of each re-partitioning mechanism. The hybrid mechanism has the major benefit of a shorter device latency, compared to the pure re-partitioning mechanism, because it ensures that time-consuming device initialization operations are not available to the GPOS. However, a software-only implementation of the hybrid mechanism requires small modifications to the VL and introduces overhead on each register access. Also, if the shared device has DMA capabilities, the VL may require further modifications in order to check that DMA memory addresses belong to the UCB.

5.4 Implementation

We implemented both re-partitioning mechanisms (pure and hybrid) and the paravirtualization approach—suitable for highly concurrent shared devices—on a physical platform for comparison. The hardware and software environment used to implement and evaluate the device sharing approaches consisted of the following elements:

- PB1176JZF-S baseboard[35].
  - ARM1176JZF-S[31] core at 210MHz.
  - 32KB Cache
  - DRAM: 128MB (Non-Secure memory)
  - PSRAM: 8MB (Secure memory)
- RTOS: TOPPERS/ASP v1.6[30].
- GPOS: Linux v2.6.33 with buildroot[34].
- VL: TOPPERS/SafeG v0.3.

The following device peripherals were used for the implementation:

- *Sound device*: an ARM PrimeCell Advanced Audio CODEC Interface connected to an LM4549 audio CODEC that is compatible with AC'97 Rev 2.1. The device in the board provides an audio channel with 512-depth transmit and receive FIFOs for audio playback and audio capture respectively.

- *Display device*: an ARM PrimeCell Color LCD controller (CLCDC) that provides a display interface with outputs to a DVI digital/analog connector for connecting to a CLCD monitor. The controller has dual 16-deep programmable 64-bit wide FIFOs for buffering incoming display data through a DMA master interface. The controller is configured through a slave interface, and has a color palette memory for low-resolution configurations.

Both devices can be configured to be part of the TrustZone Secure or Non-Secure worlds through the TrustZone Protection Controller (TZPC[36]). In particular, the master and slave interfaces of the CLCDC can be selectively configured as Secure and Non-Secure.

For the implementation of the paravirtualization approach, the GPOS was extended with a new ALSA[37] sound driver that acts as the front-end driver; and a simplified back-end sound driver—without capturing features—was added to the RTOS. GPOS operations on the sound card are forwarded to the RTOS back-end driver through the SafeG dual-OS communications system (see chapter 4). The GPOS video driver was also splitted in two parts.

The GPOS front-end driver implements the Linux framebuffer interface by sending requests to a simplified RTOS back-end driver which uses a low-resolution configuration. After that, pixel operations are performed directly on a region of Non-Secure memory accessed by DMA. The RTOS back-end driver validates that DMA addresses sent by the GPOS front-end driver belong to the UCB.

For the implementation of the two re-partitioning mechanisms, we used the baseline feature-rich (e.g., with audio capturing or high resolution video) GPOS sound and video drivers; and simplified drivers for the RTOS. The GPOS re-partition manager executes with a high SCHED_FIFO priority and handles hotplug events by killing/restarting tasks associated to a device; and removing/installing the corresponding device driver modules. The hybrid mechanism was implemented in software (i.e., through VL calls) because the TrustZone controller currently does not support bit granularity for the configuration of a device interface as Secure or Non-Secure. Therefore, the SafeG monitor was extended with a lightweight system call—implemented with a few assembly instructions—for the GPOS to access the Runtime interface. This system call involves a secure monitor call (SMC) instruction; a branch that depends on the value of the Untrust_Access variable (placed in Secure memory); validating the bits being accessed (including DMA addresses); and returning back to the GPOS.

5.5 Evaluation

This section presents the results of the evaluation of the device sharing implementations described above. The evaluation environment is the same as the one used for the implementation in § 5.4. All time measurement experiments were repeated for 10,000 times.

5.5.1 Overhead

In this section we evaluate the overhead that each mechanism causes on the handling of shared devices. The RTOS has direct device access (i.e., no overhead) in all mechanisms, and therefore we only evaluate the overhead on the GPOS.

First, we configured a system in which the RTOS is always idle and the GPOS is used either to play a 16bits/48Khz OGG Vorbis music file; or to show an MP4 video with 1024x768 pixels and 16 bpp resolution streamed from a network server. Both applications are executed with lower priority than the GPOS re-partition manager. Table 5.2 shows the measured execution time overhead per register access for each mechanism. In the pure re-partitioning mechanism, the GPOS can access shared devices directly, and therefore no overhead ap-

Table 5.2: Execution time overhead per register access.

	Paravirtual			Pure			Hybrid		
	min	avg	max	min	avg	max	min	avg	max
Sound	61μs	122μs	182μs	0	0	0	30μs	41μs	52μs
Video	47μs	117μs	187μs	0	0	0	30μs	42μs	53μs

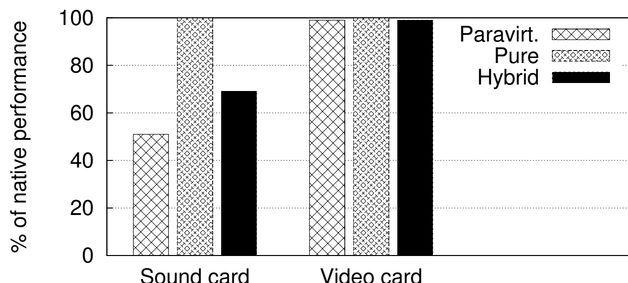


Figure 5.7: CPU performance for each mechanism.

pears. The overhead incurred by the paravirtualization mechanism is caused by the communications between the back- and front-end drivers. Note that we measured the overhead as per-register access because a single paravirtual operation may involve the reading or writing of several registers at once. Finally, the hybrid mechanism has lower overhead because register accesses do not cause a full context switch to the RTOS as in the paravirtualization approach.

Then, we repeated the same experiment but this time we also executed the *Dhrystone*[34] benchmark on the GPOS (with a lower priority) for quantifying the performance decrease caused by each mechanism. Figure 5.7 shows the performance of each mechanism as a percentage of the native performance. As expected, pure re-partitioning achieves 100% of native performance for both devices. The overhead of the paravirtualization and the hybrid mechanisms is considerably more pronounced for the sound card than for the video card. The reason is that the sound card is completely handled through registers; while the video card—once initialized through its slave interface registers—is managed simply by modifying a block of RAM memory that the master interface accesses through DMA. Currently, the overhead of the hybrid mechanism is higher than what we had expected because we found a cache coherence problem between the Secure and Non-Secure worlds. We have temporarily solved this problem by flushing the data cache for each register access, which introduces significant overhead. We also observed that the overhead of the paravirtualization approach in the handling of the sound card can be reduced by increasing the size of the buffer used to store music samples inside the ALSA front-end driver in the GPOS.

Table 5.3: Device latency of each mechanism.

	Paravir-tual	**Pure**	**Hybrid**
Sound	83μs	10.53ms	113μs
Video	3μs	20.22ms	10μs

Table 5.4: Number of source lines of code modified.

	Paravirtual	**Pure**	**Hybrid**
GPOS(user)	0	153	113
GPOS(kernel)	297	0	54
RTOS	38	43	32
VL	0	0	37

5.5.2 Device latency

Device latency is the worst-case amount of time that the RTOS may have to wait until a shared device can be used reliably. We modified the system described in § 5.5.1 (without the Dhrystone benchmark) so that every 10 seconds the GPOS audio or video playback application is interrupted by the RTOS, in order to emit a short beep sound (a raw PCM linked to the RTOS binary) or display a black and white alert message on the screen.

Table 5.3 shows the worst-case measurements for the device latency of each mechanism. The measurements for the paravirtualization and hybrid mechanisms are an order of magnitude smaller than the ones observed for pure re-partitioning. The reason for that is the fact that both the paravirtualization and the hybrid approach can limit GPOS access to critical bits of the device interface. For example, the GPOS is not allowed to set the AC'97 CODEC or the LCD in low power mode. In contrast, the pure re-partitioning approach allows the GPOS to access the device directly, and therefore shared devices must be fully reset every time the RTOS needs to use them. This must be taken into account during the real-time scheduling analysis of the system. The device latency of the hybrid mechanism is slightly longer than the latency observed for the paravirtualization approach. This can be explained by the fact that in the paravirtualization approach the usable functionality of a device is limited by the support included in the simplified RTOS driver. In contrast, in the hybrid approach the GPOS uses its own feature-rich drivers (e.g., with support for audio capturing and high video resolutions), and therefore there are a few more registers that need to be reconfigured.

5.5.3 Code modifications

Table 5.4 displays the number of source lines of code (C code, except the VL which is written in assembly) modified for each implementation. The paravirtualization mechanism required a new GPOS sound driver and modifications to the GPOS video driver in order to communicate with the RTOS drivers, which also required modifications. In the pure re-partitioning mechanism, most modifications occurred at user level where the re-partition managers execute. Finally, the hybrid approach required modifications both in user and kernel level. In particular, GPOS drivers were modified to perform calls to the VL, which was extended to handle this new paravirtual call.

5.5.4 Discussion

From the evaluation it seems that paravirtualization and re-partitioning are complementary approaches suitable for different target applications. In particular, we identify an interesting trade-off between the the lower overhead and higher functionality of the re-partitioning approaches; and the shorter device latency of the paravirtualization approach.

Dual-OS system engineers will probably prefer to apply the paravirtualization approach for device sharing scenarios that require high concurrency and flexibility (e.g., network or data storage controllers). However, they will have to pay the penalty of a considerable overhead and a more complex TCB.

In contrast, systems with low device sharing concurrency can take advantage of the re-partitioning approaches to increase the performance and smooth the maintainability of the system. The hybrid approach seems to be a good compromise between the pure and the paravirtualization approach because it provides lower overhead and better functionality than the paravirtualization approach at the cost of limiting the reset configuration of a device and a slightly higher device latency. However, we expect that this device latency can be reduced through simple hardware customizations.

Finally, we believe that these methods are independent of the SafeG architecture or the TrustZone extensions, and can be applied to any other dual-OS system architecture that satisfies the requirements listed in § 2.2.2.

5.6 Related work

While techniques for virtualizing processing time and memory resources have usually a rather low overhead, it is challenging to efficiently virtualize I/O devices. There exists

a substantial amount of literature describing methods to virtualize hardware devices. In particular, virtualization of high-bandwidth network interface devices in the context of enterprise virtualization for data centers has been the subject of extensive research.

- *Full device emulation* is used by fully virtualized systems[62]. In this approach, guest OS accesses to a virtual legacy device interface are trapped by a hypervisor, which converts them into operations on a real device. The main benefits of this approach are the fact that guest OSs do not require modifications; and the ability to migrate them between heterogeneous hardware. However, this approach incurs a significant performance degradation due to frequent context switches between the guest OS and the hypervisor.

- *Paravirtualization* is the de-facto approach to device sharing in most popular enterprise hypervisors[16, 17]. In this approach, guest OSs contain device drivers that are hypervisor-aware. A paravirtualized device driver operates by communicating with the real device driver which runs outside the guest. The real device driver that actually accesses the hardware can reside in the hypervisor or in a separate device driver domain with privileged access. The level of abstraction is raised from low-level bus operations to device-level operations. For that reason, paravirtualized devices achieve better performance than emulated ones. Nonetheless, paravirtualization introduces a rather high CPU overhead compared to a non-virtualized environment which also leads to throughput degradation in high bandwidth networks[63]. Several techniques to improve the performance of paravirtualized drivers have been presented. In [64] the authors report a 56% reduction of execution overhead on the receive path for conventional network interfaces through improvements on the driver domain model. [65] introduces improvements to the memory sharing mechanism used by paravirtualized drivers to communicate with the real device driver, reporting a reduction of up to 31% in the per-packet overhead. [66] proposes a software architecture which runs middleware modules at the hypervisor level. Their approach reduces I/O virtualization overhead by increasing the level of abstraction which allows to cut down the number of guest-hypervisor context switches. Despite the numerous performance improvements, paravirtual solutions are still far from native performance.

- *Direct device assignment*—also known as pass-through access—provides guest OSs with direct access to the real device, maximizing performance. With direct device assignment, an untrusted guest OS could potentially program a DMA device to overwrite the memory of another guest or the hypervisor itself. [67] presents a study on

available protection strategies. The most extended strategy involves the use of I/O memory management units (IOMMUs)[68]. Software-based approaches have also been presented[69, 70]. Recently, in [71] the authors report up to 97%-100% of bare-metal performance for I/O virtualization in a system that combines the usage of IOMMU and a software-only approach for handling interrupts within guest virtual machines. Despite its benefits, direct device assignment does not allow guest OSs to share the same device and makes live migration difficult[72, 73].

- *Self-virtualizing devices* have been introduced[33, 70, 74, 75] to avoid the high performance overhead of software-based device virtualization. This approach allows guest OSs to access devices directly, through separate interfaces that can be assigned independently to each guest OS. The main drawbacks of this approach are its increased hardware cost and limited availability.

Micro-kernels use a technique close to paravirtualization. Device drivers are implemented as user-space processes and applications communicate with them through interprocess communication[15]. Finally, direct device assignment is not easy to implement in embedded systems because they are not usually equipped with an IOMMU to provide the necessary isolation. Fortunately, recent ARM high-end embedded processors include TrustZone hardware security extensions[29] which provide similar functionality for separating up to two domains.

5.7 Conclusions

In this chapter, we investigated several device sharing mechanisms for dual-OS systems, where the most fundamental requirement is protecting the reliability of the RTOS. We observed that previous approaches are not well suited to device sharing patterns where the GPOS share greatly exceeds that of the RTOS. For that reason, we proposed two new approaches (pure and hybrid) that are based on dynamically re-partitioning devices between the RTOS and the GPOS at runtime. The reliability of the RTOS is ensured by the fact that before a device is re-partitioned to the RTOS, the device (or its run-time interface) is reset and configured as a TCB resource, which prevents further accesses by malicious GPOS applications. Additionally, when a device is re-partitioned back to the GPOS, its buffers are flushed to avoid leaking sensitive data. We evaluated both approaches and compared them with the paravirtualization approach, popular in cloud virtualization. We observed a tradeoff between the lower overhead and higher functionality of the re-partitioning approaches; and the shorter device latency of the paravirtualization approach.

Chapter 6

Conclusions and future work

This chapter contains a summary of the contributions of this thesis, together with some suggestions for future work.

6.1 Summary

In this thesis, we considered the reliable integration of dual-OS systems that consolidate an RTOS and a GPOS onto the same platform through a virtualization layer. The main three novel contributions proposed in this thesis were: an integrated scheduling framework; efficient dual-OS communications; and repartition-based device sharing.

The integrated scheduling framework supports the mixing of execution priority levels of both OSs with high granularity, and uses execution-time reservations for guaranteeing the timeliness of the RTOS. The evaluation results showed that the framework is suitable for enhancing the responsiveness of the GPOS time-sensitive activities without compromising the reliability and real-time performance of the RTOS.

We present a very efficient approach to dual-OS communications for enabling the collaboration of RTOS and GPOS applications in more sophisticated applications. Compared to traditional approaches that are usually implemented by extending the virtualization layer with new communication primitives, our approach minimizes the communication overhead caused by unnecessary copies and context switches; and satisfies the strict reliability requirements of the RTOS.

Finally, we investigated reliable device sharing mechanisms for dual-OS systems. We observed that existing approaches, in particular paravirtualization, are not well suited to device sharing patterns where the GPOS share greatly exceeds that of the RTOS. For that reason, we proposed two new approaches based on dynamically re-partitioning devices

between both OSs at runtime. The evaluation results showed an interesting trade-off between the lower overhead and higher functionality of the re-partitioning approaches; and the shorter device latency of the paravirtualization approach.

The three mechanisms were implemented on a physical platform and evaluated against the strict reliability requirements of a dual-OS system. From the evaluation results, we confirmed their suitability for improving the integration of a partitioning-based dual-OS system (SafeG) without affecting its low overhead, isolation, maintainability and real-time performance.

6.2 Suggestions for future work

It seems that multi-core processors pose new issues to the reliable integration of a dual-OS system. One of those well-known issues is the lock-holder preemption problem, which occurs whenever the GPOS gets suddenly preempted by the RTOS while in the middle of a critical section (e.g., holding a spinlock), and causes rather long blocking times to other cores trying to access it. It seems that an extension to the integrated scheduling framework presented in chapter 3 for solving this issue is worth exploring.

Another problem that arises on multi-core platforms is the need to coordinate both operating systems in the power management of the platform. In particular, reducing the frequency of a processor's clock could affect the real-time performance of the RTOS activities.

Additionally, in § 5.3.1 we assumed that the system runs on a single processor. On a multi-core implementation, both re-partitioning algorithms need to address a race condition that may occur if the GPOS accesses a device just after being reset by the RTOS, and before being configured as a TCB resource (e.g., lines 8 and 9 of the RTOS re-partition manager in figure 5.4). To solve this problem, a mechanism for the RTOS to block UCB accesses to the shared device, while still configured as an UCB resource, is needed. The implementation could be done in software by extending the VL with support for TCB critical sections; or in hardware by adding a new flag for blocking UCB accesses to the shared device.

There is also scope to improve the performance of the hybrid approach to device sharing presented in chapter 5. We suggest that TrustZone hardware could be extended to allow configuring device interfaces with finer granularity for the hybrid approach to be implemented with near-native performance. This will become practical soon with the release of new FPGAs containing ARM TrustZone-enabled cores.

Finally, the use of dual-OS systems as a method for implementing monitoring mechanisms against external attacks to the GPOS seems promising.

Bibliography

[1] LeVasseur, J. and Uhlig, V. and Chapman, M. and Chubb, P. and Leslie, B. and Heiser, G.: Pre-Virtualization: soft layering for virtual machines, Fakultät für Informatik, Universität Karlsruhe, Technical Report (2006).

[2] Hwang, J. and Suh, S. and Heo, S. and Park, C. and Ryu, J. and Kim, C.: Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones, Proc. 5th Annual IEEE Consumer Communications & Networking Conference, USA (2008).

[3] Uhlig, V. and LeVasseur, J. and Skoglund, E. and Dannowski, U.: Towards scalable multi-processor virtual machines, Proc. 3rd Conference on Virtual Machine Research And Technology Symposium, California, USA (2004).

[4] Heiser, G.: Hypervisors for consumer electronics, Proc. 6th IEEE Conference on Consumer Communications and Networking Conference, Las Vegas, USA, pp. 614–618 (2009).

[5] Cereia, M. and Bertolotti, I.: Asymmetric virtualisation for real-time systems, Proc. IEEE International Symposium on Industrial Electronics, Cambridge, UK, pp. 1680–1685 (2008).

[6] Yoo, S. and Liu, Y. and Hong, C. and Yoo, C. and Zhang, Y.: MobiVMM: a virtual machine monitor for mobile phones, Proc. 1st Workshop on Virtualization in Mobile Computing, Breckenridge, USA, pp. 1–5 (2008).

[7] Wilson, P. and Frey, A. and Mihm, T. and Kershaw, D. and Alves, T.: Implementing Embedded Security on Dual-Virtual-CPU Systems, Journal IEEE Design & Test, Vol. 24, No. 6, pp. 582–591 (2007).

[8] Cereia, M. and Bertolotti, I.: Virtual machines for distributed real-time systems, Computer Standards Interfaces, Elsevier, Vol. 31, No. 1, pp. 30–39 (2009).

[9] Marquet, P. and Piel, E. and Soula, J. and Dekeyser, J.: Implementation of ARTiS, an asymmetric real-time extension of SMP Linux, Proc. 6th Real-Time Linux Workshop, Singapore (2004).

[10] Takada, H., Iiyama, S., Kindaichi, T. and Hachiya, S.: Linux on ITRON: A Hybrid Operating System Architecture for Embedded Systems, Proc. 2002 Symposium on Applications and the Internet Workshops, Nara, Japan, IEEE, pp. 4–7 (2002).

[11] Masmano, M., Ripoll, I., Crespo, A. and Metge, J.: XtratuM: a Hypervisor for Safety Critical Embedded Systems, Proc. 11th Real-Time Linux Workshop, Dresden, Germany (2009).

[12] Wilson, P., Frey, A., Mihm, T., Kershaw, D. and Alves, T.: Implementing Embedded Security on Dual-Virtual-CPU Systems, Design & Test of Computers, IEEE, Vol.24, No.6, pp. 582–591 (2007).

[13] Heiser, G.: The Role of Virtualization in Embedded Systems, Proc. 1st Workshop on Isolation and Integration in Embedded Systems, pp. 11–16, Glasgow, UK (2008).

[14] Beltrame, G., Fossati, L., Zulianello, M., Braga, P. and Henriques, L.: xLuna: a Real-Time, Dependable Kernel for Embedded Systems, Proc. 19th IP based electronics system conference and exhibition (IP-SoC), Grenoble, France (2010).

[15] Armand, F. and Gien, M.: A practical look at micro-kernels and virtual machine monitors, Proc. 6th IEEE Conference on Consumer Communications and Networking Conference, pp. 395–401, Piscataway, USA (2009).

[16] Chisnall, D.: The Definitive Guide to the Xen Hypervisor, Prentice Hall Press, First edition (2007).

[17] Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A.: kvm: the Linux Virtual Machine Monitor, Proc. Ottawa Linux Symposium (OLS'07), pp. 225–230, Ottawa, Canada (2007).

[18] Nakajima, K., Honda, S., Teshima, S. and Takada, H.: Enhancing Reliability in Hybrid OS System with Security Hardware, IEICE Transactions on Information Systems, Vol.93, No.2, pp. 75–85 (2010).

[19] Hergenhan, A. and Heiser, G.: Operating Systems Technology for Converged ECUs, Proc. 6th Embedded Security in Cars conference (ESCAR), Hamburg, Germany (2008).

[20] Popek, G., and Goldberg, R.: Formal requirements for virtualizable third generation architectures. Communications of the ACM, vol.17, no.7, pp. 412–421 (1974).

[21] Kinebuchi, Y. and Koshimae, H. and Oikawa, S. and Nakajima, T.: Virtualization techniques for embedded systems, Proc. 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (Work-in-Progress Session), Sydney, Australia (2006).

[22] Ponsini, N.: Implementation Report of the Logical TrustZone/TPM integration, TECOM deliverable (2010).

[23] Freescale Semiconductor Inc.: Rich applications in real time, Introducing Vybrid Controller Solutions. BeyondBITS no.7, Whitepaper (2012).

[24] Airlines Electronic Engineering Committee: ARINC 653 Avionics Application Software Standard Interface (2003).

[25] OKL4: The OKL4 File System, Inter-VM communications. http://wiki.ok-labs.com/OKL4FS/.

[26] Burns, A. and Wellings, A.: Real-Time Systems and Programming Languages, 4th edition, Addison Wesley (2009).

[27] Lemieux, J.: Programming in the OSEK/VDX Environment, CMP Books, Elsevier (2001).

[28] Corbet, J., Rubini, A., and Kroah-Hartman, G.: Linux device drivers 3rd edition, O'Reilly Media Inc. (2005).

[29] ARM Ltd.: ARM Security Technology. Building a Secure System using TrustZone Technology, PRD29-GENC-009492C (2009).

[30] TOPPERS project: Official website. http://www.toppers.jp/.

[31] ARM Ltd.: ARM1176JZF-S Technical Reference Manual, DDI 0301G (2008).

[32] ARM Ltd.: AMBA3 TrustZone Interrupt Controller Technical Reference Manual, DTO 0013B (2008).

[33] PCI-SIG: I/O Virtualization, http://www.pcisig.com/specifications/iov/.

[34] Buildroot: Official website. http://buildroot.uclibc.org/.

[35] ARM Ltd.: RealView Platform Baseboard for ARM1176JZF-S User Guide (2011).

[36] ARM Ltd.: AMBA3 TrustZone Protection Controller TRM, DTO 0015A (2004).

[37] ALSA project: Official website. http://www.alsa-project.org/.

[38] Android project: Official website. http://www.android.com/.

[39] RTAI: Official website. https://www.rtai.org/.

[40] FRESCOR project: Official website. http://www.frescor.org.

[41] Hokuto Electronics: Puppy robot website. http://www.hokutodenshi.co.jp/7/PUPPY.htm.

[42] MPlayer: Official website. http://www.mplayerhq.hu/.

[43] Sloss, A. and Symes, D. and Wright, C. and Rayfield, J.: ARM System Developer's Guide (Designing and Optimizing System Software), ISBN: 978-1-55860-874-0, Elsevier Inc. (2004).

[44] Takada, H. and Sakamura, K.: μITRON for small-scale embedded systems, IEEE Micro, vol. 15, pp. 46–54, (1995).

[45] Ongaro, D. and Cox, A. and Rixner, S.: Scheduling I/O in virtual machine monitors, Proc. 4th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, Seattle, USA, pp. 1–10 (2008).

[46] Kinebuchi, Y. and Sugaya, M. and Oikawa, S. and Nakajima, T.: Task Grain Scheduling for Hypervisor-Based Embedded System, Proc. 10th IEEE International Conference on High Performance Computing and Communications, Dalian, China, pp. 190–197 (2008).

[47] Joseph, M., Pandya, P.: Finding response times in a real-time system. *The Computer Journal*, Vol. 29, No. 5, pp. 390–395 (1986).

[48] Kato, S. and Ishikawa, Y. and Rajkumar, R.: CPU Scheduling and Memory Management for Interactive Real-Time Applications, Real-Time Systems Journal, Kluwer Academic Publishers, Vol. 47, No.5, pp. 454–488 (2011).

[49] Bernat, G. and Burns, A.: New results on fixed priority aperiodic servers, Proc. 20th IEEE Real-Time Systems Symposium, Phoenix, USA (1999).

[50] Rostedt, S.: Finding origins of latencies using Ftrace, Proc. 11th Real-Time Linux Workshop, Dresden, Germany (2009).

[51] Kaiser, R.: Alternatives for scheduling virtual machines in real-time embedded systems, Proc. 1st workshop on Isolation and integration in embedded systems, Glasgow, Scotland, ACM, pp. 5–10 (2008).

[52] Regehr, J. and Duongsaa, U.: Preventing interrupt overload. *ACM SIGPLAN Notices*, Vol. 40, No. 7, pp. 50–58 (2005).

[53] Chen, S., Xu, J., Sezer, E., Gauriar, P., and Iyer, R.: Non-control-data attacks are realistic threats, Proc. 14th conference on USENIX Security Symposium, Baltimore, USA, USENIX, pp. 177–192 (2005).

[54] Wang, J.: Survey of State-of-the-art in Inter-VM Communication Mechanisms, Technical Report (2009).

[55] Zhang, X., McIntosh, S., Rohatgi, P. and Griffin, J. L.: XenSocket: a high-throughput interdomain transport for virtual machines, Proc. International Conference on Middleware, Newport Beach, USA, Springer, pp. 184–203 (2007).

[56] Li, D., Jin, H., Shao, Y., Liao, X., Han, X., and Chen, K.: A High-Performance Inter-Domain Data Transferring System for Virtual Machines. *Journal of Software*, Vol. 5, No. 2, pp. 206–213 (2010).

[57] Xia, L., Lange, J., Dinda, P., and Bae, C.: Investigating Virtual Passthrough I/O on Commodity Devices. *ACM Operating Systems Review*, Vol.43, No.3, pp. 83–94 (2009).

[58] Ram, K., Santos, J., Turner, Y. and Cox, A. and Rixner, S.: Achieving 10 Gb/s using safe and transparent network interface virtualization, Proc. International conference on Virtual execution environments, Washington, USA, ACM, pp. 61–70 (2009).

[59] Ram, K., Santos, J., Turner, Y.: Redesigning Xen's Memory Sharing Mechanism for Safe and Efficient I/O Virtualization, Proc. 2nd conference on I/O virtualization, Berkeley, USA, USENIX (2010).

[60] Gordon, A., Ben-Yehuda, M., Filimonov, D., and Dahan, M.: VAMOS, Virtualization Aware Middleware, Proc. 3rd Workshop on I/O Virtualization, Portland, USA (2011).

[61] Landau, A., Ben-Yehuda, M., Gordon, A.: SplitX, Split Guest/Hypervisor Execution on Multi-Core, Proc. USENIX Workshop on I/O Virtualization, Portland, USA, (2011).

[62] Sugerman, J., Venkitachalam, G. and Lim, B.: Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, Proc. USENIX 2001 Annual Technical Conference, pp. 1–14, Boston, USA (2001).

[63] Menon, A., Santos, J., Turner, Y., Janakiraman, G. and Zwaenepoel, W.: Diagnosing performance overheads in the XEN virtual machine environment, Proc. 1st ACM/USENIX international conference on Virtual execution environments (VEE '05), pp. 13–23, Chicago, USA (2005).

[64] Santos, J., Turner, Y., Janakiraman, G. and Pratt, I.: Bridging the gap between software and hardware techniques for I/O virtualization, Proc. USENIX 2008 Annual Technical Conference, pp. 29–42, Boston, USA (2008).

[65] Ram, K., Santos, J., Turner, Y.: Redesigning Xen's Memory Sharing Mechanism for Safe and Efficient I/O Virtualization, Proc. 2nd conference on I/O virtualization (WIOV'10), Pittsburgh, USA (2010).

[66] Gordon, A., Ben-Yehuda, M., Filimonov, D., and Dahan, M.: VAMOS, Virtualization Aware Middleware, Proc. 3rd conference on I/O virtualization (WIOV'11), Portland, USA (2011).

[67] Willmann, P., Rixner, S. and Cox, A.: Protection strategies for direct access to virtualized I/O devices, Proc. USENIX 2008 Annual Technical Conference, pp. 15–28, Boston, USA (2008).

[68] Ben-Yehuda, M., Xenidis, J., Ostrowski, M., Rister, K., Bruemmer, A. and Doorn, L.: The Price of Safety: Evaluating IOMMU Performance, Proc. Ottawa Linux Symposium (OLS'07), pp. 9–20, Ottawa, Canada (2007).

[69] Xia, L., Lange, J., Dinda, P., and Bae, C.: Investigating Virtual Passthrough I/O on Commodity Devices, Operating Systems Review, Vol.43, No.3, pp. 83–94 (2009).

[70] Willmann, P., Shafer, J., Carr, D., Menon, A., Rixner, S., Cox, A. and Zwaenepoel, W.: Concurrent Direct Network Access for Virtual Machine Monitors, Proc. 13th IEEE International Symposium on High-Performance Computer Architecture (HPCA-13), pp. 306–317, Phoenix, USA (2007).

[71] Gordon, A., Amit, N., HarEl, N., Ben-Yehuda, M., Landau, A., Schuster, A. and Tsafrir, D.: ELI: Bare-Metal Performance for I/O Virtualization, Proc. 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012), London, UK (2012).

[72] Zhai, E., Cummings, G. and Dong, Y.: Live Migration with Pass-through Device for Linux VM, Proc. Ottawa Linux Symposium (OLS'08), pp. 261–268, Ottawa, Canada (2008).

[73] Kadav, A. and Swift, M.: Live migration of direct-access devices, Operating Systems Review, Vol.43, No.3, pp. 95–104 (2009).

[74] Raj, H., and Schwan, K.: High performance and scalable I/O virtualization via self-virtualized devices, Proc. 16th international symposium on High performance distributed computing, pp. 179–188, California, USA (2007).

[75] Rauchfuss, H., Wild, T., and Herkersdorf, A.: A network interface card architecture for I/O virtualization in embedded systems, Proc. 2nd conference on I/O virtualization (WIOV'10), Pittsburgh, USA (2010).

List of publications by the author

Journal papers

1. Sangorrin, D., Honda, S., and Takada, H.: Integrated Scheduling for a Reliable Dual-OS Monitor, IPSJ Transactions on Advanced Computing Systems, Vol. 5, No.2, pp. 99–110 (2012).
2. Sangorrin, D., Honda, S., and Takada, H.: Reliable and Efficient Dual-OS Communications for Real-Time Embedded Virtualization, Journal of Computer Software, Japan Society For Software Science and Technology (to appear).

International conference papers

1. Sangorrin, D., Honda, S., and Takada, H.: Dual Operating System Architecture for Real-Time Embedded Systems, Proc. 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), pp. 6–15, Brussels, Belgium (2010).
2. Sangorrin, D., Honda, S., and Takada, H.: Reliable Device Sharing Mechanisms for Dual-OS Embedded Trusted Computing, Proc. 5th International Conference on Trust and Trustworthy Computing, Vienna, Austria (2012).

Domestic conference papers

1. Sangorrin, D., Honda, S., and Takada, H.: Integrated Scheduling in a Real-Time Embedded Hypervisor, 情報処理学会 組込みシステム研究会 第18回研究発表会, 公立はこだて未来大学 (2010).
2. Sangorrin, D., Honda, S., and Takada, H.: Inter-OS Communications for a Real-Time Dual-OS Monitor, 情報処理学会 第117回OS研究会, 那覇市 (2011).

Domestic conference papers (2nd author)

1. 太田貴也，Sangorrin, D.，一場利幸，本田晋也，高田広章， 組込み向け 高信頼デュアルOSモニタの マルチコアアーキテクチャへの適用, 情報処理 学会第117回OS研究会 (2011).
2. 太田貴也，Sangorrin, D.，本田晋也，高田広章， 組込み向け高信頼デュ アルOSモニタとそのマルチコア拡張, 第13回 組込みシステム技術に関する サマーワークショップ（SWEST13)ポスター，予稿集, 下呂市 (2011).

Awards and honors

1. Monbukagakusho (MEXT) scholarship from April 2009 to September 2012.