Scheduling for Mixed-criticality Hypervisor Systems in the Automotive Domain

Christos Evripidou

Doctor of Engineering

University of York
Computer Science

September 2016

Abstract

This thesis focuses on scheduling for hypervisor systems in the automotive domain. Current practices are primarily implementation-agnostic or are limited by lack of visibility during the execution of partitions. The tasks executed within the partitions are classified as event-triggered or time-triggered. A scheduling model is developed using a pair of a deferrable server and a periodic server per partition to provide low latency for event-triggered tasks and maximising utilisation. The developed approach enforces temporal isolation between partitions and ensures that time-triggered tasks do not suffer from starvation. The scheduling model was extended to support three criticality levels with two degraded modes. The first degraded mode provides the partitions with additional capacity by trading-off low latency of event-driven tasks with lower overheads and utilisation. Both models were evaluated by forming a case study using real ECU application code. A second case study was formed inspired from the Olympus Attitude and Orbital Control System (AOCS) to further evaluate the proposed mixed-criticality model. To conclude, the contributions of this thesis are addressed with respect to the research hypothesis and possible avenues for future work are identified.

List of Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

I would first like to thank my academic supervisor Prof. Alan Burns for his guidance and support throughout the duration of my course. Alȯso, a big thanks to my assessor Dr Leandro Soares Indrusiak for all his input during our TAP meetings.

A big thanks goes tȯ my industrial supervisor Dr Gary Morgan for his v̇aluable insights and for all our "brain-picking" sessions.

Many thanks to my family Pipis, Myria and Skevi for thėir love and support.

Dealing with the ups and downs would not be possible without the valuaḃle support of my friends Antonis Chatzimarkos, John Lenihan, Peter Skoutaris, Phil Dalton and Thanos Zolotȧs.

Of ċourse, I would like to thank all the people involved with the LSCITS group without whom I would not have received this opportunity. Special thanks to all the lecturers of the taught modules that were part of this course for all the knowledge and skills they helped me develop.

None of this would be possible without the suppȯrt of my sponsoring organisation, ETAS Ltd, and all the people I had the pleasure and privilege to work with.

Ending, I would like to thaṅk the University of York for all the experiences over the past nine years of being a student.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Patent Applications

The work presented in Chapter 3 was filed for patent grant. The published patent applications are as follows:

- C. Evripidou, A. Burns, and G. Morgan, "Method and apparatus for hosting a multitasking guest on a host system", EP Patent App. EP20,150,177,684 [42].
- C. Evripidou, A. Burns, and G. Morgan, "Method and apparatus for hosting a multitasking guest on a host system", CN Patent App. CN 201,610,826,878 [43].
- C. Evripidou, and G. Morgan, "Method and apparatus for hosting a multitasking guest on a host system", US Patent App. 15/215,113 [41].

Workshop Paper

The following paper was published at the Workshop on Mixed Criticality Systems (WMC 2016):

"C. Evripidou, and A. Burns - Scheduling for Mixed-criticality Hypervisor Systems in the Automotive Domain" [40].

Statement

I hereby give consent for my thesis, if accepted, to be made available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed .....(candidate)

Date .....

CHAPTER 1

Introduction

Complying with functional specifications alone is not sufficient to guarantee the correctness of a system. Many systems have both temporal and functional requirements that need to be fulfilled. Systems that require operations to take place within timing constraints are referred to as real-time systems [21].

Failure to meet these requirements can cause systems to exhibit consequences of varying severities [61]. Safety-critical systems are those whose failure can lead to unacceptable consequences [91], such as loss of life or significant financial damages. Completely proving the safety of such systems is not feasible due to their high complexity, however using safety standards, like ISO 26262 [54], can help reduce the risk and mitigate the consequences of failure. Standards have different Safety Integrity Levels (SILs), like Automotive Safety Integrity Levels (ASILs) A-D for ISO 26262, which are determined by performing hazard and risk assessment. Higher SILs can significantly increase the development costs, and make any estimates and/or assumptions about the system pessimistic. Typically, safety-critical systems with many different components would need to be verified at the highest safety assurance level, however this would result in high costs and underutilisation of resources [20, 38, 103]. The integration of components with different levels of criticality within the same system, is increasingly becoming a trend in real-time and embedded systems [20]. Such systems are referred to as Mixed-Criticality Systems (MCSs).

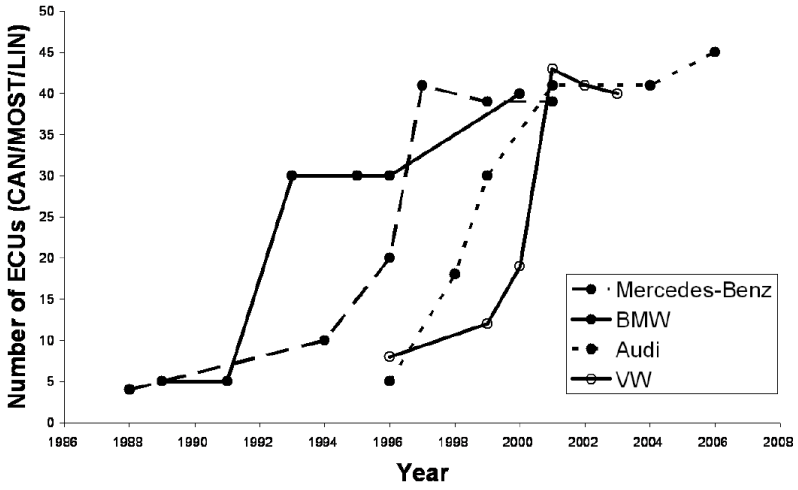


Figure 1.1: The growth in the number of ECUs [80].

1.1 Challenges in the Automotive Domain

The automotive industry has been using software in cars for over 30 years and this is increasing at a very fast pace [17, 18]. The software in a vehicle typically runs within ECUs, which are embedded hardware platforms responsible for controlling different subsystems within a vehicle. Examples of ECU are door control units, transmission control units and engine control units.

As stated by Broy [18] in 2006, about 40% of the production cost of a vehicle was spent on electronics and software. After the 30-year growth of the volume of software in vehicles, a modern premium car may contain in excess of 100 processors spread across 70 ECUs [1, 17, 18, 52]. Nolte [80] illustrates this problem in Figure 1.1 by plotting the number of ECUs present in cars by major manufacturers from 1988 to 2006. This is a clear indication of the increased complexity and the added hardware costs that are prominent in modern vehicles.

When they were first introduced, ECUs were functionally independent and were connected solely to sensors and actuators [17]. As ECUs were required to provide additional functionality, there was a need to establish communication channels between them. This change has resulted in multiple ECUs cooperating to provide a certain piece of functionality. In addition to numbers, ECUs can be heterogeneous and be responsible for different types of tasks; hard real-time (vehicle control) and soft real-time (infotainment). Given the large number of ECUs per vehicle, the dependencies between them and their lack of homogeneity, it can be inferred that they form a complex system that

is hard to reason about.

Apart from the structural complexity of the electronic parts of modern vehicles, another challenge in software engineering for the automotive domain is the difference in lifetime of car models and ECU hardware [17,83]. Specifically, a car model typically has a production lifetime of about 7 years, whereas a microprocessor 5 years. In addition to the 7 years of production lifetime, a car manufacturer, or Original Equipment Manufacturer (OEM), needs to provide service and spare parts for an additional 15 years. The result of this difference in lifetimes is that during a car's lifetime it is very likely that some ECU hardware components may stop being available in the market. Given that the ECU software is usually highly optimised for the underlying hardware, porting to a newer platform can be difficult and expensive.

The increased complexity that characterises modern vehicles led OEMs, suppliers and other relevant companies to form a worldwide development partnership. The result of this partnership is the AUTOSAR [9]. AUTOSAR aims to provide a common architecture as well as a methodology that will help with the understanding of the interaction of ECUs, allow software reuse and enable the combination of multiple functions on a single ECU [60].

The allocation of several functions on an ECU and code reuse are achieved by the layered architecture [9,16,60] shown in Figure 1.2. The bottom layer of this architecture is the ECU hardware, which interacts directly with the Basic Software layer. The Basic Software layer provides the necessary services that are needed by the AUTOSAR Software Components (SWCs) to be functional. Access to the underlying hardware is routed through the Microcontroller Abstraction Layer (MCAL). The AUTOSAR Runtime Environment (RTE) provides an abstraction for ECU communication. Specifically, it supports both inter-ECU communication and intra-ECU communication.

AUTOSAR supports the migration of SWC from one ECU to another. In order to do this, the Basic Software (BSW) modules need to be reconfigured to facilitate the new SWC. The BSW has over 80 modules, with approximately 200 configurable parameters each. With the current tooling support reconfiguring the BSW is an expensive procedure [79].

Although AUTOSAR provides a good solution in managing complexity and enabling reuse, it was designed for the provision of the facilities required for vehicle control. In modern vehicles there is an increasing number of applications that require a rich, typi-

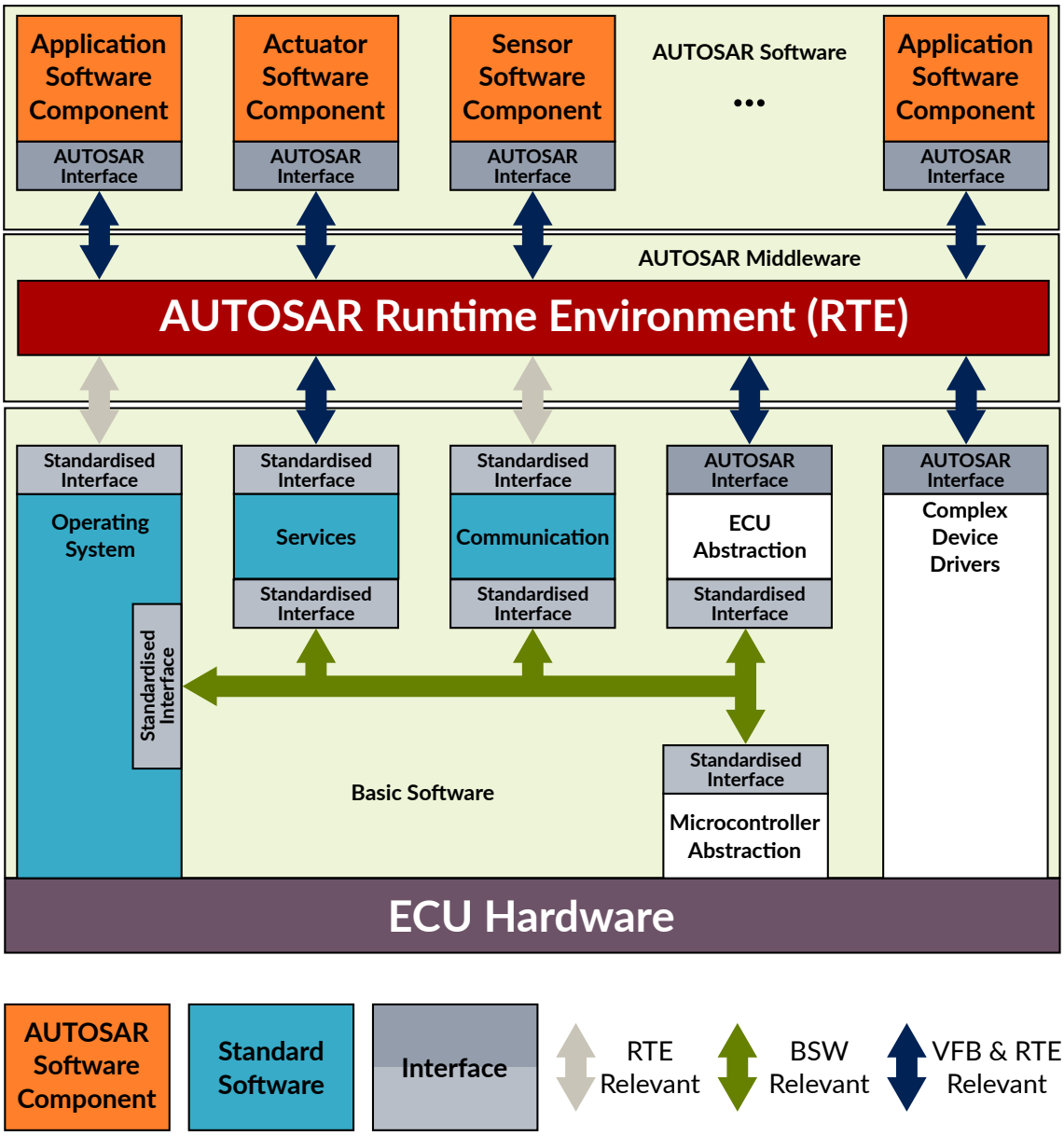


Figure 1.2: AUTOSAR architecture [9].

cally general purpose, Operating System (OS) [52]. Running such applications as SWCs is by design infeasible and require separate hardware. In addition to this, the real-time properties required by infotainment and the vehicle control functions are fundamentally different. These are contributing factors that make merging of these two functions on a single ECU under a single OS a difficult problem. Hergenhan and Heiser [52] argue that having infotainment and vehicle control on the same ECU is becoming a desirable feature, due to the increased interaction between the two, as well as the under-utilisation of the available processing power.

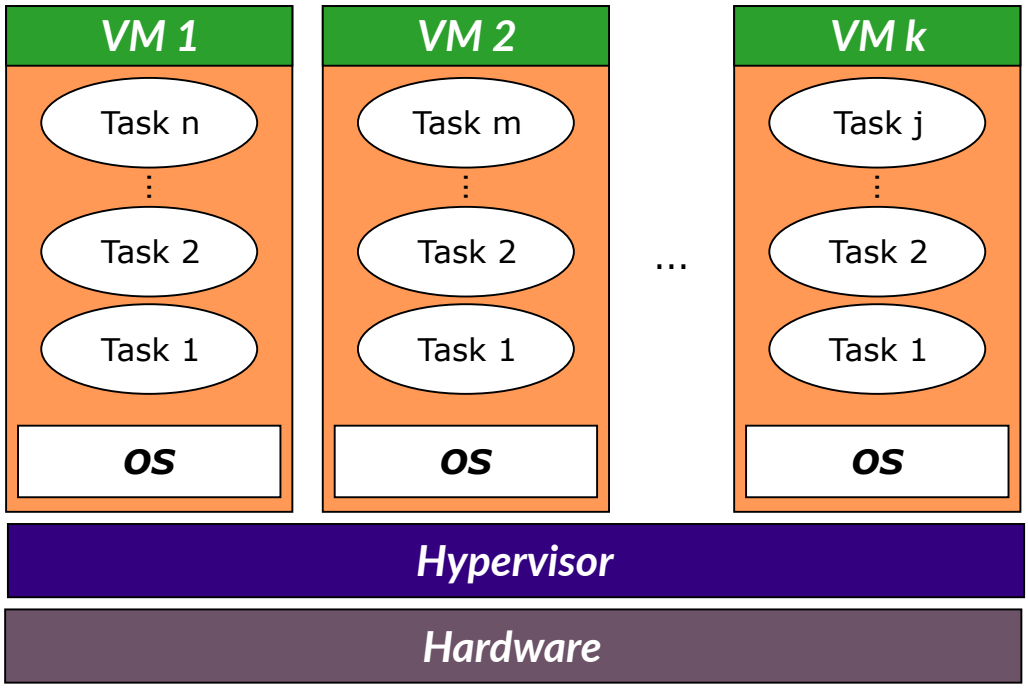


Figure 1.3: Virtual machine architecture.

1.2 Industrial Context and Motivation

This research project is a collaboration with ETAS Ltd[1] as the sponsoring organisation. ETAS Ltd is a company owned by ETAS GmbH, specialising in integrated tools and tool solutions for the development of automotive ECUs. ETAS GmbH is a sister company of Robert Bosch GmbH.

Virtualisation is a technique, initially developed in the early 60's [24], where logical resources are created in order to allow one or more applications to execute on the same hardware platform. The logical resources are created and managed by the Hypervisor (HV), also referred to as Virtual Machine Manager (VMM). Figure 1.3 is an example of a virtualisation architecture. From our experience with ETAS, there is increasing interest in the automotive industry for the use of virtualisation to alleviate some of the problems identified in Section 1.1.

The main use case for HV technology in the automotive domain is the reduction of ECU count by combining multiple ECUs on a single hardware platform. The key properties that must hold in a HV system is spatial and temporal isolation of the HVs. Spatial isolation is achieved by prohibiting the Virtual Machines (VMs) from accessing

[1] `http://www.etas.com`

memory areas outside of their memory space. Temporal isolation, which is the focus of this research project, is the property under which a VM's behaviour cannot cause another VM to violate its real-time properties.

The interest of exploring the use of virtualisation in the automotive domain is reinforced by secondary use cases that are relevant to emerging trends or problems in the industry. The following use cases were elicited from the collaboration with ETAS Ltd.

Mixed Criticality

AUTOSAR has been successful in combining multiple SWCs into multi-function ECUs, however this raised the need for software of different ASILs to coexist on the same address space. The integration of multiple ECUs on the same hardware platform would typically require all components to be certified at the highest ASIL [54], which would pose significant cost overheads. Virtualisation can be used to provide sufficient isolation between the constituent VMs, allowing for a lower level of pessimism in the real-time properties of the system and the potential reduction of costs associated with certification. Moreover, AUTOSAR's safety mechanisms when it comes to failure typically require reset of the ECU. Resetting a multi-function ECU will result in a temporary severe loss of functionality. The separation offered by virtualisation allows resetting individual VMs, therefore limiting the overall impact of the failures.

Multicore

Support for multicore platforms was only introduced in ECUs in recent years. Exploiting multicore platforms in older ECUs at the OS level would require a considerable cost. In a virtualisation system, scheduling happens in a hierarchical manner (see Section 2.4). In a hierarchical system, the VMs can be scheduled for execution on multiple cores, without requiring modifications to their local schedulers.

Portability

Porting automotive ECUs is an expensive task, due to the highly hardware optimised code. Virtualisation offers an additional layer of abstraction between the guest OSs and the hardware. In a virtualised system, the guest OSs will only need to be ported to work on top of the hypervisor, potentially being hardware independent. The hardware-specific functions are implemented by the hypervisor, therefore in future ports only the

hypervisor will require significant modifications.

Security

The introduction of security as an emerging property of supporting virtualisation can help protect Intellectual Property (IP) and prohibit tampering from unauthorised sources. SWC code is typically provided by different vendors that do not trust each other with their IP. In a hypervisor system, entire ECU images can be provided as object code for the use of the hypervisor instead of just individual SWCs. This provides a layer of protection against IP theft. Moreover, a hypervisor can take advantage of hardware encryption facilities offered by modern processors, therefore further protecting IP and at the same time prohibiting tampering with the ECU OS code.

1.3 Thesis Aim and Hypothesis

The focus of this research project is the use of virtualisation in order to solve some of the problems faced by the automotive industry due to an exponential increase in complexity. The use of software in vehicles is becoming more extensive at a very fast pace, which has resulted in a great increase in the number of ECUs per vehicle. Even though there was some effort to reduce this complexity with the development of a standardised architecture (AUTOSAR) [9], there is still much room for improvement. The use of virtualisation enables multiple control systems to run on common hardware under the control of a HV.

The use of HVs is typically associated with some degradation in the performance of the visualised application due the overheads associated with virtualisation. In non-real-time environments, the difference in temporal behaviour between an application executing natively or executing on a hypervisor-based system is often acceptable. In the automotive domain systems in vehicles are classified as safety-critical real-time systems. The temporal behaviour differences may result in deadline misses, with severe consequences, such as financial loss or even loss of life.

As it is explained in more detail in Section 2.6.2, at the time of authoring this thesis, the industrial sponsor of this research project had a working prototype hypervisor, RTA-HV. The current version of RTA-HV provides virtualisation support for a multi-core hardware platform, however having the limitation of only executing one VM per core.

This can result in the underutilisation of CPU resources. The value of the work in this thesis to the industrial sponsor is the proposal of a hypervisor-based system architecture which allows for the scheduling of multiple applications on a single core. Emphasis in the evaluation of the proposed architecture is the formulation of case studies, in order to provide confidence of the relevance of the findings of this thesis to the industrial sponsor.

The aim of the work done towards this thesis is to investigate the use of hypervisor technology from a real-time scheduling perspective. Specifically, we investigate on how to meet the requirement for low response times for event-driven tasks, while maintaining high utilisation. Particular focus is the incorporation of overheads in the scheduling model using realistic data. We then investigate extending the scheduling model in order to accommodate multiple levels of criticality. The scheduling models for both single and multiple levels of criticality are then evaluated using a realistic case study that was obtained via a detailed examination of a representative set of applications provided by ETAS Ltd.

> The hypothesis of this research project is that virtualisation can be used in the automotive industry to combine the functionality of more than one ECUs on a single hardware platform, while being able to make guarantees about the real-time properties of the system.

1.4 Thesis Outline

Chapter 2 - Field Survey and Review

Chapter 2 presents the relevant literature. The topics reviewed are timing predictability, real-time scheduling principles, mixed criticality scheduling and hierarchical scheduling. The literature survey concludes with a review of existing hypervisors.

Chapter 3 - System Architecture

Chapter 3 details a description of the proposed architecture and scheduling model of the system. We propose a memory configuration that supports spatial protection, providing the facilities required to allow the execution of multiple VMs on a single core. We then define the task model and the scheduling approach followed. The modifications required to the code of the applications that are run within the VMs are also identified.

Chapter 4 - Case Study: Engine Controller

Chapter 4 contains a realisation of the developed model using a case study. First the chosen hardware platform is overviewed, emphasising the features that are relevant to the development of a hypervisor. A taskset was then composed using timing characteristics that were obtained by performing timing analysis on real ECU code and a partial hypervisor implementation. The taskset was used to evaluate the tightness of the analysis and the scheduling model.

Chapter 5 - Extension to Mixed-Criticality

Chapter 5 extends the proposed model of Chapter 3 to support three levels of criticality using three modes: normal, first degraded and second degraded. A response time analysis was produced for all execution modes and mode changes.

Chapter 6 - Case Study: Mixed-criticality Engine Controller

Chapter 6 provides an evaluation of the proposed mixed-criticality model of Chapter 5 using sensitivity analysis. The sensitivity analysis provided the maximum Worst-case Execution Time (WCET) scaling for each criticality level, given three system configurations based on real ECU code.

Chapter 7 - Case Study: Olympus Attitude and Orbital Control System

Chapter 7 contains a case study inspired by the real-time characteristics of the Olympus Attitude and Orbital Control System (AOCS). The purpose of this case study is to study the performance of the system while exhibiting average-case behaviour. The findings of this case study provide additional insight to the expected behaviour of the system in each criticality mode.

Chapter 8 - Conclusion

Chapter 8 first provides a chapter-by-chapter summary of this thesis. The contributions made with respect to addressing the research hypothesis of Section 1.3 are presented. Limitations and possible areas for future work are then identified.

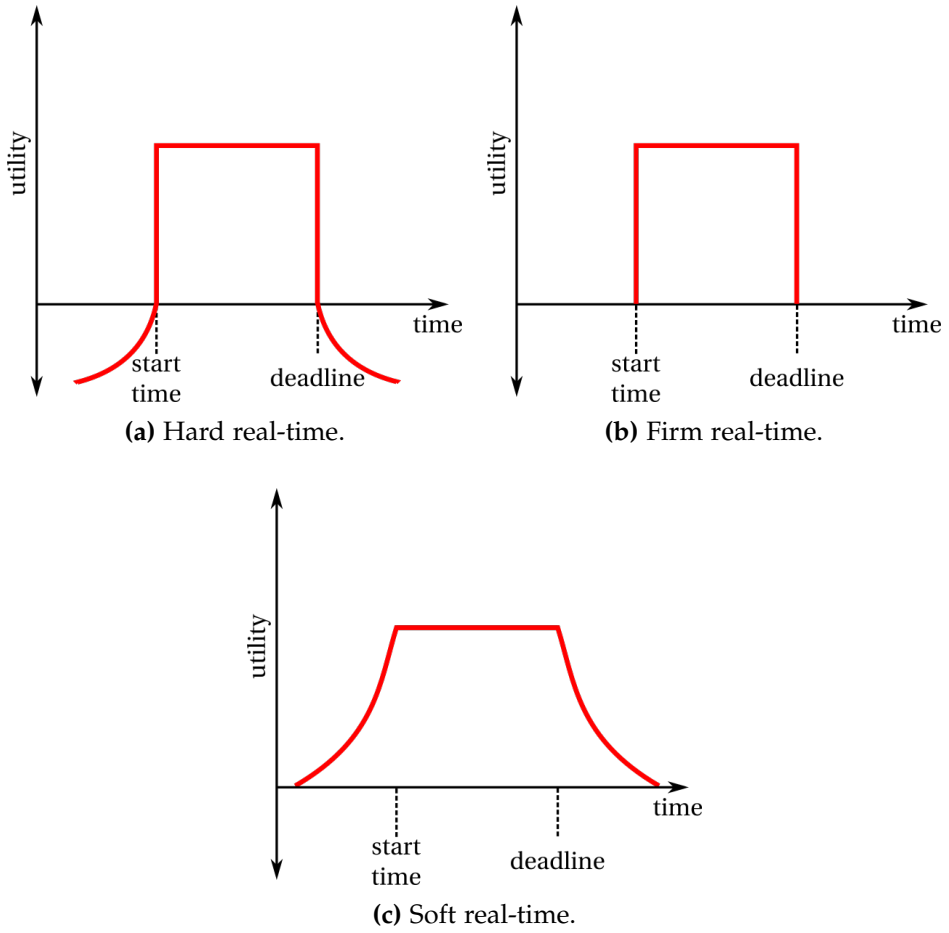
CHAPTER 2

Field Survey and Review

A system is considered as *real-time* if it is required to respond to external stimuli within defined time frames [21]. It is therefore dependent not only on the logical correctness of the software, but also on the timeliness of the output. Depending on the consequences of not complying with the timing requirements, a real-time system can be classified as *hard real-time*, *firm real-time* or *soft real-time*. Missing a deadline in a hard real-time system results in potentially disastrous consequences, whereas a soft real-time system can continue functioning with occasional deadline misses. Specifically, in the case of a soft real-time system having a late delivery, within a bounded limit, can result in reduced utility. In a firm real-time system missing occasional deadlines does not provide any utility. These are summarised in Figure 2.1.

In this chapter, we first introduce the concept of timing predictability, and in particular timing analysis. The review then moves to Fixed-priority Scheduling (FPS), as that is the scheduling approach followed by AUTOSAR-based OSs. Note that Dynamic-priority Scheduling (DPS) approaches, such as Earliest Deadline First (EDF) [71] and Constant Bandwidth Server (CBS) [2, 3] are not reviewed as they typically impose significant overheads and they are not used in the automotive domain for scheduling.

In order to make the review material relevant to the virtualisation problem set by the sponsoring organisation, applications of using hierarchical scheduling in relevant domains are reviewed. We then present the approach followed by the industrial sponsor of this project and identify the research gap to be investigated.



(a) Hard real-time.

(b) Firm real-time.

(c) Soft real-time.

Figure 2.1: Utility representation for hard, firm and soft real-time systems [19].

2.1 Timing Predictability

A real-time system consists of a number of tasks [106]. Each task shows some variation in execution time depending on the input and the environment's state. In order to explore timing predictability it is necessary to introduce the notions of WCET and Best-case Execution Time (BCET).

The calculation or estimation of a task's execution bounds is a difficult task [84, 106]. The first piece of information that is necessary for the evaluation of WCET is the program's worst execution path. The identification of this execution path is not trivial as it can be dependent on the state of the input or the environment [106]. Additionally, using just the source code with a known execution path is not sufficient to evaluate the WCET [84]. Another factor that contributes in the complexity of estimating the WCET is the process of compiling the source to machine code [84]. Due to compilers optimising the object code by rearrangements and transformations, the execution paths

are modified from those that were identified at the source level.

Timing anomalies are present in advanced processor architectures. This is because of some of the features that these architectures offer introduce a dependency between timing and execution history [84, 106]. Specifically, the time required to execute an instruction is dependent on performance-oriented features such as pipelines, branch prediction and caches [50, 100]. These features impact the timing predictability of a system, which is undesirable in hard real-time applications.

Wilhelm et al. [106] present an overview of timing analysis methods. These are classified as static or measurement-based methods. A static method obtains timing bounds by considering the possible control flow paths of a task's code in combination with a hardware model. With measurement-based methods the task code is executed on physical or simulated hardware, producing estimates of WCET and BCET. Static methods are safety-oriented, since they allow the analysis of hard real-time systems, covering corner cases that are potentially left unexplored by measurement-based methods.

2.1.1 Static Timing Analysis

Static timing analysis techniques obtain the execution time bounds by combining the task code with an abstract system model [44, 106]. Under the assumption that the system model is correct, static timing analysis provides a safe estimate for a task's execution bounds [25]. There are various phases that can be used to acquire WCET and BCET estimates. These include value analysis, control flow analysis, processor-behaviour analysis, estimate calculation and symbolic simulation.

Value Analysis

A *value analysis* is used to identify the memory addresses a task might require access to during its execution [106]. The purpose of determining the effective addresses of a task is to extract information regarding the amount of time required to perform each memory access. In their approach, Ferdinand et al. [44] perform value analysis by calculating an interval of possible values for each processor register. Having calculated the effective address range, it is possible to identify some infeasible execution paths, further informing the subsequent stages of the timing analysis.

Control-flow Analysis

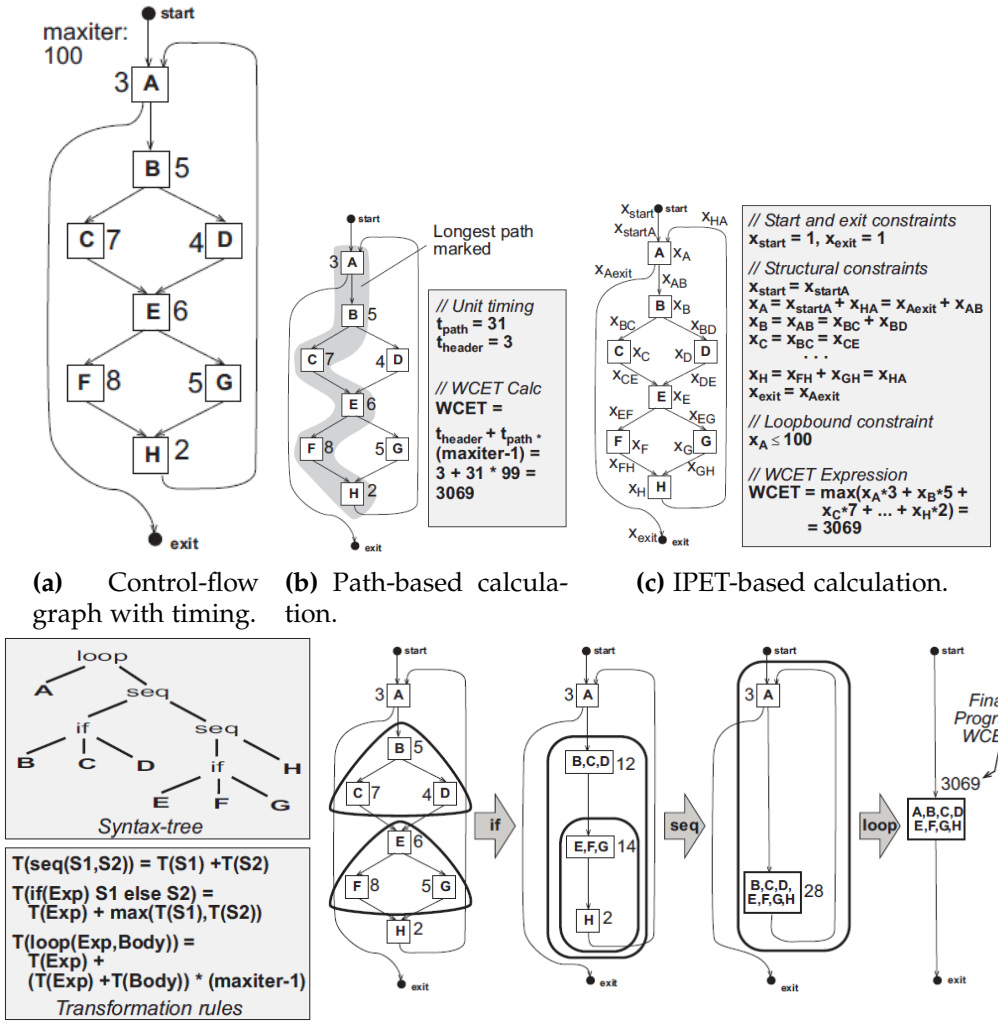
Another method used for the evaluation of WCET is the analysis of execution paths in order to gather information about them [106]. This is referred to as Control-flow Analysis (CFA). A key requirement of hard real-time systems is that every job of a task must terminate. This implies that there is a finite set of possible execution paths, some of which are not feasible. In general, this distinction is challenging, however it is possible to eliminate some paths from the analysis. This results in having to analyse a superset of the exact set of tasks, which still returns a safe WCET estimate.

An example of CFA is presented by Engblom et al. [36], using an automatic flow analysis based on abstract interpretation. Specifically, run-time behaviour properties are extracted by interpreting the program using abstract values instead of concrete ones, as well as using abstract semantics. Using this method, the program can be proven to be safe with respect to its run-time behaviour, provided that the abstraction of values and semantics are also safe. The analysis is performed on intermediate code representation, which suggests that the possible flows are identified in the executed code [48].

Processor-behaviour Analysis

The timing behaviour of the hardware that the code is running on must also be taken into consideration when performing timing analysis. As identified in Section 2.1, it is a difficult undertaking, especially with advanced processors, due to the timing behaviour being dependent on the execution history. This requires for *processor behaviour analysis* to improve the accuracy of the WCET estimates by including the hardware's properties with the task's source code [106]. The processor's occupancy state is analysed for all execution paths of the task in question.

Cousot P. and Cousot R. [28] introduce the notion of abstract interpretation. Abstract interpretation uses approximate semantics of the underlying structure of computations in order to obtain some information about the program behaviour without actually executing it. The principles of abstract interpretations are used in processor behaviour analysis. In general, with processor behaviour analysis an abstract processor model is used, which overestimates the timing requirements of each instruction. The overestimate is to ensure the calculation of safe WCETs. An invariance about these states is calculated using the results of the analysis. On relatively simple processors with some



(a) Control-flow graph with timing. **(b)** Path-based calculation. **(c)** IPET-based calculation.

(d) Tree-based calculation.

Figure 2.2: Different calculation methods [37].

performance features the analysis can be performed in a modular manner, where different processor features are analysed in isolation before combining the findings. In their paper, Heckmann et al. [50] argue that a modular approach is not suitable for more complex processor architectures because the high level of dependency of the different processor components require large safety margins during the analysis. A direct result of these conservative margins is a largely overestimated WCET, which is often not useful in practice.

Estimate Calculation

Bound calculation is used to produce an estimate of the WCET using the upper bound of the results of value analysis, control flow analysis and processor-behaviour analysis [37,

106]. As stated by Ermedahl [37], the WCET calculation methods are split in three main categories: tree-based, path based and Implicit Path Enumeration Technique (IPET). Examples of these are illustrated in Figure 2.2.

Tree-based calculation is performed by first generating a control-flow graph, whose nodes contain timing information. An example of a control-flow graph is shown in Figure 2.2a. In order to obtain timing timing information for the whole program, a bottom-up traversal of the control-flow graph is performed, as shown in Figure 2.2d. Although this approach is computationally cheap, it is not able to deal with dependencies between statements and with unstructured, possibly optimised code.

Path-based calculation methods produce a WCET estimate by calculating the maximum execution time of all the identified possible execution paths. This approach is generally straightforward, unless the code has loops. In the case of a loop the WCET of the loop's body is calculated. The body's WCET is then combined with the loop's flow information to obtain the overall WCET. An example of this approach is shown in Figure 2.2b.

IPET was first introduced by Li and Malik [66] as an efficient method for producing the execution time bounds of a program running on a given processor. With IPET, the WCET estimates are calculated by combining the execution time bounds of the basic blocks and the program flow into linear constraints. Every block in the control-flow graph of the task is allocated a time coefficient (t_{entity}) and a count variable (x_{entity}) [106]. The time coefficient represents the upper time bound of the entity. The count variable the maximum number of times the entity will be executed. WCET is then estimated by calculating the sum of products of the time coefficients and count variables of each entity ($\sum_{i \in entities} x_i t_i$).

Symbolic Simulation

With a *symbolic simulation* approach the WCET is calculated by running the task on an abstract processor model [106]. No input is used during the task's execution, which requires a combination of control-flow analysis, processor behaviour analysis and bound calculation. A drawback of this approach is that the time required for the time bound estimates to be calculated is proportional to the task's execution time.

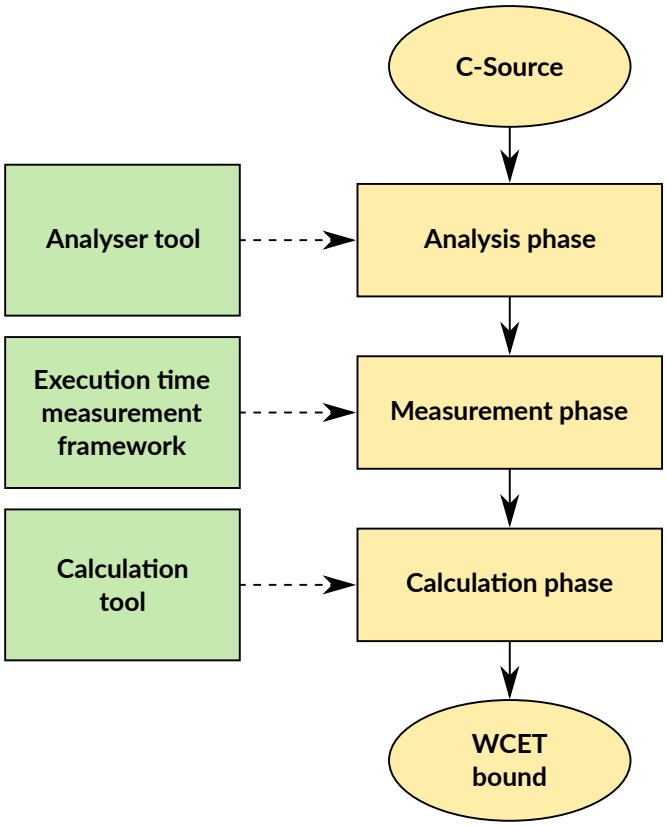


Figure 2.3: The three phases of measurement-based analysis [104].

2.1.2 Measurement-based Analysis

Measurement-based methods measure the execution times of tasks by executing them given a set of inputs either on hardware or a simulator [106]. This approach is used to produce estimates or distributions instead of bounds, unless the worst-case input is known. Wenzel et al. [104] present a Measurement-based Timing Analysis (MBTA) method as a WCET analysis technique, which uses a combination of static analysis techniques with measurements that were obtained with the execution of the program. The approach presented by Wenzel et al. [104] is performed in three steps, as shown in Figure 2.3.

The first phase of MBTA is the *analysis phase*. In this phase the source code is analysed, extracting path information and partitioning the program into segments. Test data is then generated in order for the program to use as input to obtain information about the time spent by the task in each segment. The second phase of MBTA is the *measurement phase*, where the execution times of each segment are obtained by running the program using the test data generated in the analysis phase. Lastly, in the *calculation*

phase, the execution times are combined with the path information in order to calculate the WCET bound estimates.

2.1.3 Static vs Measurement-based Analysis

In this section we provide a brief comparison between static and measurement-based methods.

As it was previously stated, static methods calculate execution time bounds using abstraction [106]. It is therefore necessary that an abstract processor model is used in the analysis. These models tend to be pessimistic and error prone, which can result to imprecise results [104]. Measurement-based methods, however, do not require processor models, instead they require special equipment, such as hardware or simulation to run the code on. This equipment can often be complex and expensive.

Following from the lack of processor models, measurement-based methods are prone to inaccuracies that result from the dependency of execution time on the processor's execution history. Even though measurement-based methods are referred to as unsafe, Wilhelm et al. [106] claim that in some cases the WCET and BCET estimates are more accurate than the ones produced by static analysis. This is more often the case in complex processors.

Measurement-based methods have difficulties dealing with timing anomalies. In order to overcome these, the execution times have to be measured for all initial states of the program, which is very time consuming and difficult. Anomalies make the definition of abstract models used in static analysis difficult.

The current practice in the automotive industry, as observed by our experience with ETAS, is the use of measurement-based techniques. Specifically, measurements are taken by defining two points in the code and counting the number of cycles elapsed between the two. This can be done using cycle-accurate simulators and/or physical development boards. Applications in the automotive industry are primarily linear with no dynamic data structures, therefore making their WCETs predictable.

2.2 Real-time Scheduling

In their book, Burns and Wellings [21] state that concurrent programs are aimed "to model parallelism in the real world", in order to be able to interact with entities in

it. The interaction with the real-world entities is typically performed through sensors and actuators. These are usually much slower than the processor, therefore possibly resulting in under-utilisation of the available resources. Introducing concurrency in programs can help minimise the time the processor remains idle. Additionally, it allows the exploitation of parallelism in problem solving, which can significantly lower the time required to reach a solution.

Concurrent programs require the specification of the order that tasks are executed at any point in time [21]. This is referred to as scheduling. Scheduling ensures the execution of tasks in a deterministic manner, while enforcing synchronisation primitives in order to ensure local ordering constraints.

2.2.1 Basic Concepts

In this section we introduce tasks and scheduling algorithms as fundamental real-time scheduling concepts.

Tasks

Real-time tasks can either be periodic, aperiodic or sporadic [19]:

A *periodic task* is characterised by its period (T), deadline (D) and execution time (C) [19]. To constrain these characteristics, a periodic task must have an execution time less than its period [71]. Additionally, it is usually required that the deadline of a process is not greater than its period, however this is not a necessary requirement.

An *aperiodic task* differs from a periodic one in the sense that it is triggered from an external source [19]. The associated timing information of an aperiodic task is its required execution time and its deadline. Typically, an aperiodic task have a greater level of criticality within a system, since they are usually responses to critical events.

A *sporadic task* is one that can be invoked at any time, however it has a predefined minimum inter-arrival time.

In simple models there is an underlying assumption that tasks are released in a perfectly periodic manner [21]. In reality there is a variation between the invocation of a task and its release. *Release jitter* (J) is defined as the maximum deviation of a tasks release time from its invocation.

Scheduling Algorithms

A *scheduling algorithm* provides facilities for managing the system resources, as well means of determining the worst-case behaviour of a system when it is used [19, 21, 71]. There are two categories of scheduling algorithms: static and dynamic. A *static* scheduling algorithm determines the task schedule prior to the program's execution, whereas a *dynamic* scheduling algorithm determines the task schedule at run-time.

Scheduling tests are used to determine whether a system is able to meet its timing requirements using a certain scheduling policy [21]. A schedulability test is said to be *sufficient* if it can guarantee that all deadlines will always be met. A *necessary* schedulability test indicates whether the system will miss a deadline miss during its execution. When a test is both sufficient and necessary is said to be *exact* (or *optimal*).

A goal of using scheduling algorithms is to maximise *processor utilisation*. Processor utilisation is the portion of the time the processor spends executing tasks. This is defined as the sum of execution time and period ratios for all tasks of the system, as shown in Equation 2.1.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2.1}$$

Task *interference* is another factor that has a very important role in determining whether a system is schedulable. The interference suffered by a runnable task is the time the processor spends executing higher priority tasks. Joseph and Pandya [55] derived Equation 2.2, which represents the cumulative interference a task τ_i may experience from all other tasks in the system of higher priority.

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.2}$$

where $hp(i)$ is a function that returns the set of tasks with a priority level that is higher than the priority level of task τ_i (see Table 3.1).

2.2.2 Rate Monotonic Scheduling

In a rate monotonic priority assignment each task is allocated a priority (P), which is inversely proportional to its period (T) ie. the shorter the period, the higher the priority [21, 65]. Specifically, for any two tasks τ_i and τ_j, $T_i < T_j \Rightarrow P_i > P_j$. In the case where two tasks share the same period the priority allocation is resolved in an arbitrary

manner [65]. Rate monotonic priority assignment is guaranteed to have optimal priority assignment [64] for uniprocessor systems in $O(n \log_2 n)$ time.

Using a rate-monotonic priority ordering, a schedulability test can be derived using the algorithm's worst-case utilisation bound [21, 64, 71]. Specifically, if the condition of Equation 2.3 holds, then all N tasks of the system will meet their deadlines. Specifically, for a system with $N \to \infty$, it is guaranteed that all of its timing requirements are met if the processor utilisation for all tasks is under 69.3% using rate-monotonic scheduling. This schedulability test was first introduced by Liu et al. [71]. In their paper, Devillers and Goossens [33] identify a mistake in the proof of this test, which does not affect the worst-case utilisation bound.

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{\frac{1}{N}} - 1) \tag{2.3}$$

In their paper, Bini et al. [15] developed a schedulability test of equal complexity but less pessimistic than the one by Liu et al. [71] in Equation 2.3. The proposed schedulability test is shown in Equation 2.4.

$$\prod_{i=1}^{n} \left(\frac{C_i}{T_i} + 1 \right) \leq 2 \tag{2.4}$$

Schedulability tests are generally not accurate and they are not easily applicable to more general task models [21]. Response-time Analysis (RTA) is a more computationally expensive but accurate approach for determining whether a system is schedulable. The main idea of RTA is to identify the response time of each task of the system (R) and check these values against the corresponding deadlines.

The response time of a task τ_i is defined in Equation 2.5 as the sum of its execution time (C_i) and the interference (I_i) from higher priority tasks. Equation 2.6 is derived by substituting I_i with the definition of Equation 2.2.

$$R_i = C_i + I_i \tag{2.5}$$

$$= C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{2.6}$$

In their paper, Audsley et al. [7] use Equation in 2.6 to form the monotonically non-decreasing recurrence relationship of Equation 2.7. In order to determine whether a set

of tasks is schedulable, the w_i^n values are evaluated. When two successive values are equal ($w_i^n = w_i^{n+1}$), then the task set is schedulable. If the value exceeds the corresponding task's deadline then the test fails.

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n + J_j}{T_j} \right\rceil C_j \tag{2.7}$$

The response time analysis can be extended to account for task blocking. Specifically, a task is *blocked* if it is suspended until a lower priority releases a system resource which is required for the higher priority task's execution [21, 94]. The situation where a lower priority task blocks a higher priority task is referred to as *priority inversion*. In their paper, Sha et al. [94] introduce the priority inheritance protocol, where if a lower priority task uses a shared resource, its priority is raised to the highest task priority using that resource. This protocol provides an upper bound on the blocking time of a task.

$$B_i = \sum_{k=1}^{K} usage(k, i) C(k) \tag{2.8}$$

The upper bound of a task's blocking time (B_i) is given by the sum of all the critical sections of the resources it uses ($C(k)$). Specifically, $usage(k, i)$ is a binary function which returns 0 if a resource k is used by task i.

In their paper, Davis et al. [32] identify the case of push-through blocking, where in some cases the response time analysis is optimistic. Assume a task model where the WCET is composed of the task main body followed by a non-preemptive region (ie. $C_i = C_i^{body} + C_i^{post}$). The task's deadline is met if its main body finishes its execution before its deadline. The non-preemptive section can therefore be executed after the deadline of the task, therefore adding additional blocking in its next release. The maximum amount of blocking B^{MAX} received by a task τ_i is therefore given by:

$$B_i^{MAX} = \max(B_i + C_i^{post}) \tag{2.9}$$

Incorporating the blocking time in the response time analysis equation:

$$R_i = C_i + B_i^{MAX} + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{2.10}$$

Similarly, the recurrence relationship becomes:

$$w_i^{n+1} = C_i + B_i^{MAX} + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n + J_j}{T_j} \right\rceil C_j \tag{2.11}$$

2.3 Mixed-Criticality Scheduling

In recent years, industry has shown interest in developing and certifying components independently, in order to reduce development costs and improve the system's performance [20, 38]. The different constituent components of such systems can have varying degrees of criticality, therefore requiring verification to different safety levels. Systems that consist of components with different levels of criticality are referred to as Mixed-Criticality System (MCS). The criticality levels are typically classified with respect to safety standards. Examples of these standards are RCTA DO-178B and ISO-26262, which are used in aviation and automotive respectively.

In his 2007 paper, Vestal [103] states that the WCET estimate is dependent on the level of certification of the application or system component. Therefore higher criticality WCET estimates are more pessimistic than lower criticality ones. Altmeyer et al [4] discuss the need for quantification of WCET estimate confidence. They discuss static and measurement-based timing analysis methods. They state that static analysis methods are believed to be superior to measurement based ones, since they can be proven to be safe, given the correctness of the models, and the confidence of the estimate can be obtained by comparing the models against real systems.

The predominant MCS task definition in the literature is: $(\vec{T}, D, \vec{C}, L)$, where $\vec{T}$ and $\vec{C}$ are vectors with period and execution times, respectively, for each criticality level. For any two criticality levels, L_1 and L_2, such that $L_1 > L_2$, $C(L_1) \geq C(L_2)$ and $T(L_1) \leq T(L_2)$.

$$R_i = C_i(L_i) + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(L_i) \tag{2.12}$$

Vestal [103] states that Audsley's algorithm [8] can be applied for MCS. Dorin et al [34] formalised Vestal's approach by proving that Audsley's algorithm is optimal for priority assignment in MCS. Vestal's model was extended by incorporating release jitter and adapted traditional sensitivity analysis to apply to MCS.

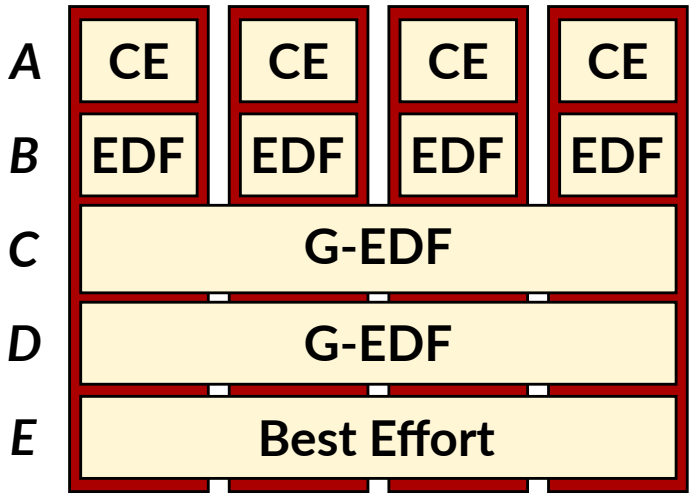


Figure 2.4: Container scheduling for a four-core system [77].

Building on Vestal's model, Baruah and Vestal [11] extended the approach to support use with sporadic tasks. They show that feasibility analysis can be performed using algorithms for traditional sporadic task systems. They introduce a hybrid algorithm that combines EDF and Vestal's algorithm. In their algorithm, each task is assigned a priority that is not necessarily unique. Tasks with the same priority are scheduled using EDF. Baruah and Vestal's approach outperforms Vestal's original model but not FPS.

Baruah and Burns [13] address the issue of low schedulability of Vestal's approach by monitoring the execution time of tasks. In his approach, Vestal evaluates the response time of tasks at the highest level of criticality. Monitoring the execution times of tasks prevents overruns, therefore improving the resource utilisation.

The execution of MCS starts in the lowest criticality mode. If a task misses its low criticality deadline, $C(LO)$ then a mode change is triggered and the execution of the system changes to HI. In their review, Burns and Davis [20] state that although there is research on mode change protocol, there is the problem that a system can be schedulable in every mode, but not during mode changes [101].

In their paper, Mollison et al. [77] propose an architecture for scheduling mixed-criticality tasks on a multi-core platform, which is referred to as MC^2 in subsequent work by Herman et al. [53]. Their proposed architecture follows a criticality classification, which is similar to the one proposed by RCTA DO-178B. Specifically, there are five levels of criticality, labelled from A to E, A being the highest level and E to the lowest. The tasks are scheduled by intra-container schedulers for each criticality level. Figure 2.4 summarises the proposed architecture on a four-core platform.

Tasks of criticality A are scheduled using a *cyclic executive* scheduling approach. Cyclic executive is a table-based approach, where tasks are scheduled according to a precomputed table. Level B tasks are scheduled using a Partitioned Earliest Deadline First (P-EDF) scheduler, because of it has relatively low overheads and has been theoretically shown to be optimal on on single-core. Level C and D are scheduled using a Global Earliest Deadline First (G-EDF) scheduler to support tasks where a small amount of tardiness can be tolerated. Lastly, level E tasks are scheduled whenever the processor is idle.

The architecture that was proposed by Mollison et al. [77] was implemented by Herman et al. [53], as stated in their 2012 paper. MC^2 was implemented using the LITMUS^{RT}[1] Linux kernel extension. LITMUS^{RT} extends the Linux kernel to support modular scheduler plugins. The implementation of MC^2 shows that the overheads introduced by the architecture are relatively small. Furthermore, Herman et al. [53] argue that MC^2 is robust with respect to mistakes in the WCET estimates.

2.4 Hierarchical Scheduling

In their journal article, Lipari and Bini [70] identify a use case for hierarchical scheduling, which is very relevant to the requirements posed by the sponsoring organisation of this project, ETAS Ltd. Specifically, they state that in many applications, it is desirable to move well functioning applications that were implemented on older processors without having to spend a significant amount of time re-designing and re-implementing. Hierarchical scheduling is proposed as a possible way of enabling a number of applications to work on common hardware, while ensuring that the timing requirements are still being met.

Hierarchical scheduling is a partitioned scheduling framework, where tasks and processes are grouped together into applications that are to be executed on underlying hardware [14, 26, 46, 98]. In a hierarchical scheduling system, each application implements its own local scheduling algorithm [30]. The applications are then allocated CPU bandwidth according to a global scheduler. Specifically, the global scheduler selects which of the applications is to execute at any time. Each application is then responsible to utilise its execution time by executing tasks according to its local scheduler. An addi-

[1] LITMUS^{RT} - Linux Testbed for Multiprocessor Scheduling in Real-Time Systems: `http://www.litmus-rt.org/`

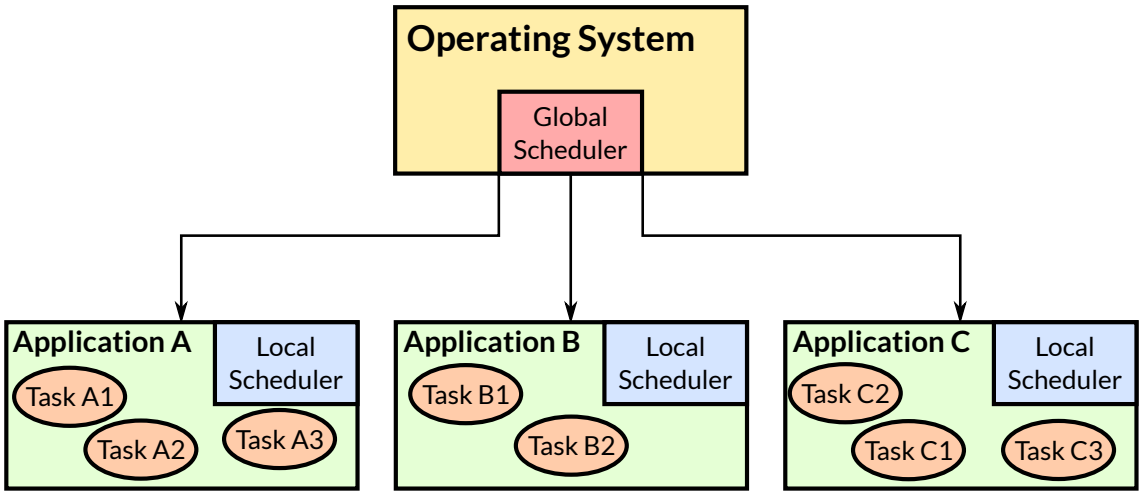


Figure 2.5: Hierarchical scheduler structure [70].

tional functionality of the global scheduler is to provide temporal protection to each of the applications [70]. This structure is illustrated in Figure 2.5.

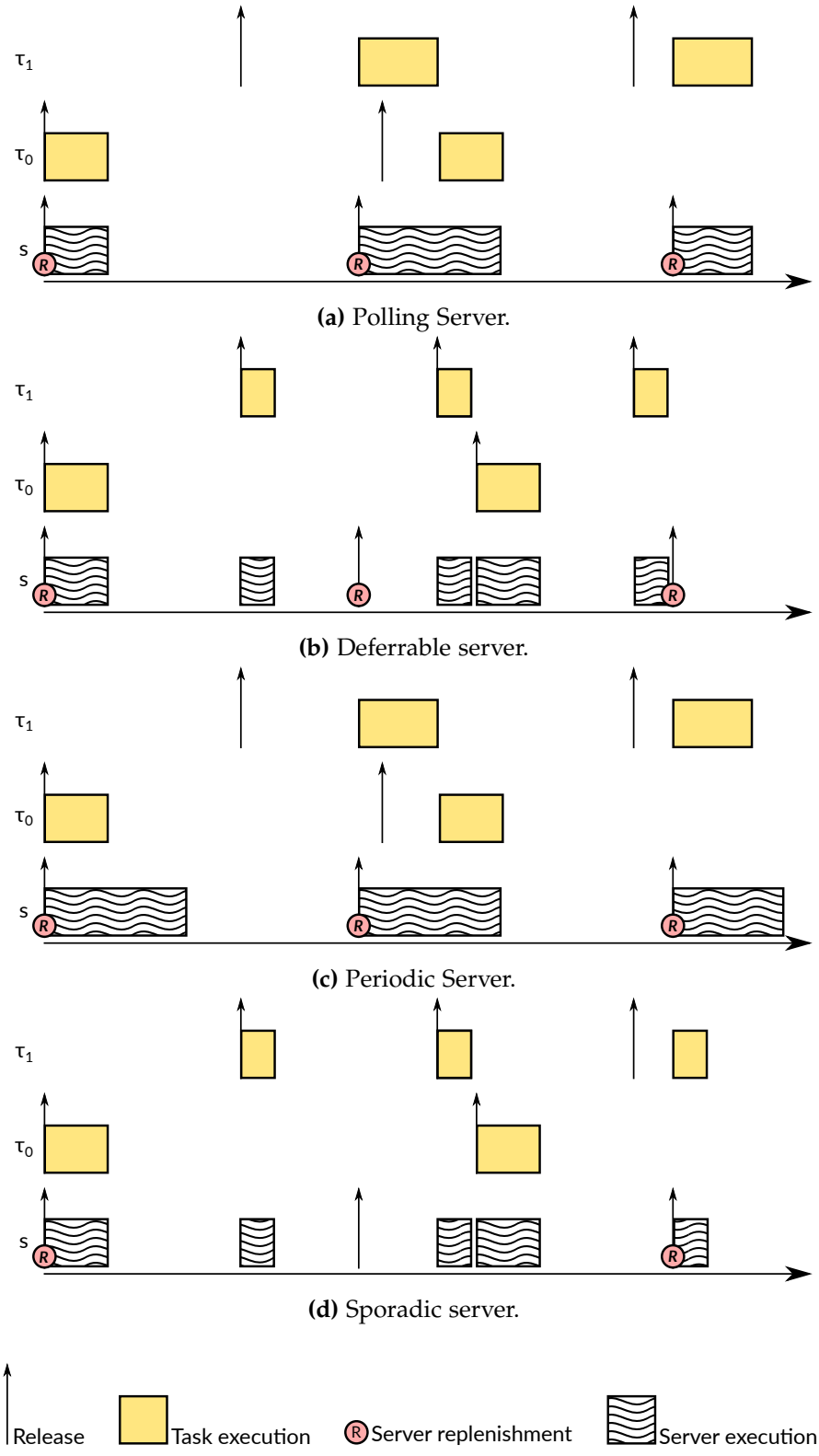
2.4.1 Execution Servers

The applications within a hierarchical system are scheduled using execution servers. Execution servers can be viewed as tasks that provide the applications with the time slots, during which they are allowed to execute. In this section we identify the main types of execution servers. Figure 2.6 summarises the behaviour of each of the identified execution servers by example. The capacity of the execution sever of the illustrated examples for polling, deferrable and periodic servers is set to $C_s = C_0 + C_1$, whereas for sporadic server it was set to $C_s = 2C_0 + C_1$, in order to clearly demonstrate its behaviour, where C_0 and C_1 are the execution times of τ_0 and τ_1 respectively.

Polling Server

A polling server is invoked at a set period and has a maximum capacity. All application tasks that are eligible for execution are run until the capacity is exhausted [23,30,93,97]. If the application is idle (ie. no tasks are eligible for execution), the remaining capacity is discarded. At the end of the period, the server's capacity is replenished. The polling server has low memory requirements, computational complexity and is relatively easy to implement [23]. However, it suffers from low performance.

The behaviour of the polling server is illustrated by the example of Figure 2.6a. At the start of the server period, τ_0 is released. The server is running and therefore τ_0 exe-



(a) Polling Server.

(b) Deferrable server.

(c) Periodic Server.

(d) Sporadic server.

The examples illustrated in this figure demonstrate the functionality of execution servers. We consider a simple non-preemptive system with two tasks, τ_0 and τ_1, that execute using the server s.

Figure 2.6: Examples of execution servers.

cutes to completion. Since there are no runnable tasks, the server's capacity is discarded before τ_1's release. The server is replenished after its period has passed and $t1$ starts executing to completion and τ_0 is released during that time. After τ_1 terminates τ_0 starts executing immediately to completion. The server's remaining capacity is discarded after τ_1's termination since there are no runnable tasks at that point.

Deferrable Server

A deferrable server [23, 30, 97] has a capacity and a fixed period at which it is invoked. The application tasks are allowed to execute for as long as the server capacity is not exhausted. If the application is idle, the server's capacity is preserved. The capacity is replenished after each period. Deferrable servers are computationally and memory efficient, and are relatively easy to implement [23]. They also outperform polling servers in terms of performance.

In Figure 2.6b the server starts executing and services τ_0 to completion. The server's capacity is saved and it stops executing. τ_1 is then released and serviced immediately since the server has enough capacity for it. After the server's replenishment it retains its capacity until a task becomes runnable (τ_0 in this case).

Periodic Server

A periodic server [23, 30, 31, 93] is similar to the polling server, with the difference that it continues executing and using the server's capacity even if the application is idle. The periodic server has similar implementation and computational complexity as the polling server approach. Although it outperforms deferrable servers in terms of schedulability [30], in some cases it may suffer from lower performance.

The periodic server in the example of Figure 2.6c starts executing, servicing τ_0, which executes to completion. The server continues to execute with no runnable tasks, exhausting its capacity. τ_1 is then released and waits until the server's replenishment, since it was previously exhausted. After the server is replenished τ_1 starts executing and τ_0 is released. Both tasks execute to completion, exhausting the server's capacity.

Sporadic Server

A sporadic server functions in a similar way to Deferrable Servers, with the difference that its capacity is replenished only after it is exhausted. The sporadic server is char-

acterised by good performance, however it is has greater implementation and computational complexity, as well as higher memory requirements than the other servers that were discussed [23].

The sporadic server of Figure 2.6d is assumed to have a greater capacity than the previous examples, for sake of demonstration: $C_s = 2C_0 + C_1$. The server starts executing as τ_0 is released, servicing it to completion. Its capacity is reserved until τ_1 is released, which is also serviced to completion. At its first period, the server has a remainder capacity of C_0 and is therefore not replenished. τ_0 is released and serviced to completion, exhausting the server's capacity. τ_1 is then released before the server's next period and waits until its replenishment. After the server capacity is replenished, τ_1 starts executing.

2.4.2 Work on Hierarchical Scheduling

A proportional-share scheduling algorithm was proposed by Goyal et al. [47] in 1996. The algorithm is referred to as Start-time Fair Queueing (SFQ) and was initially targeted for integrated services networks, however it was also deployed for use in a hierarchical scheduling environment. In the SFQ, each network packet is associated with a start and a finish tag. The packages are then scheduled in increasing order of their start tags.

SFQ is claimed by Goyal et al. [46] to achieve fair CPU bandwidth allocation for a uniprocessor system. Additionally, the algorithm provides bounds for the delay and throughput of the threads in a realistic environment. In 2000, Chandra et al. [27] identify that SFQ suffers from unbounded unfairness and starvation when used in a multiprocessor environment because of the inability to partition the CPU bandwidth appropriately.

In their 2009 paper, Åsberg et al. [85] present a hierarchical scheduling framework for use in the AUTOSAR infrastructure. The authors suggest integrating the global scheduler of the framework with the AUTOSAR Basic Software (BSW). The global scheduler can either interface with the OS or the existing scheduler in order to acquire access to standardised scheduling functions.

Following from their work in [85], Åsberg et al. [86] show that their proposed framework can be used in practice in their 2010 paper. In order to achieve this, they used the Times[2] tool, which supports schedulability analysis, formal verification and code

[2]Times - A Tool for Modeling and Implementation of Embedded Systems: `http://www.timestool.com`.

generation. Using the Times tool they were able to perform schedulability analysis under fixed-priority preemptive hierarchical scheduling. Additionally, they adjust the code generated from Times in order to be executable on VxWorks. Lastly, they state that there is significant difference between the response times of simulated and executed code.

In their 2012 work, Lackorzyński et al. [63] identify that there is incompatibility between mixed-criticality guests and the current virtualisation technology. They propose an interface that allows the guest operating systems to allocate budgets and switch between them. Specifically, the guests are provided budgets by the global scheduler in the form of *scheduling contexts* (SCs) during startup or on request. Each SC is described by its global priority and a budget that can be replenished if certain predefined conditions are met. The SCs are then mapped to virtual CPUs (vCPUs), allowing the VM to select which SC to run at a time on each of its allocated vCPUs. This approach is shown to require a very small number of modifications to the guest OSs code; 10 and 22 additional lines of code for FreeRTOS and Linux respectively. Additionally, the latency that was introduced from the activation of context switches was measured to be $0.4\mu s$ and $2.8\mu s$ for paravirtualised and fully-virtualised guests respectively, showing the difference in overheads between the two. The main limitation of this approach is that it assumes access to the guest OS source code and the ability to modify it.

2.5 Hypervisor Systems

One of the key challenges identified in Section 1.1 is the high ECU count, which contributes to development costs and complexity of the software and hardware in vehicles. A possible way to alleviate this is the use of virtualisation. *Virtualisation* is the use of software in order to integrate and concurrently execute multiple operating systems and applications on the same hardware. As explained in the previous chapter, this is often achieved using a Virtual Machine Manager (VMM) or Hypervisor (HV) in order to provide temporal and spatial isolation between the Virtual Machines (VMs) [58, 74].

Early work on virtualisation by Popek and Goldberg [82] in 1974 identify three characteristics of an HV. First, an HV needs to be able to provide its hosted VMs with an execution environment which is indistinguishable from real hardware. Second, execution is efficient by mapping a large subset of the virtual processor instruction set to a physical processor. Third, the HV has complete control over all hardware resources and

is able to allow or prohibit VMs access, according to the system configuration.

Popek and Goldberg [82] also identify a set of properties for HVs:

- *Efficiency:* all non-privileged instructions are executed directly on hardware.
- *Resource control:* it is impossible for a VM to interfere with any system resources that are not allocated to it.
- *Equivalence:* a VM produces the same results when executing as if it was executing without a HV.

The HV characteristics by Popek and Goldberg [82] refer to full virtualisation. With full virtualisation the applications in VMs can be executed without requiring any modifications [58]. In order to maintain the properties identified above in a fully virtualised environment it is necessary to have adequate hardware support. Specifically, allowing a VM to execute directly most of the time on the underlying hardware for efficiency requires that the HV will be able to identify attempts to execute privileged instructions. The HV is then responsible for checking whether the VM is allowed to perform the operation it attempted to and act accordingly.

Paravirtualisation was introduced as means of alleviating the lack of hardware support and to simplify the development of HVs. Specifically, in a paravirtualised environment, VMs execute directly on hardware using modified versions of their application code [58], using HV calls to replace the functionality of privileged instructions.

2.5.1 Review of Existing Hypervisors

This section contains a review of existing virtualisation systems. The criteria for selecting these systems to review is the availability of information on them and the relevance to the domain of this project. The virtualisation systems that were considered but not reviewed extensively are:

- **Freescale's embedded hypervisor** [45] is mainly targeted at high-end PowerPC chips and is primarily used for fast networking switches and infrastructure. The supported platforms typically have a considerably richer set of features than the targets typically used by ETAS Ltd (eg. Infenion TriCore), therefore this HV was not reviewed further.

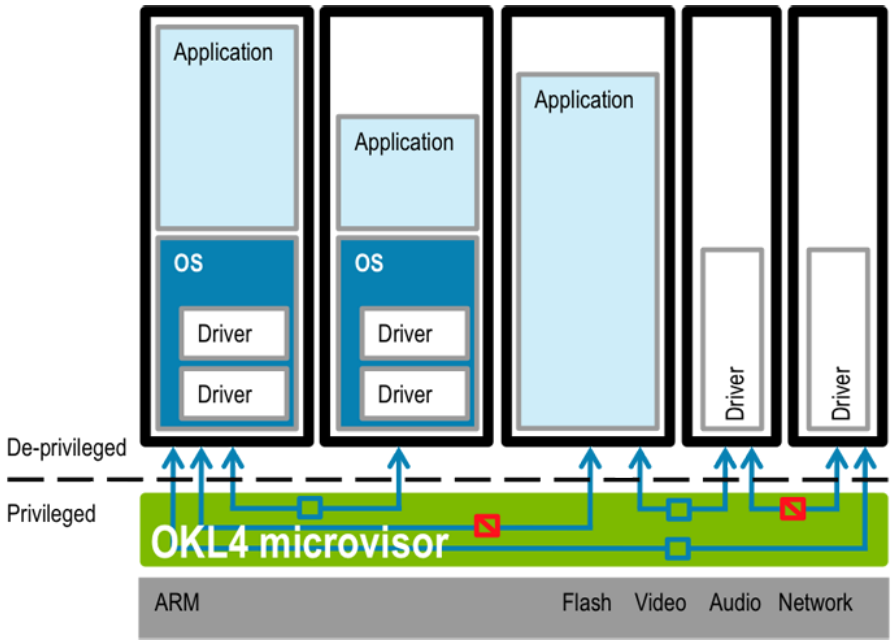


Figure 2.7: OKL4 microvisor with secure HyperCell™ technology [62].

- **INTEGRITY Multivisor** [95] was developed by GreenHills Software. Although there was very limited visibility of information available for it, the supported architectures (ARM, PowerPC and Intel) imply a requirement for hardware support for virtualisation and high-end Memory Management Unit (MMU) capabilities. This hypervisor therefore was not investigated further.

- **Wind River Hypervisor** [92] follows a paravirtualisation approach. The supported OSs are Wind River Linux and VxWorks. There is multicore support, however the supported processor architectures are x86 and PowerPC (MPC85xx upwards), which have considerably more resources than the hardware platforms typically used by ETAS Ltd.

For the purpose of this thesis, out of all the virtualisation systems that were studied, four will be reviewed: OKL4 [62], XtratuM [87], Xen [10] and PikeOS [99]. Table 2.1 summarises the key features of each of the reviewed hypervisors.

OKL4

OKL4 is a general purpose microkernel-based hypervisor developed by Open Kernel Labs, in order to provide minimal hardware abstraction to accommodate multiple operating systems [51, 62, 81]. The supported hardware architectures for OKL4 are ARM,

	OKL4	**XtratuM**	**Xen**	**PikeOS**
Supported Architectures	ARM, Intel, MIPS	LEON2, LEON3, LEON4, x86, ARM Cortex R4f	x86, x86_64 and ARM. Legacy support for IA64	ARM, MIPS, PPC, SH4, x86, x86-64, SPARC V8, LEON
Virtualisation approach	Full virtualisation and paravirtualisation	Paravirtualisation	Full virtualisation and paravirtualisation	Paravirtualisation
Static memory allocation	Yes	Yes	Yes	Yes
Multiple partitions	Yes	Yes	Yes	Yes
Temporal isolation	Partly	Yes	Yes	Yes
Spatial Isolation	Yes	ARINC-653	Yes	ARINC-653
Inter-process communication	Yes	ARINC-653	Partial support via shared memory	ARINC-653
Direct device access	Yes	Yes	Yes	Yes
Shared peripheral support	Yes	Yes	Yes	Yes
Interrupt virtualisation	Yes	Yes	Yes	Yes
Real-time support	Partly	Yes	Yes	Yes
Scheduling	Round-robin, priority-based preemptive scheduler	ARINC-653 based fixed cyclic scheduling	Credit, Credit 2, RTDS and ARINC-653	Combination of time-driven and priority-based scheduling
Bare-metal	Yes	Yes	Yes	Yes

Table 2.1: Summary of reviewed hypervisors

MIPS and Intel. Since ARM and MIPS are already being used in the automotive industries for ECUs, and because OKL4 is characterised by efficiency and following a minimalist approach, it was chosen for review.

The main claim of OKL4 is that it provides the facilities to host multiple isolated VMs. Each VM runs in a non-privileged mode within the boundaries of a *secure cell*. These cells are part of the Secure HyperCell™ Technology, which offers an infrastructure for the development of complex software systems using simpler components. A key feature of using this technology is spatial protection, using the underlying hardware MMU.

Secure cells are also used for resource management by allocating each resource to the appropriate VMs. This enables the microkernel to identify invalid accesses to resources that could potentially violate the integrity of the system. Although the OKL4 microkernel provides access protection, the system is not deadlock free. Instead, it offers deadlock detection functionality that terminates all threads in the dependency chain [81]. This behaviour can potentially cause problems that affect the integrity of the system, as it may erroneously terminate a VM.

Another claim of OKL4 is its capability to support real-time OSs. Specifically, it is stated that this is due to the performance optimisations to functions that have a big impact on real-time response. These functions include context switching and interprocess communication (IPC). Additionally, it allows the guest OSs to allocate priorities to their tasks, in order to help achieve the desired real-time properties. Even though OKL4 seems to be very optimised, no analysis of the real-time properties was found.

OKL4 also tries to make the development of device drivers easier. Instead of having to develop a device driver for each VM, OKL4 supports sharing drivers from general purpose OSs, which typically offer a richer set of facilities. Specifically, for each device a physical driver is needed, which is then used by the guest OSs with the implementation of virtual drivers. This can be interpreted as device virtualisation.

Temporal protection in OKL4 is enforced by a round-robin, priority-based, preemptive scheduler. The guest operating systems are able to implement their own local scheduling policy for ordering the execution order of their tasks. OKL4 is responsible for determining which VM executes at any time with respect to the priority of each VM. Whenever the CPU time allocation of a VM is used up, it is preempted in order to allow the next VM to execute.

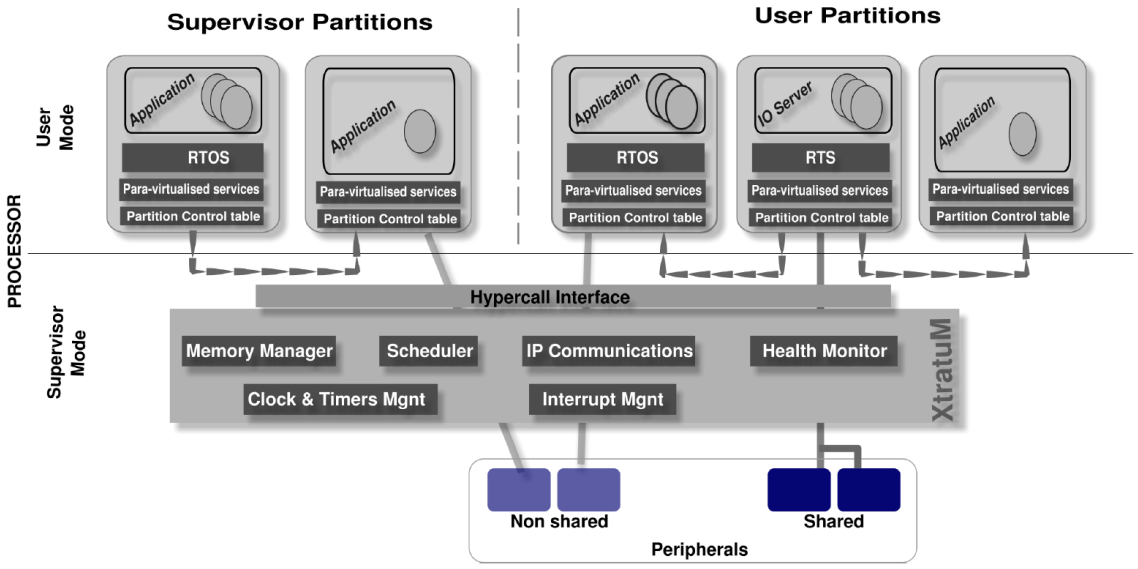


Figure 2.8: XtratuM architecture [29].

XtratuM

XtratuM [29,72,73,87] is a hypervisor, that was designed for the use in real-time safety critical systems by following the ARINC-653 standard for spacial and temporal partitioning in real-time operating systems for the avionics domain. Since avionics is a real-time safety-critical domain, it offers properties that are also relevant in the automotive domain.

A requirement for XtratuM is the modification of guest OSs in order to work with XtratuM [29]. Specifically, privileged instructions are replaced with hypervisor calls (hypercalls) that are processed by the hypervisor, which acts as a service provider. This technique is referred to as paravirtualisation. Para-virtualisation offers two main benefits [58] first it simplifies the hypervisor design and implementation, and second it is generally faster than having full virtualisation. The obvious drawback is the requirement to port an OS to work with XtratuM.

Figure 2.8 illustrates an architectural overview of the XtratuM hypervisor [29]. This architecture can be divided in three layers: hardware-dependent layer, internal service layer and virtualisation service layer. The hardware-dependent layer includes a set of driver that manage low-level, hardware-dependent functions such as interrupt handling, timers, memory management etc. In order to hide the complexity, a hardware abstraction layer (HAL) is used. The internal service layer is responsible for booting the system, as well as the provision of essential C functions like `memset` and `strcpy`. Lastly, the vir-

tualisation service layer functions as a service provider that will enable the hypervisor to support the paravirtualisated partitions.

XtratuM always runs in privileged mode, whereas the guest OSs execute in a non-privileged mode [29,73]. Given that the VMs do not share any memory, and this property cannot be modified in a non-privileged mode, XtratuM provides strong spacial isolation.

In order to provide temporal isolation, each partition is scheduled for execution using the hypervisor's fixed cyclic scheduler [29,73]. On one hand, due to the simplicity of the scheduler, it is possible to reason about the system's real-time properties. On the other hand, this can prove to be insufficient to support multiple OSs with different levels of criticality.

In order to implement IPC, ARINC-653 is used as a standard to provide a port-based communication system. Since ARINC-653 is a standard used in the avionics industry, which is has high robustness and safety requirements, it can be said that XtratuM provides robust communication mechanisms.

The guest OSs are not able to do interrupt and trap handling without support from XtratuM. Specifically, when a trap or an interrupt occurs, the hypervisor determines whether it should be forwarded to a guest OS according to a configuration file. If so, the exception is propagated to the appropriate guest OS, which then handles it in a non-privileged mode. This indirection has some impact in performance, however it ensures that a guest OS cannot violate the system's integrity when exceptions are raised.

Xen Project

Xen Project is an open source hypervisor, developed by Barham et al [10] in 2003, initially for x86 architectures and was later ported to support ARMv7 and ARMv8 architectures [67]. Xen is used in many applications, including server virtualiation, desktop virtualisation and embedded systems [69].

The architecture of Xen is summarised in Figure 2.9. The Xen HV sits directly on top of the hardware and to manage CPU, interrupts and memory accesses. There are two supported modes of virtualisation: paravirtualisation, and full virtualisation. Virtual machines using full virtualisation can also use paravirtualisation features in order to improve performance with fewer guest modifications. The guest VMs are isolated from one another and prohibited from having privileged access to hardware or IO.

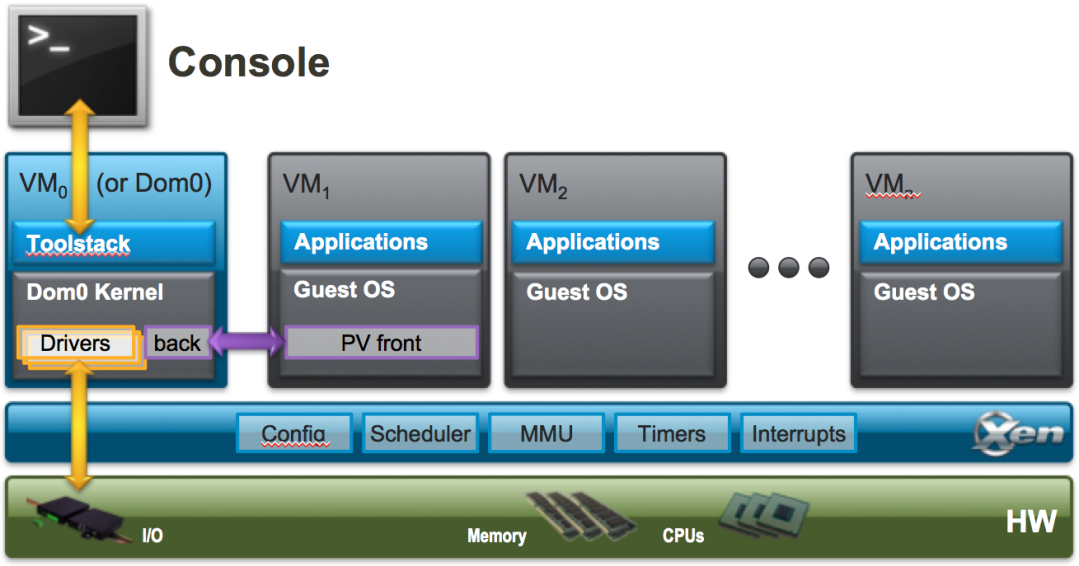


Figure 2.9: Xen Project Hypervisor architecture [69].

At system initialisation, the first VM created is Domain 0, also referred to as the control domain. Domain 0 offers core functionality required by the Xen Project Hypervisor to function. This specialised VM has direct access to hardware and is responsible to mediate IO operations and communication between VMs. It also provides a control interface, which allows the system to be controlled by the outside world. Domain 0 is also able to create, destroy and configure VMs using the Toolstack.

The currrent version of Xen (v 4.5) supports four schedulers [68]: Credit, Credit 2, RTDS [107, 108] and ARINC 653. Credit 2 and RTDS are currently in an experimental phase, however they are expected to be supported in the next release (v 4.6). Credit and Credit 2 are proportional share schedulers, where each VM is allocate a weight and a cap. The weight represents the portion of CPU time the VM is allowed to use with relative to the weights of the other VMs. The cap is the maximum CPU time the VM will receive. The Real-time Deferrable Server (RTDS) scheduler [68, 107, 108] is designed to provide a guaranteed amount of CPU time to each VM. The RTDS scheduler allocates each VCPU is allocated a budget and a period. The VCPUs then function as deferrable servers and are scheduled using preemptive EDF.

PikeOS

PikeOS [99] is a hypervisor developed by SYSGO AG. The focus of this hypervisor is to use virtualisation in embedded safety-critical domains [58, 59]. Similarly to OKL4,

Figure 2.10: PikeOS partitioning according to ARINC-653 [99].

PikeOS is also a microkernel-based hypervisor. It is available on a wide variety of hardware architectures: ARM, MIPS, PPC, SH4, x86, x86-64, SPARC V8, LEON. As suggested by Figure 2.10, ARINC-653 is used in order to ensure safe partitioning.

For non-safety-critical OSs, PikeOS supports dynamic memory allocation [59]. Specifically, the guest OS is able to request for additional memory or release memory at runtime. In the case of safety-critical real-time OSs, a static memory allocation is supported. Whereas, for richer OSs it is possible to have requests for additional memory or for releasing memory. A potential problem of supporting both dynamic and static memory allocation is the temporal interference between two VMs due to changes in the memory mapping.

In order to simplify and increase the performance of accessing IO devices, drivers are implemented with user-level code in the VMs [58]. Therefore, in order to access an IO device a guest OS only requires read/write access to the memory location where the device registers are mapped to. This avoids the problem of having to virtualise the devices, therefore having good performance without jeopardising the system's integrity. This implies that PikeOS supports device virtualisation, however no evidence was found to indicate support of shared devices.

As stated by Kaiser [58], PikeOS supports non-real-time, time-driven real-time and event-driven real-time VMs. The VMs in the system are partitioned into a set of domains, τ_i [56, 57]. The first domain, τ_0, is referred to as the background domain and the rest as foreground. The VMs are allocated priorities, with real-time VMs having medium to high priorities and non-real-time a common low priority. The event-driven and non real-time VMs are allocated to τ_0, whereas the time driven ones are spread

across the foreground domains, $\tau_i (i \neq 0)$. At each point in time two domains are active at a time: the background and one of the foreground ones. The scheduler then decides which VM is allowed to execute according to the priorities of the VMs in the active domains. This scheduling approach addresses the ARINC-653 fixed cyclic scheduler's low-utilisation issue.

Similarly to XtratuM, PikeOS is based on the ARINC-653 standard for IPC. Apart from using IPC to establish communication between VMs, it is also used for interrupt handling. Specifically, when an interrupt is raised, the hypervisor determines whether it is to be handled by a VM or not. If so, then it is converted to an IPC message, which is sent to the corresponding VM. Then, the VM is responsible for processing the IPC thereby handling the interrupt in a non-privileged mode.

Verbeek et al [102] worked on the formalisation of PikeOS API calls. In their paper, they propose a methodology for developing high-level functional specifications of separation kernels. They used their methodology to formalise and prove transitive non-interference of the PikeOS API calls that require secure information flow.

2.6 Industrial Context and Research Gap

This section sets the industrial context of this research project, by summarising the industry practices and trends as observed throughout this research project. First we provide a brief description of BMW's Domain Controlled Architecture. RTA-HV, which is a real-time hypervisor developed by ETAS Ltd, is then reviewed in order to identify the gap addressed by this research project.

2.6.1 Domain Controlled Architecture

The high complexity of the electronics and software in vehicles motivated the development of the Domain Oriented Architecture by BMW, which proposes a partitioning method where ECUs are allocated to Domain Control Units (DCU) [88–90]. Although the decentralised approach, which is currently the most common practice in the automotive industry, offers a high level of flexibility, it also increases the development costs. Moreover, there is an increasing need for cross-ECU communication, therefore potentially overloading the communication buses.

An example of a domain oriented architecture is presented in Figure 2.11. The ECUs

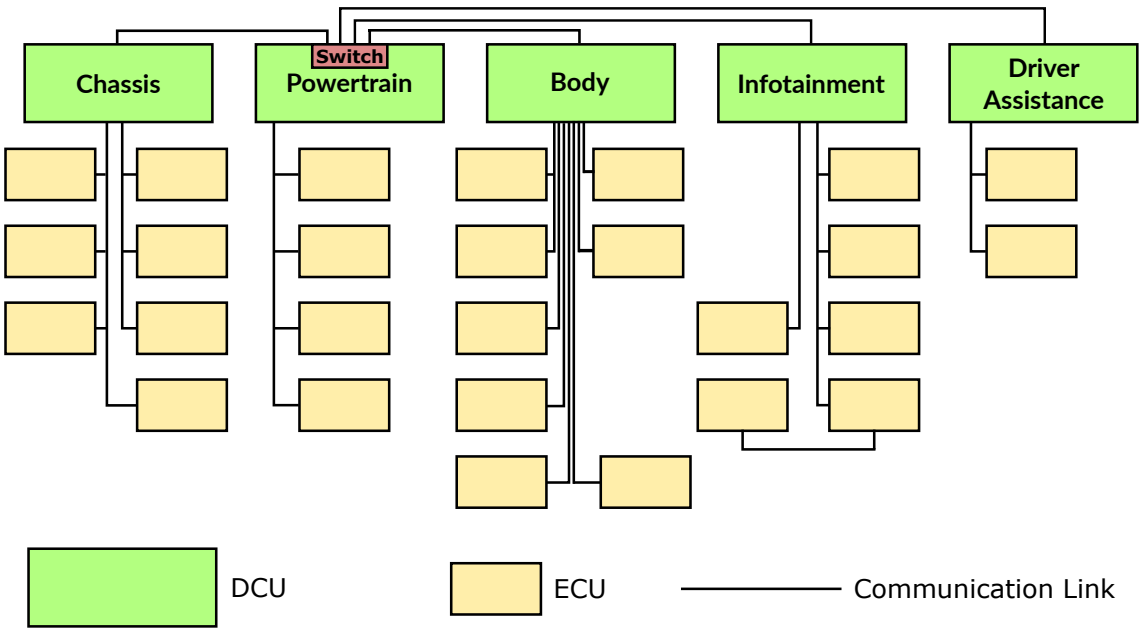


Figure 2.11: Domain Oriented Architecture [88].

are clustered with respect to their functionality. The ECUs within a domain communicate using a number of communication channels, that are connected to their respective domain controller. This arrangement ensures that the communication paths between tightly related software components are kept as short as possible. Domain controllers are connected with one another using fast links, such as Ethernet, allowing communication across domains.

2.6.2 ETAS Hypervisor (RTA-HV)

A current trend in the automotive industry is the use of multicore hardware platforms for the reduction of power consumption [90]. Multicore is supported since AUTOSAR 4.0, however adapting legacy software to take advantage of multicore is a difficult job. Virtualisation is identified as a possible solution for the integration of multiple ECUs into a single DCU [90]. RTA-HV, which is a deeply embedded hypervisor developed by ETAS Ltd was investigated. RTA-HV is a multi-core bare-metal hypervisor, implemented on AURIX and ARM processors [78, 90].

Figure 2.12 summarises the abstract architecture of the RTA-HV. RTA-HV is logically located directly above the hardware and executes in a privileged mode on the processor. It primarily follows a paravirtualisation approach, providing the VM applications with an API of hypervisor calls and a set of emulated instructions that can be decoded and

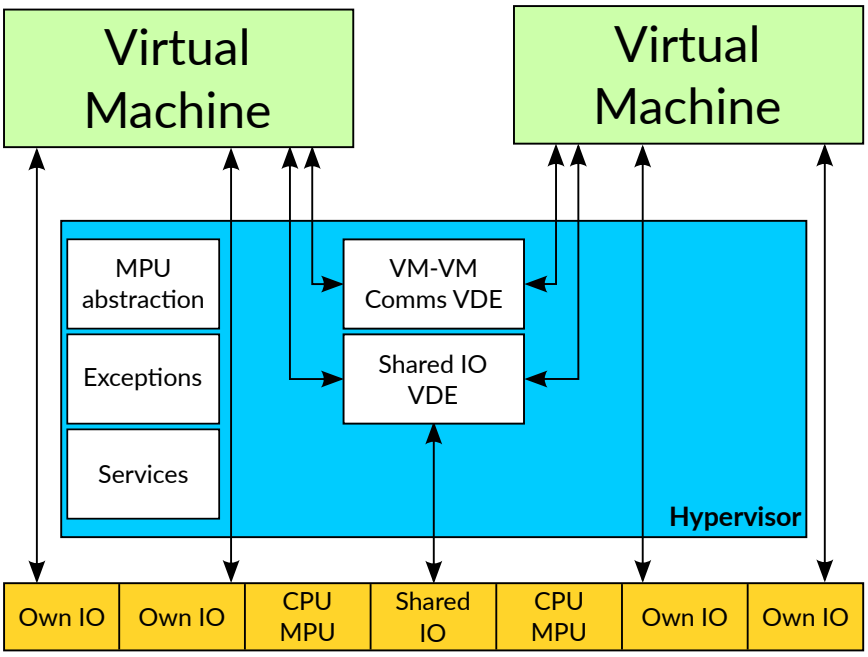


Figure 2.12: Abstract architecture of RTA-HV [79, 90].

executed by the HV on the VM's behalf. The Virtual Machines or Partitions execute in user mode, which restricts their access to privileged states.

Device Management

The VM applications can access non-shared IO devices both directly and through the hypervisor. Direct access to IO devices has no performance degradation and, in the case of AUTOSAR, requires no changes to MCAL, however depending on the system configuration it may not be possible due to hardware limitations. Specifically, in a system with many IO devices the number configurable memory areas provided by the MMU/MPU may not be sufficient. This limitation can be overcome by allowing the VM applications to access IO through HV calls. In this case, the HV is responsible for checking whether the HV call is permitted before it is serviced.

Access to shared devices is managed using Virtual Device Emulators (VDE). VDEs provide an abstract interface that allows VMs to access devices. The primary use case for VDEs is to allow sharing of hardware resources between HV in a manner that prohibits data corruption due to concurrent access. VDEs are treated as part of the hypervisor and are executed in a privileged mode. Apart from managing concurrent access to hardware devices, VDEs can also be used to simplify potentially hard-to-use device interfaces, or act as a virtual communication device between HV, allowing for fast inter-partition

Benchmarks	Number of Cycles		
	Native	Virtualised	Overhead
Two `nops`	2	3	50%
Setting `BIV`	7	211	2914%
Four writes to the `PSW`	2	469	23350%
Read from `STM_CLC` (`HV_ReadUI()`)	5	164	3180%
Retrieving priority	5	13	160%
Setting priority	2	166	8200%
Write to interrupt register (`HV_WriteUI()`)	6	153	2450%
Return from interrupt	13	27	107%
Interrupt entry	3	105	3400%
Entering trap	26	100	284%
Return from trap	11	13	18%
Set untrusted mode	10	1503	15030%

Table 2.2: Timing Measurements and Virtualisation Overheads of Sample Application on Infineon AURIX TC27x [90].

communications without requiring access to external devices, such as CAN controllers. Part of the RTA-HV specification is an API for the development of custom VDEs.

Partition Interrupts

RTA-HV assumes an interrupt-driven system. Interrupts have unique priorities and are either handled by the partitions or VDEs. The priority level of any partition interrupt is strictly lower than the priority of VDE interrupts. Temporal protection is enforced by prohibiting partitions to disable all interrupts or raise the core's Interrupt Priority Level (IPL) above that of their highest priority interrupt.

The minimisation of overheads, implementation/porting costs and low interrupt latency drove the design of RTA-HV to uniquely map each VM to a single core. Table 2.2 summarises the main virtualisation overheads, comparing them to the overheads of a non-virtualised system on Infineon AURIX TC27x hardware. Forwarding interrupts to the applications running in the VMs has significant overheads due to lack of hardware support. The main source of the additional latency is the need to emulate the `BISR` (Begin Interrupt Service Routine) instruction. `BISR` is used to set the core in an interrupt handling state by saving the execution context and changing the current priority level.

Integration Overheads

Without the use of virtualisation, integrating AUTOSAR SWC can have high costs [79] due to the complexity of RTE and BSW, which are configured using many parameters,

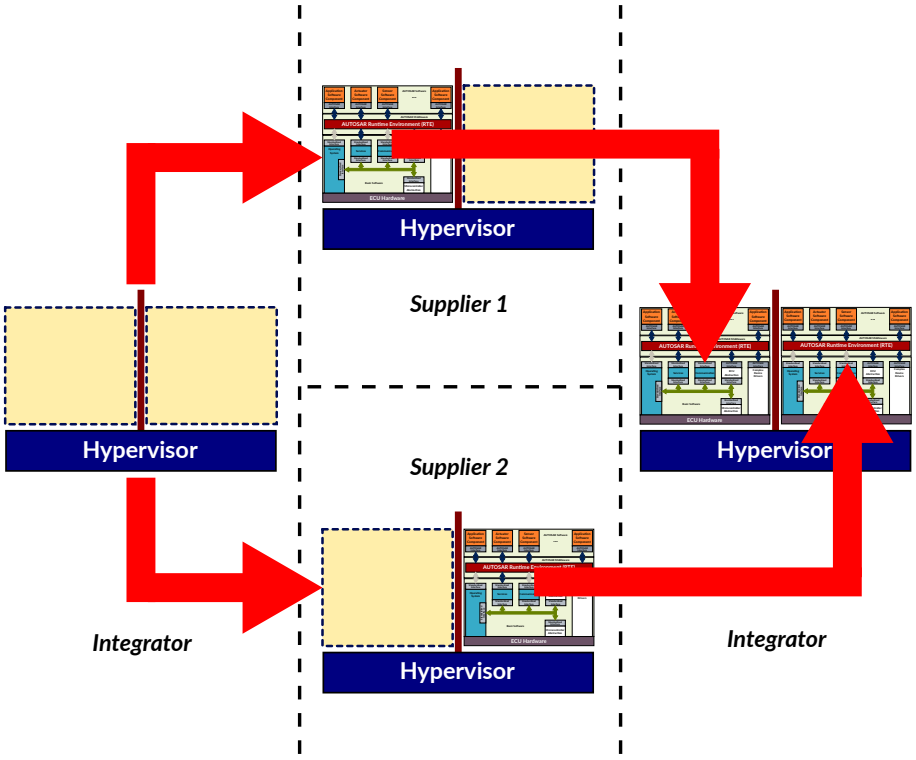


Figure 2.13: Hypervisor based cross-company workflow [79].

as stated in Section 1.1. In order for applications to work in a RTA-HV VM some porting effort is required, viewing the ECU code as a single entity, since it follows a paravirtualisation approach. The first part that needs to be modified is the startup code, since the HV is responsible for some of the initialisation code. The OS code also needs to be modified by replacing privileged operations, such as changing interrupt priorities, with HV calls. Porting an application to work on top of RTA-HV is expected to be simpler in comparison to the integration of individual SWCs. Enabling inter-VM communication is another area that requires some porting effort. Specifically, a VDE needs to be created to emulate a communication interface between VMs.

Figure 2.13 summarises the cross-company workflow for the integration of a HV system using components from multiple suppliers. The Integrator provides the Suppliers with example development hardware and the hypervisor configuration. The Suppliers develop their ECU software to work within a virtual machine and deliver the complete ECU images to the Integrator. The Integrator is then responsible for flashing the ECU images and the HV on the same ECU, perform the integration testing.

2.6.3 Research Gap

RTA-HV is successful in reducing the ECU count with fewer, more powerful units. The one VM per core approach offers the lowest possible latency on a virtualised system, however it can potentially result in the underutilisation of resources. This problem becomes more apparent when considering the integration of ECUs with low CPU utilisation, such as the heater controller or the wiper module. The identified research gap is the investigation of allowing multiple VMs to execute on a single core, in order to maximise the resource utilisation, while taking into account the need for minimal interrupt latency and overheads.

From a scheduling point of view, the methods used by traditional hypervisor systems assume limited to no visibility with regards to the execution of the applications within the VMs. In the case of having automotive software running in the VMs, some structural information is available. Additionally, the Integrator and Suppliers of Figure 2.13 can be the same company, or collaborate in allowing visibility to some degree. A research gap is therefore identified, where a scheduling method for a hypervisor system can schedule and allocate the CPU resources at the task level rather than the VM level.

2.7 Summary

In this chapter we present a review of the relevant literature, as well as identify the industrial relevance of this research project. We provide an introduction to timing predictability and real-time scheduling concepts. The concept of mixed criticality scheduling is then introduced, providing an overview of the literature relevant to this research project. The focus of this chapter then shifts to hierarchical scheduling and hypervisor technology. After both topics are introduced, we provide a review of existing hypervisor systems. Having reviewed the relevant literature, we then proceed to provide the industrial context and identify the research gap using the experience of collaborating with ETAS Ltd.

In the next chapter we propose a scheduling model that enables the hypervisor to take advantage of having some visibility inside the VMs at the task level.

CHAPTER 3

System Architecture

A modern vehicle may contain software from different suppliers, as well as a mixture of legacy and modern code. Modifying legacy code in order to run in a paravirtualised system can be an expensive process. Therefore, in the case of virtualising legacy systems, these modifications should be kept to a minimum. In modern code, such as AUTOSAR-based OSs, the modularity of the software can be taken advantage of to provide the additional flexibility offered by our approach.

In this chapter, we present an architectural design for a hypervisor system and a scheduling framework that supports the execution of multiple applications on a single core. The proposed architecture is highly driven by the automotive industry requirements, while taking into consideration the ISO 26262 standard.

(a) Current configuration of RTA-HV.

(b) Configuration of the proposed architecture.

Figure 3.1: Comparison between the current version of RTA-HV and the proposed architecture.

3.1 Requirements and Assumptions

Figure 3.1a summarises the currently supported configuration of RTA-HV. As stated in Section 2.6.1, RTA-HV, the hypervisor developed by the industrial sponsor, supported a one-to-one partition to CPU mapping during the work done towards this research project. The high level requirement of the proposed architecture is the support for more than one partition per CPU. This requirement is decomposed into lower level requirements based on domain-specific requirements as set by the industrial sponsor and compliance with the automotive standard ISO-26262 [54]. As dictated by ISO 26262-1:2011(E) Definition 1.49 [54], Freedom From Interference (FFI) is defined as:

"absence of *cascading failures* between two or more *elements* that could lead to the violation of a safety requirement"

A cascading failure is defined as a failure of one element in the system, which can cause a failure in another element. In terms of the proposed hypervisor architecture, the partitions are considered as the constituent elements. The hypervisor is the mechanism for enforcing the necessary isolation between the partitions.

3.1.1 Spatial Isolation

A key guideline introduced by ISO 26262-6:2011(E) Annex D is FFI [54,88]. Content corruption and protection from read/write access to the memory space of other partitions are identified as potential effects of faults in the software components. In hypervisor systems, these effects are mitigated by enforcing spatial isolation. Specifically, partitions are prohibited from accessing and modifying the hypervisor's or another partition's memory space. Ensuring that a partition is unable to access the memory space of another partition (read or write) is also a protection mechanism for IP rights, which partly falls in the *security* use case of Section 1.2.

Modern processors support spatial protection, using either their memory management unit (MMU) or memory protection unit (MPU). The MMU/MPU is configured directly by the hypervisor, as shown in the example of Figure 3.2. It is assumed that the hypervisor is allowed to have full read/write access to the entire memory space. The partitions only have access to their own memory space. Inspired by RTA-HV, in the case where a partition attempts an illegal memory access, it will be restarted by the hypervisor, unless an alternative recovery routine is defined.

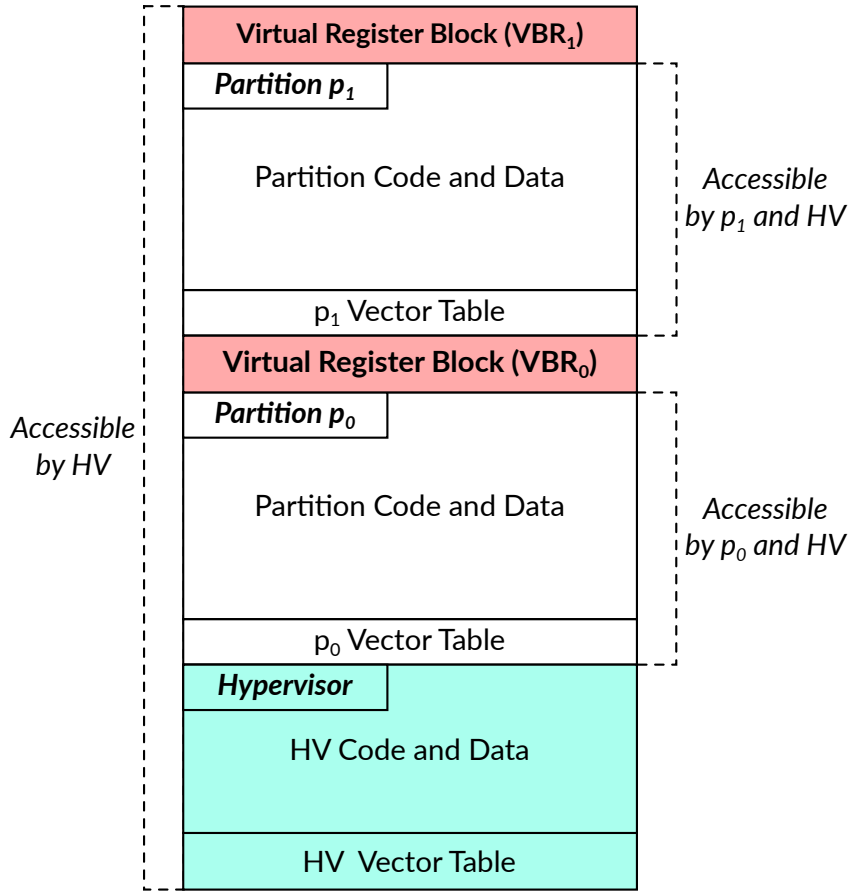


Figure 3.2: Example of logical memory layout in a two-partition system.

The Virtual Register Blocks (VRB) are memory regions that are only accessible by the hypervisor and are associated with their corresponding partitions. VRBs are used to store the processor state information during preemption, to allow suspending and resuming the execution of partitions.

3.1.2 Temporal Isolation

The primary requirement of the proposed architecture is the support for multiple partitions executing on a single processor. The hypervisor must ensure the temporal isolation between partitions, by enforcing FFI. Specifically, failure of one partition to meet its temporal requirements must not cause other partitions or the hypervisor to miss their deadlines. Temporal isolation is achieved using execution servers, as described in Sections 3.3 and 3.5.

It is assumed that each partition in the system may contain event-driven tasks, time-driven tasks or a collection of both. A key driver is the minimisation of the response

time of event-driven tasks. The minimisation of event-driven task response times is a requirement set by the industrial sponsor. This can potentially reduce the overall resource utilisation of the system, as it can introduce a tradeoff between utilisation and latency. The use of separate execution servers for each type of task, as discussed in Section 3.3, aims to meet the need for low latency of event-driven tasks, as well as maximise utilisation of time-driven tasks.

The developed architecture must have low and predictable implementation overheads. This requirement is also present in AUTOSAR-based OS. The use of FPS in AUTOSAR meets this requirements, as it is relatively simple to implement and analyse. Additionally, FPS has low implementation overheads, which are incorporated in the schedulability analysis described in Section 3.7.

3.2 Task Model

Traditional hypervisor scheduling approaches were developed assuming no visibility at the task level in the partitions. In this section we define a flexible task model, which allows exploiting visibility, where that is available, taking into account implementation overheads. First, we classify tasks as periodic or sporadic. Periodic tasks are strictly periodic, whereas sporadic are event-triggered tasks with a known minimum interarrival time. The main motivation behind the proposed scheduling method is that sporadic tasks require quick response times, while periodic ones can be serviced in a more efficient, lower overheads approach. All operations performed by the hypervisor are executed in a non-preemptive manner and are described as highly predictable pieces of code.

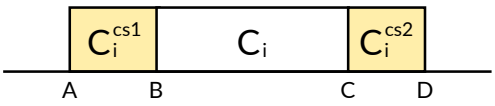


Figure 3.3: Task structure.

The order of execution of the regions of a task τ_i is shown in Figure 3.3. A task τ_i is defined by the tuple $(C_i^{cs1}, C_i, C_i^{cs2}, T_i, P_i)$:

- C_i^{cs1}: the scheduling and context switching overheads required before the execution of the main task body. In sporadic tasks, this section is performed by the hypervisor and is therefore cannot be preempted. Periodic tasks can be preempted

by the hypervisor during this section but not by other periodic tasks of the same partition. This is because this operation is performed directly by the partition.

- C_i: the time required for the task's main body to execute. Partition tasks (periodic and sporadic) can be preempted while in this section, however hypervisor tasks run with no preemption.

- C_i^{cs2}: the overheads of terminating the execution of the task. The preemption rules that apply for C_i^{cs1} also apply during the execution of this section.

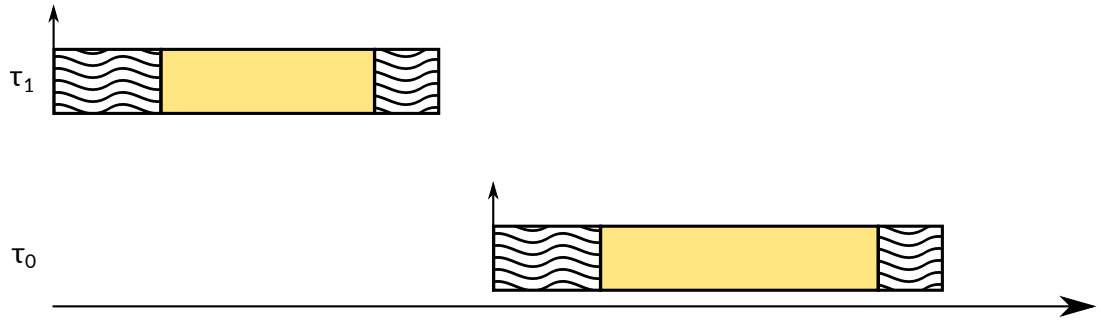
- T_i: the period or minimum interarrival time of the task.

- P_i: the priority level of the task. The smaller the numeric value of P, the higher the priority.

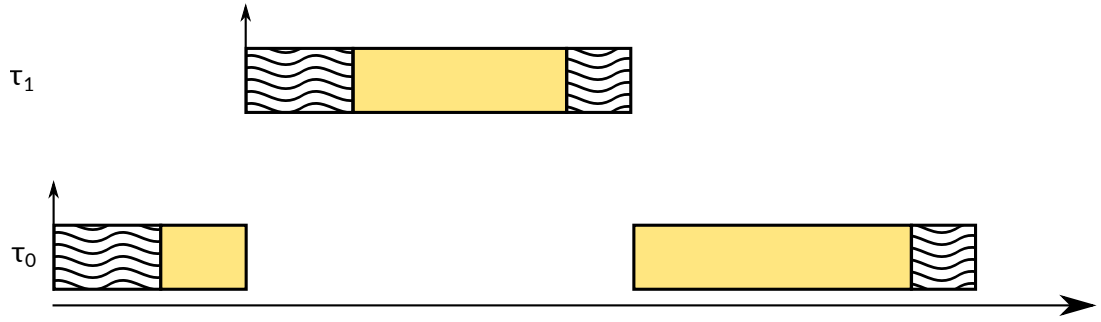
Figure 3.4 demonstrates the preemption mechanism described in this section, using an example of a simple system with two sporadic tasks, τ_0 and τ_1. In the simple case, as shown in Figure 3.4a, τ_1 is released, the hypervisor forwards the event to the partition, which then executes to completion before the release and execution of τ_0.

The second example, shown in Figure 3.4b, demonstrates preemption, focusing on the case where a higher priority task is released during the execution of a preemptive region of a lower priority one. In this example, τ_0 is released and is forwarded to the appropriate partition, which in turn starts executing the task body. During the execution of the preemptive region of τ_0 a higher priority task, τ_1, is released. The execution of τ_0 is suspended, allowing for τ_1 to be forwarded and executed to completion. The execution of τ_0 is then resumed until its termination.

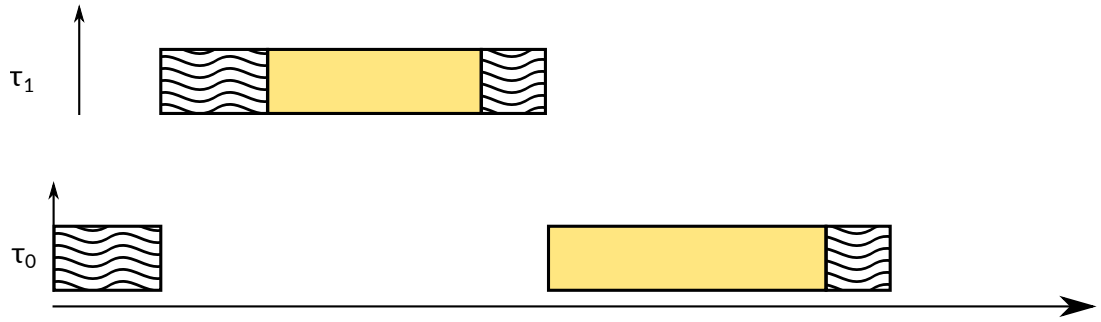
In Figure 3.4c, τ_0 is released and the hypervisor is executing the forwarding routine when τ_1 is released. During the execution of the forwarding routine, all interrupts are masked and therefore τ_1 is blocked. The execution of the forwarding routine of τ_0 terminates, unmasking the interrupt line associated with τ_1. The forwarding routine of τ_1 is performed, allowing τ_1 to execute to completion since it is the highest priority task in the system that is eligible to execute. After the termination of τ_1, the execution of τ_0 resumes.



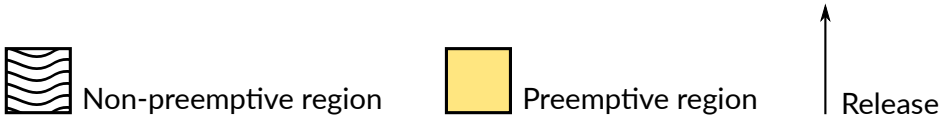
(a) Simple case of sporadic task execution.



(b) Aperiodic task execution without blocking.



(c) Aperiodic task execution with blocking.



The examples in this figure demonstrate the execution of sporadic tasks. We consider a system with two sporadic tasks, τ_0 and τ_1. For simplicity, we assume that the servers associated with both tasks always have enough capacity and τ_1 is of higher priority than τ_0.

Figure 3.4: Aperiodic task execution examples.

3.3 Execution Servers

Our approach aims to minimise the response time for event-triggered tasks, while at the same time maximise schedulability and enforce temporal protection between the different partitions[1]. The CPU time is shared between the partitions using execution servers. Temporal protection is achieved by prohibiting partitions to execute for more than their servers' capacity. Each server is associated with a hypervisor task, which is responsible for replenishing its capacity.

3.3.1 Event-driven Execution Servers

Aperiodic tasks are released in response to events and therefore need to be serviced with the lowest possible latency. To facilitate this requirement, two types of execution servers were considered: sporadic servers and deferrable servers.

Sporadic servers function in a similar manner to deferrable servers. The main difference between the two is that the capacity of sporadic servers is only replenished after it is depleted. Using sporadic servers has the benefit of potentially requiring fewer replenishments, which has an impact on the required overheads. As was identified in Section 2.4.1, sporadic servers suffer from higher implementation overheads, computation complexity and memory requirements [23, 96]. Sporadic servers are therefore rejected, as one of the main requirements for the proposed architecture is low overheads.

Deferrable servers have the potential drawback of being replenished more than necessary, which translates to increased overheads. The memory and computational requirements of deferrable servers are typically lower than sporadic servers [23,96], therefore sporadic tasks execute using deferrable servers that are assumed to always have enough capacity to service all event-driven tasks, given their WCET, hypervisor overheads and period. With the use of a deferrable server, no server capacity is expended when the system is idle and events are serviced as they arrive, provided they have the highest priorities in the system.

Deferrable servers offer low response time for the sporadic tasks, however they are inferior in terms of schedulability in comparison to periodic servers. This is because deferrrable servers can use their capacity back-to-back, making the jitter of the serviced tasks $J_i = R_s - C_s$ (see Section 3.8.3).

[1] A similar approach by Missimer et al. [76] using sporadic and priority inheritance bandwidth preserving servers (PIBS) was published during the write-up of this Thesis.

3.3.2 Time-driven Execution Servers

For servicing time-driven tasks, the focus shifts from low latency to high schedulability. Therefore two execution server candidates are condidered: polling servers and periodic servers.

Polling servers and periodic servers have similar behaviour. At the start of the period, the servers are at maximum capacity and service tasks until their capacity is depleted. Both servers have low computational and memory requirements. The difference between polling and periodic servers is that polling servers lose their remaining capacity when no task is ready to execute. Although the additional capacity can potentially be utilised by other entities in the system, polling servers suffer from low performance, therefore they were rejected.

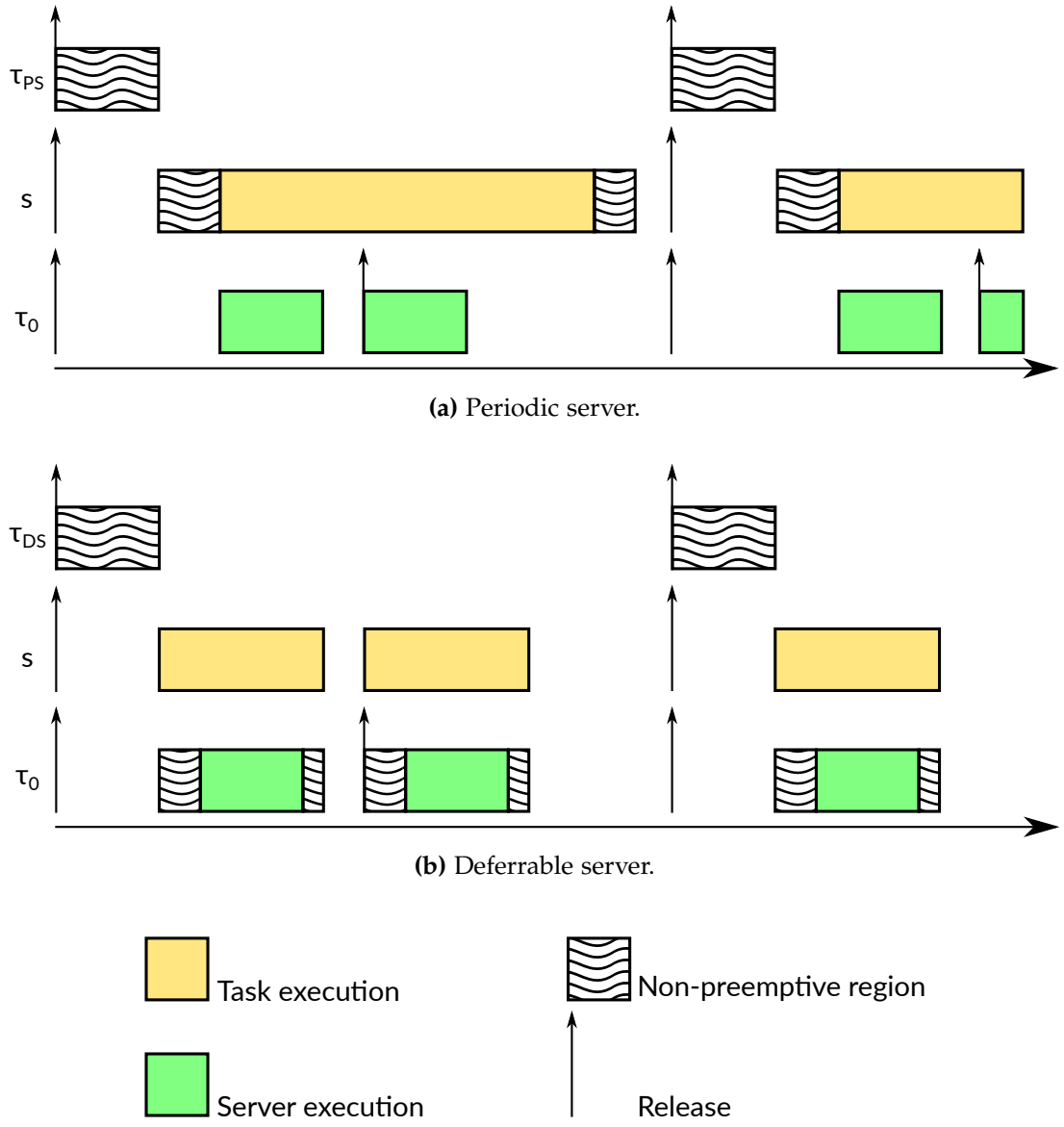
The time-driven tasks in the system are executed using a periodic server in order to alleviate this trade off, therefore improving schedulability, without compromising on the low latency required by event-driven tasks.

3.3.3 Operation of the Execution Servers

The association between servers and tasks is defined using matrix M. The rows of the matrix represent the tasks in the system, whereas the columns are the servers. All elements can take the values 0 or 1. If a task τ_i is serviced by s_j, then $M_{\tau_i,s_j} = 1$, otherwise $M_{\tau_i,s_j} = 0$. Moreover, a task can be serviced by exactly one server, which implies that the sum of each row results in 1.

$$M = \begin{array}{c} \\ \tau_0 \\ \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \\ \tau_6 \\ \tau_7 \end{array} \begin{array}{c} \begin{array}{ccccc} HV & DS_0 & DS_1 & PS_0 & PS_1 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{3.1}$$

Equation (3.1) is an example configuration of an association matrix of a simple sys-



(a) Periodic server.

(b) Deferrable server.

Figure 3.5: Execution servers examples.

tem with two partitions, p_0 and p_1. Each partition is associated with a deferrable and a periodic server. Specifically, DS_0 and PS_0 are associated with partition p_0, and DS_1 and PS_1 are associated with partition p_1. Each of the servers requires a hypervisor task so that its capacity is replenished periodically. The hypervisor tasks that are responsible for replenishing the server's capacity are τ_0, τ_1, τ_2 and τ_3. Partition p_0 has two tasks, τ_5 and τ_6, and p_0 also has two tasks, τ_4 and τ_7. τ_4 and τ_5 are sporadic tasks and are therefore associated with DS_0 and DS_1 respectively. Similarly, τ_6 and τ_7 are periodic tasks and are associated with PS_0 and PS_1 respectively.

The examples of Figure 3.5 demonstrate the execution of periodic and deferrable servers. In both examples we assume a hypervisor task (τ_{PS} or τ_{DS}) that is responsible

for replenishing, an execution server (*PS* or *DS*) and an application-level task, τ_0, that associated with the server. In this example we assume that both servers have the same period and enough capacity to service τ_0.

The hypervisor overheads associated with the periodic server of Figure 3.5a are shown as the non-preemptive regions of *PS*. Periodic servers have generally lower hypervisor overheads, since the hypervisor only needs to switch to the partition's execution context at the start of the server's period. During the server's execution the partition is responsible for scheduling its periodic tasks.

In the case of deferrable servers, the hypervisor is responsible for dispatching the sporadic tasks, preserving the server's capacity when it is not used. The hypervisor overheads for sporadic tasks are shown as the non-preemptive regions of τ_0 in Figure 3.5b. The deferrable server's capacity required to service a sporadic task also includes the hypervisor overheads associated with it.

3.4 Priority Space

Figure 3.6 shows the relationship between the execution modes in terms of their corresponding priority levels. The hypervisor executes in hypervisor mode at the highest system priority region. Since the hypervisor's code is trusted and typically consists of short, highly predictable non-preemptive tasks. The motivation behind this approach is to allow event-triggered tasks to execute at a priority level that is strictly higher than any time-triggered tasks.

Periodic tasks have the lowest priority range in the system. Specifically, a partition executes in periodic mode if no event is pending and eligible to be handled. The eligibility of handling an event is directly associated with the remaining capacity of the server used by the task it triggers. A partition executing in periodic mode is responsible for doing its own task scheduling and may therefore maintain its own internal priority space.

3.5 Resource Management

We propose the use of a modified version of the priority ceiling protocol (PCP) for managing the use of system resources. At the system level, resources are classified as *shared* or *non-shared*. Shared resources can be used by more than one partition and therefore

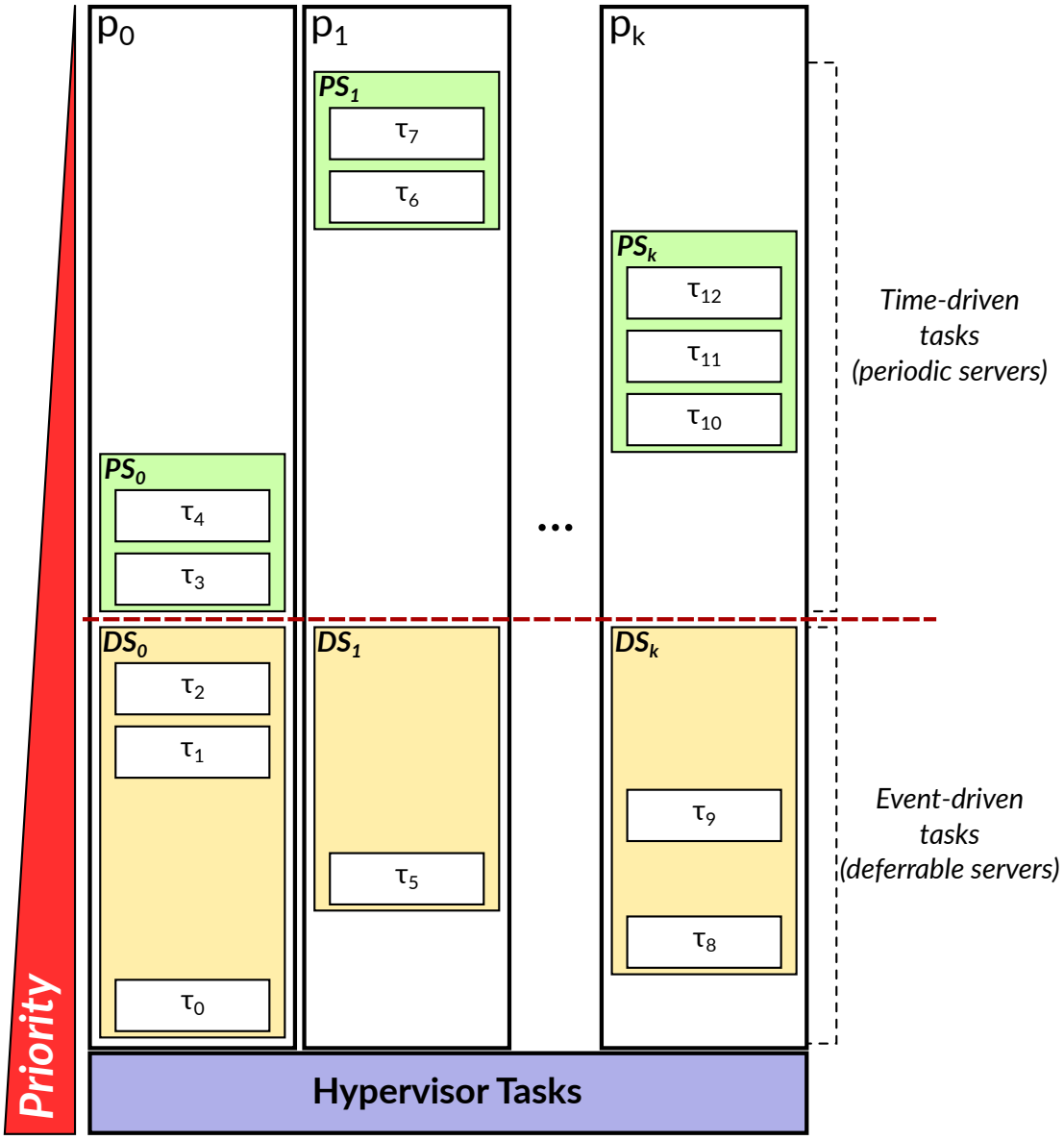
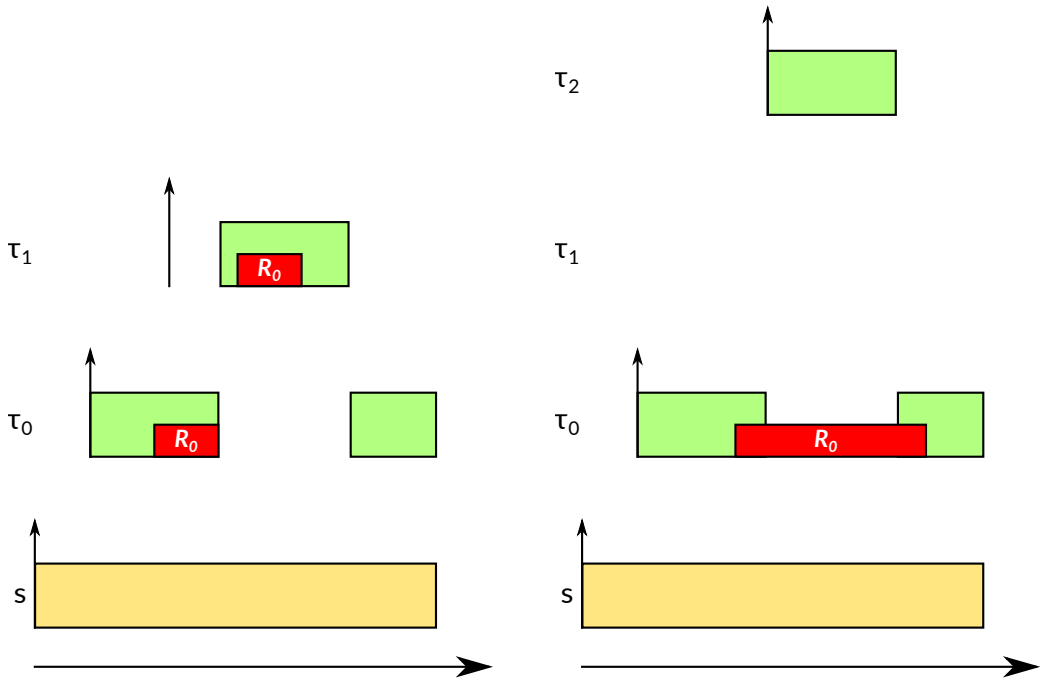


Figure 3.6: Example of a k-partition system priority space.



(a) Blocking of higher priority task with equal ceiling priority.

(b) Preemption by a task with a higher priority than the ceiling.

We assume three tasks, τ_0, τ_1 and τ_2 with increasing priorities. Tasks τ_0 and τ_1 execute using the server s, whereas τ_2 is considered a hypervisor task and can therefore execute without a server. The resource R_0 is shared by tasks τ_0 and τ_1.

Figure 3.7: IPCP preemption examples.

need to be accessed through the hypervisor in order to protect from unbounded blocking.

A non-shared resource is used by a single partition, however it may be shared across multiple tasks within that partition. In the non-shared resources case, hypervisor support is only required if the resource is used in the sporadic mode. The use of hypervisor support for the use of resources is required due to the overlap in priorities between partitions.

Similarly with the immediate priority ceiling protocol (IPCP) [94], each task in the system is assigned a static priority as described in Section 3.4. For every hypervisor managed resource a ceiling priority is assigned, which is defined as the maximum priority of the tasks that use it. The dynamic priority of a task is therefore evaluated as the ceiling of the task's static priority and highest priority of all the resources it uses.

The use of IPCP guarantees that a higher priority task can only be blocked by a lower priority task at most once. By definition of the protocol, mutual exclusion is en-

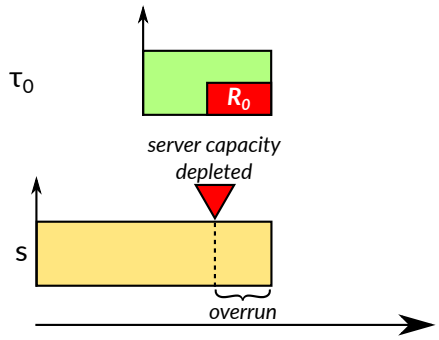


Figure 3.8: Server capacity overrun example.

forced if tasks cannot self suspend. Additionally, deadlocks and transitive blocking are prevented. Although these are necessary properties, they are not sufficient for fault isolation with respect to the level of interference between partitions. We therefore propose two constraints.

The first constraint is to have an upper limit on the amount of time a task can spend accessing a hypervisor managed resource. Having a limit on the time a task can access is a mechanism for controlling the amount of time a high priority task (i.e. partition ISR) can be blocked by a lower priority one (e.g. partition tasks executing in periodic mode). This constraint also serves as a fault detection and isolation mechanism.

Figure 3.7 demonstrates the use of IPCP in the proposed system model. In the example of Figure 3.7a we show the case where a higher priority task, τ_1, can be blocked by a lower priority one, τ_0. Specifically, τ_0 is released first and starts executing at its own priority level. During its execution it starts using resource R_0, which raises its priority level to that of τ_1, which is the ceiling priority. τ_1 is released while τ_0 holds the resource and is therefore blocked until the resource is released. Releasing R_0 drops τ_0's priority to its default level, therefore allowing τ_1 to preempt τ_0 and execute to completion. τ_0 then resumes its execution to completion since it is the highest priority task in the system that is ready to execute.

In the scenario exampled in Figure 3.7b, τ_0 starts executing and starts using the resource R_0, raising the priority to the ceiling, which is the priority of τ_1. The hypervisor task τ_2 is released and preempts τ_0, since τ_2's priority is greater than the ceiling priority of τ_0. The resource can remain locked by τ_0 even while it is preempted, since IPCP is deadlock-free. τ_2 executes to completion, allowing τ_0 to continue executing. The resource R_0 is released by τ_0, lowering its dynamic priority and continues to execute until it terminates.

The second constraint that is proposed is that a partition executing in periodic mode must have released all hypervisor managed resources before it is preempted due to its server's capacity being depleted. In order to enforce this constraint while maintaining temporal isolation, we allow a partition to overrun in its periodic mode by the maximum length of the time interval a task can use any resource [31]. This is shown in Figure 3.8. The overrun is then subtracted from the partition's periodic server capacity at the next replenishment. This ensures that when the system executes for a long amount of time, no periodic server exceeds its average resource utilisation by continuously requesting to use a shared resource when it has minimal capacity.

3.6 Modifications to Partitions

The proposed architecture assumes a paravirtualised approach, similar to the one supported by RTA-HV, as described in Section 2.6.1. Porting a partition to allow it to execute on the proposed hypervisor system first requires changes to its start-up code. Specifically, part of the partition initialisation code is no longer required, since it is performed by the hypervisor. This includes setting up the stack, interrupt tables and peripherals that are not accessible from a non-privileged mode.

Calls to privileged instructions that cannot be efficiently emulated are also replaced in the partition's code by the appropriate hypervisor calls. Examples of such instructions are changing the core's current priority level, reading/writing to IO devices and disabling/masking interrupts.

The above changes are sufficient for a partition to execute using a hypervisor, regulating its CPU quota using just periodic servers. The use of periodic servers does not require any visibility of the internal task structure of the partition. Taking advantage of deferrable servers however, requires that the hypervisor has knowledge of when partition tasks are released and terminate.

Typically, all task releases are either in response to a peripheral, such as CAN, or a scheduled timer interrupt. As in RTA-HV, all interrupts are first handled by the hypervisor and if appropriate are forwarded to their corresponding partition, according to the configuration of the system. The hypervisor is therefore able to keep track of the release times of partition tasks. The partition is responsible for informing the hypervisor the termination of its sporadic tasks using a hypervisor call.

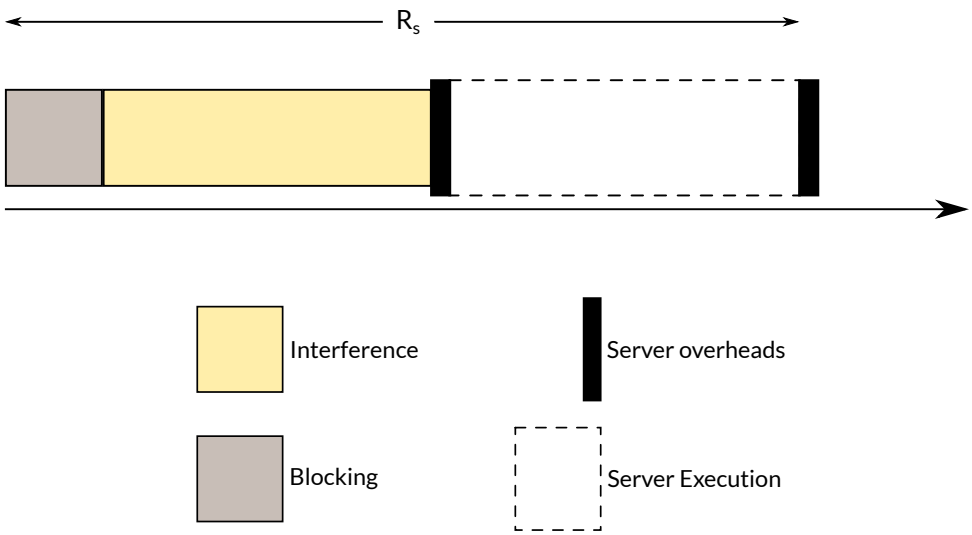


Figure 3.9: Server critical instance.

3.7 Response Time Analysis

The response time analysis is performed in two stages. The first stage is a response time analysis at the server level, to determine whether the servers are guaranteed to receive their required execution time during their periods. If the servers are guaranteed their required capacity, the response time analysis is performed on the tasks in order to determine whether they can meet their deadlines within their servers. Table 3.1 summarises the symbols used in the response time analysis of this section.

3.7.1 Server Schedulability

In this section we present the schedulability analysis of a set of servers. Specifically, we consider a server to be schedulable if between any of its replenishments it is guaranteed to receive as much CPU time as its capacity. For this analysis, we treat servers as ordinary tasks with an execution time equal to their capacity, C_s, and a period equal to their replenishment period T_s. The deadline of the server tasks is set to be equal to their replenishment period ($D_s = T_s$).

As shown in Figure 3.9, the response time of a server, R_s, is defined as the maximum amount of time required to use its capacity after its release. Specifically, it is the time required for a server to execute for $C_s^{cs1} + C_s$ time units. Note that the second context switch overhead, C_s^{cs2}, is not part of the server's response time.

A server can be blocked at most once if a non-preemptive region is being executed.

The non-preemptive region in the system are the forwarding and return overheads (C^{cs1} and C^{cs2}). Since hypervisor tasks are of higher priority than all servers, they are not calculated as blocking but as interference. Therefore, the blocking of a task is considered as the maximum execution time of all low priority non-preemptive tasks:

$$B_s = \max\left\{\max(C_j^{cs1}, C_j^{cs2}) | j \in \text{lp}(s) \wedge j \in async\right\} \cup \left\{\max(C_j^{cs1}, C_j^{cs2}) | j \in \text{lp}(s) \wedge j \in PS\right\} \tag{3.2}$$

The response time of a server can also be affected by push-through blocking as its non-preemptive region may be executed after the deadline of the server. This is an acceptable scenario, as the response time refers to the time required for the server to receive its total capacity, C_s. The context switch overhead that takes place after the server depletes its capacity, C_s^{cs2} can therefore be pushed through to the next period. The blocking factor for a server therefore becomes:

$$B_s^{MAX} = \max(C_s^{cs2}, B_s) \tag{3.3}$$

Next, we identify the sources of interference. For any server there are two sources of interference: hypervisor tasks and servers with a higher ceiling priority (deferrable and periodic).

The interference from hypervisor tasks and higher priority servers over a window w is defined as:

$$I_s(w) = \sum_{j \in \text{hp}(s)} \left\lceil \frac{w + J_j}{T_j} \right\rceil \left(C_j^{cs1} + C_j + C_j^{cs2}\right) \tag{3.4}$$

The response time of a server, R_s is given by the recurrence relation:

$$R_s = C_s^{cs1} + C_s + B_s^{MAX} + \sum_{j \in \text{hp}(s)} \left\lceil \frac{R_s + J_j}{T_j} \right\rceil \left(C_j^{cs1} + C_j + C_j^{cs2}\right) \tag{3.5}$$

3.7.2 Task Schedulability

To calculate the response time of a task τ_i we first need to define the server load. Given a task τ_i, which uses the server s, the load on the server at priority level i is given by

Symbol	*Description*
C_i^{cs1}	Time required by the overheads before the execution of the main body of a task or server.
C_i	Execution time required by the main body of a task or the capacity of a server
C_i^{cs2}	Time required by the overheads after the execution of the main body of a task or server.
T_i	The period of a task or server.
P_i	The priority of a task or server.
J_i	Release jitter.
B_i	Blocking received by tasks of priority lower than P_i.
$M_{i,s}$	Returns 1 if the task τ_i executes using the server s.
R_i	The response time of a task or server.
$L_i^s(w)$	The load on a server s at the priority level P_i over the length of a window w.
$I_i(w)$	The interference received by a task or server at priority level P_i
$\mathrm{lp}(i)$	The set of tasks or servers of lower priority than P_i.
$\mathrm{hp}(i)$	The set of tasks or servers of higher priority than P_i.
async	The set of sporadic tasks.
sync	The set of periodic tasks.
hv	The set of hypervisor tasks.
PS	The set of periodic servers.
DS	The set of deferrable servers.

Table 3.1: Table of symbols.

Equation (3.6). Therefore, the load of a server s in a window w at a priority level P_i and higher is defined as the sum of C_i and the total interference from higher priority tasks that are serviced by s.

$$L_i^s(w) = C_i^{cs1} + C_i + C_i^{cs2} + \sum_{j \in \mathrm{hp}(i)} M_{j,s} \left\lceil \frac{w + J_j}{T_j} \right\rceil \left(C_j^{cs1} + C_j + C_j^{cs2} \right) \tag{3.6}$$

The total length of gaps where there is no available server capacity is defined as:

$$\left(\left\lceil \frac{L_i^s(w)}{C_s} \right\rceil - 1 \right) (T_s - C_s) \tag{3.7}$$

A task τ_i that uses a deferrable server can be blocked at most once if the system is executing non-preemptive code regions. Such sections are the overheads of forwarding events to partitions, returning from them and the context switches associated with periodic servers. The blocking resulting from non-preemptive code regions of lower priority tasks is given by B_i^{cs}.

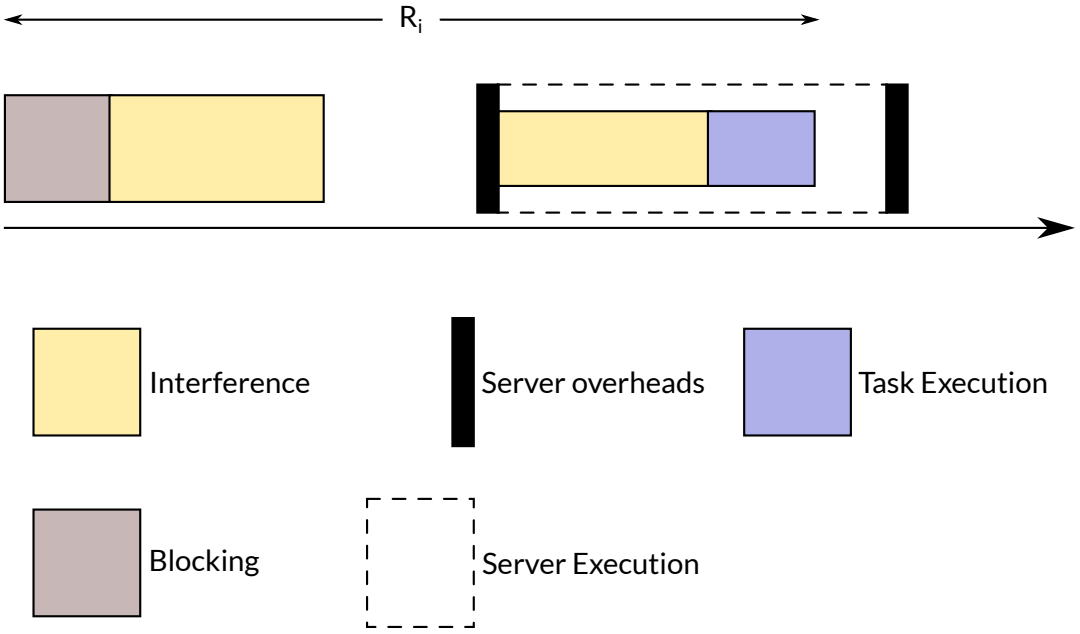


Figure 3.10: Periodic task critical instance.

$$B_i^{cs} = \max\left\{\max(C_j^{cs1}, C_j^{cs2}) | j \in \text{lp}(i) \wedge j \in async\right\} \cup \left\{\max(C_j^{cs1}, C_j^{cs2}) | j \in \text{lp}(i) \wedge j \in PS\right\} \tag{3.8}$$

As identified by Davis et al [32] and later by Whitham et al [105], an additional source of blocking for task τ_i is its own context switch overhead after the execution of its main body, as this can be propagated into the next busy period. Therefore the blocking factor for τ_i is the maximum of C_i^{cs2} and the blocking from lower priority context switching overheads B_i^{cs}.

$$B_i = \max(C_i^{cs2}, B_i^{cs}) \tag{3.9}$$

Next we will define the interference of periodic and sporadic tasks. First we consider sporadic tasks. Aperiodic tasks receive interference from higher priority sporadic tasks and hypervisor tasks. Therefore, the interference at priority level P_i received by task τ_i over a window w is defined by:

$$I_i(w) = \sum_{j \in hp(i)} \left\lceil \frac{w + J_j}{T_j} \right\rceil \left(C_j^{cs1} + C_j + C_j^{cs2}\right) \tag{3.10}$$

A periodic task receives interference from higher priority periodic servers, higher priority tasks that use the same server as itself, sporadic tasks and hypervisor tasks. Additionally, a periodic task's response time is also affected by its server's context switch overhead, C_s^{cs1}. This behaviour is shown in Figure 3.10. In the scenario of Figure 3.10 on arrival all higher priority tasks in the server execute, using up its capacity. On the next server period the task receives interference from sporadic and hypervisor tasks. When the server is able execute it first elapses C_i^{cs1} time units before it becomes able to start servicing tasks. The task then starts executing and is preempted by other tasks within the same server and subsequently by tasks outside the server.

The interference of a periodic task τ_i with priority level P_i, executing on server s ($M_{\tau_i,s} = 1$) is given by:

$$I_i(w) = \sum_{j \in hv} \left\lceil \frac{w}{T_j} \right\rceil \left(C_j^{cs1} + C_j + C_j^{cs2} \right) + \tag{3.11}$$

$$\sum_{k \in async} \left\lceil \frac{w + J_k}{T_k} \right\rceil \left(C_k^{cs1} + C_k + C_k^{cs2} \right) + \tag{3.12}$$

$$\sum_{l \in \mathrm{hp}(i)} M_{l,s} \left\lceil \frac{w + J_l}{T_l} \right\rceil \left(C_l^{cs1} + C_l + C_l^{cs2} \right) \tag{3.13}$$

The response time of partition/application tasks (periodic or sporadic) is given by the following recurring relation:

$$R_i = C_i^{cs1} + C_i + C_i^{cs1} + \left(\left\lceil \frac{L_i^s(R_i)}{C_s} \right\rceil - 1 \right) (T_s - C_s) + B_i + I_i(R_i) \tag{3.14}$$

Hypervisor tasks are not associated with execution servers, therefore their release jitter is 0. In their work, Davis and Burns [30] show that the release jitter of tasks is dependent on their period. Specifically, in the case where a task's period is a multiple of its server's period it has a release jitter of 0, otherwise $R_s - T_s$.

3.8 Worked Example

Consider a simple system configuration with two partitions, p_0 and p_1 with the taskset shown in Table 3.2. The taskset is consisted of two sporadic tasks, τ_3 and τ_4, and two periodic, τ_5 and τ_6. Tasks τ_0, τ_1 and τ_2 are hypervisor tasks responsible for replenishing the server capacity. Equation (3.15) shows the task-server association matrix. The tasks associated with partition p_0 are serviced by the servers DS_0 and PS_0. Partition p_1 is only

Task	C^{cs1}	C	C^{cs2}	T	P	Type	Partition
τ_0	0	3	0	100	1	HV	-
τ_1	0	3	0	200	2	HV	-
τ_2	0	3	0	300	3	HV	-
τ_3	2	10	1	100	4	async	p_0
τ_4	2	10	1	200	5	async	p_1
τ_5	1	30	1	300	6	sync	p_0
τ_6	1	30	1	400	7	sync	p_0

Table 3.2: Worked example tasks.

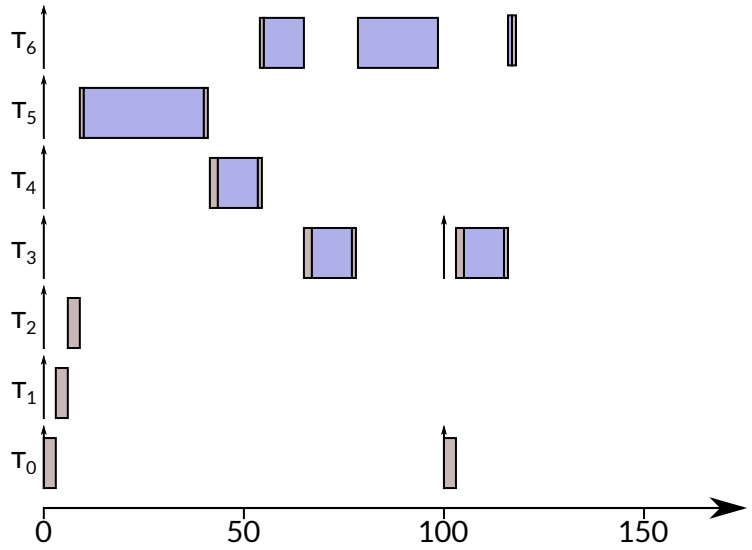


Figure 3.11: Timeline of a scenario in the worked example.

associated with an sporadic task, τ_4, which is serviced by the deferrable server DS_1.

$$M = \begin{array}{c} \\ \tau_0 \\ \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \\ \tau_6 \end{array} \begin{array}{c} \begin{array}{cccc} HV & DS_0 & DS_1 & PS_0 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{3.15}$$

Figure 3.11 shows a timeline of a potential scenario in the system described in this example. All tasks are released at the start of the timeline. Hypervisor tasks (τ_0, τ_1 and τ_2) execute first from highest to lowest priority. The execution of tasks using deferrable servers are delayed due to jitter, therefore the periodic server PS_0 is the highest priority

entity that is ready to execute. During the execution of PS_0, τ_5 is serviced and executes to completion. After the completion of τ_5, τ_4 becomes ready, preempting PS_0. The periodic server PS_0 resumes its execution after τ_4 returns. The highest priority task that is ready to execute at this point is τ_6, therefore it starts executing using ts server's capacity C_{DS_0}. During the execution of τ_6, τ_3 becomes ready, therefore PS_0 is preempted, allowing τ_3 to execute to completion. At $t = 100$, the hyperisor task τ_0 is released, therefore preempting the periodic server, suspending the execution of τ_6. At the same time, τ_3 is released and ready to execute. The periodic server PS_0 is preempted for 16 time units, as τ_0 and τ_3 execute back-to-back. After τ_3 returns, τ_6 executes for the remainder of its execution time.

3.8.1 Server Parameters

The period of each server is set as the minimum of the periods of the tasks serviced by it. Therefore, the periods of the servers DS_0, DS_1 and PS_0 are 100, 200 and 300 respectively. Given the server periods, the minimum server capacity required by the serviced tasks is given by:

$$C_s = \sum_{\forall i} M_{i,s} \left\lceil \frac{T_i}{T_s} \right\rceil (C_i^{cs1} + C_i + C^{cs2}) \tag{3.16}$$

Solving for each server:

$$C_{DS_0} = \left\lceil \frac{100}{100} \right\rceil (2 + 10 + 1) = 13 \tag{3.17}$$

$$C_{DS_1} = \left\lceil \frac{200}{200} \right\rceil (2 + 10 + 1) = 13 \tag{3.18}$$

$$C_{PS_0} = \left\lceil \frac{300}{300} \right\rceil (1 + 30 + 1) + \left\lceil \frac{400}{300} \right\rceil (1 + 30 + 1) = 96 \tag{3.19}$$

The server overheads of PS_0 are arbitrarily set as $C_{cs1} = 2$ and $C_{cs2} = 1$ for demonstration purposes. Moreover, the server periods are used as the replenishment periods for the hypervisor tasks τ_0, τ_1 and τ_2. Having the complete set of server and task parameters, we then proceed to determining whether the system is schedulable.

3.8.2 Server Response Time Analysis

Having all the required parameters, the first part of the analysis is the calculation of the server response times. Starting with $w_s^0 = C_s^{cs1} + C_s$, the server response times is given

by the recurrence relation:

$$w_s^{n+1} = C_s^{cs1} + C_s + B_s + \sum_{j \in \text{hp}(s)} \left\lceil \frac{w_s^n + J_j}{T_j} \right\rceil \left(C_j^{cs1} + C_j + C_j^{cs2} \right) \tag{3.20}$$

The blocking factor of the execution servers was defined as the maximum amount of time spent in non-preemptive regions (C^{cs1} or C^{cs1}) by lower priority sporadic tasks or periodic servers. This results in $B = 2$ for DS_0 and DS_1. PS_0 is the lowest priority server in the system and therefore does not suffer from blocking. Solving Equation (3.20) for DS_0:

$$\begin{aligned}
w_{DS_0}^0 &= 13 \\
w_{DS_0}^1 &= 13 + 2 + \left\lceil \frac{13}{300} \right\rceil 3 + \left\lceil \frac{13}{200} \right\rceil 3 + \left\lceil \frac{13}{100} \right\rceil 3 = 24 \\
w_{DS_0}^2 &= 13 + 2 + \left\lceil \frac{24}{300} \right\rceil 3 + \left\lceil \frac{24}{200} \right\rceil 3 + \left\lceil \frac{24}{100} \right\rceil 3 = 24
\end{aligned}$$

$w_{DS_0}^1 = w_{DS_0}^2$, therefore the response time of DS_0 is $R_{DS_0} = 24$. We then proceed to solve for DS_1.

$$\begin{aligned}
w_{DS_1}^0 &= 13 \\
w_{DS_1}^1 &= 13 + 2 + \left\lceil \frac{15}{300} \right\rceil 3 + \left\lceil \frac{13}{200} \right\rceil 3 + \left\lceil \frac{13}{100} \right\rceil 3 + \left\lceil \frac{13}{100} \right\rceil 13 = 37 \\
w_{DS_1}^2 &= 13 + 2 + \left\lceil \frac{37}{300} \right\rceil 3 + \left\lceil \frac{37}{200} \right\rceil 3 + \left\lceil \frac{37}{100} \right\rceil 3 + \left\lceil \frac{37}{100} \right\rceil 13 = 37
\end{aligned}$$

$w_{DS_1}^1 = w_{DS_1}^2$, therefore the response time of DS_1 is $R_{DS_1} = 37$.Similarly, for the periodic server PS_0:

$$\begin{aligned}
w_{PS_0}^0 &= 98 \\
w_{PS_0}^1 &= 98 + \left\lceil \frac{98}{300} \right\rceil 3 + \left\lceil \frac{98}{200} \right\rceil 3 + \left\lceil \frac{98}{100} \right\rceil 3 + \left\lceil \frac{98}{100} \right\rceil 13 + \left\lceil \frac{98}{200} \right\rceil 13 = 133 \\
w_{PS_0}^2 &= 98 + \left\lceil \frac{133}{300} \right\rceil 3 + \left\lceil \frac{133}{200} \right\rceil 3 + \left\lceil \frac{133}{100} \right\rceil 3 + \left\lceil \frac{133}{100} \right\rceil 13 + \left\lceil \frac{133}{200} \right\rceil 13 = 149 \\
w_{PS_0}^3 &= 96 + \left\lceil \frac{149}{300} \right\rceil 3 + \left\lceil \frac{149}{200} \right\rceil 3 + \left\lceil \frac{149}{100} \right\rceil 3 + \left\lceil \frac{149}{100} \right\rceil 13 + \left\lceil \frac{149}{200} \right\rceil 13 = 149
\end{aligned}$$

The response time of PS_0 is: $R_{PS_0} = 149$. The response times of all servers are less than their periods, therefore they are guaranteed to receive the required CPU time to meet

their capacity.

3.8.3 Task Response Time Analysis

The last part of the analysis is to calculate the response time of all partition tasks in the system to make guarantees on their worst case response time. The response time analysis of partition tasks is calculated using the recurrence relation:

$$w_i^{n+1} = C_i^{cs1} + C_i + B_i + \left(\left\lceil \frac{L_i^s(w_i^n)}{C_s} \right\rceil - 1 \right) (T_s - C_s) + I(w_i^n) \tag{3.21}$$

where s is the server associated with task τ_i, such that $M_{i,s} = 1$.

Aperiodic tasks

The blocking received by sporadic tasks is calculated as the maximum non-preemptive region of lower priority sporadic tasks and periodic servers. Therefore, the blocking factor of all sporadic tasks is 2.

Using the simple taskset provided in this example allowed the selection of server parameters that guarantee that there is always enough capacity to service all tasks. At any priority level the load of the server is less than or equal to the server's maximum capacity, $L(w)_i^s \leq C_s$, therefore there are no gaps where servers have no capacity.

$$\left(\left\lceil \frac{L_i^s(w_i^n)}{C_s} \right\rceil - 1 \right) (T_s - C_s) = 0$$

The response time of τ_3 is calculated solving the recurrence relation of Equation (3.21):

$$\begin{aligned}
w_3^0 =& 12 \\
w_3^1 =& 12 + 2 + \left\lceil \frac{12}{300} \right\rceil 3 + \left\lceil \frac{12}{200} \right\rceil 3 + \left\lceil \frac{12}{100} \right\rceil 3 = 23 \\
w_3^2 =& 12 + 2 + \left\lceil \frac{23}{300} \right\rceil 3 + \left\lceil \frac{23}{200} \right\rceil 3 + \left\lceil \frac{23}{100} \right\rceil 3 = 23
\end{aligned}$$

$w_3^2 = w_3^1 = 23$, therefore the response time of τ_3 is $R_3 = 23$. The next sporadic task we consider is τ_4. The task τ_4 receives interference from all hypervisor tasks and τ_3. Interference received by sporadic tasks are subject to release jitter, $J_i = T_i - R_s$, since

they utilise a deferrable server. Therefore the response time of τ_4 is calculated as:

$$
\begin{aligned}
w_4^0 =& 12 \\
w_4^1 =& 12 + 2 + \left\lceil \frac{12}{300} \right\rceil 3 + \left\lceil \frac{12}{200} \right\rceil 3 + \left\lceil \frac{12}{100} \right\rceil 3 + \left\lceil \frac{12 + (100 - 37)}{100} \right\rceil 13 = 49 \\
w_4^2 =& 12 + 2 + \left\lceil \frac{49}{300} \right\rceil 3 + \left\lceil \frac{49}{200} \right\rceil 3 + \left\lceil \frac{49}{100} \right\rceil 3 + \left\lceil \frac{49 + (100 - 37)}{100} \right\rceil 13 = 49
\end{aligned}
$$

The response time of τ_4 was calculated as $R_4 = 49$. The response times of both τ_3 and τ_4 are less than their deadlines, which are assumed to be their periods, therefore both sporadic tasks in the system will meet their deadlines.

Periodic Tasks

Next we calculate the response times of the periodic tasks in the system. Similarly with the deferrable servers, the parameters of the periodic servers were selected to avoid gaps where there is no available server capacity. Therefore,

$$
\left(\left\lceil \frac{L_i^s(w_i^n)}{C_s} \right\rceil - 1 \right) (T_s - C_s) = 0
$$

Periodic tasks are blocked by lower priority tasks within the same server, or lower priority periodic servers. In the case of τ_5, the blocking factor is 1, since it can be blocked by τ_6 for up to $C_6^{cs1} = 1$ time units. Periodic tasks receive interference from hypervisor and sporadic tasks, higher priority periodic servers and higher priority periodic tasks that use the same servers. The response time of τ_5 was calculated using the recurrence relation:

$$
\begin{aligned}
w_5^0 =& 31 \\
w_5^1 =& 31 + 1 + \left\lceil \frac{31}{300} \right\rceil 3 + \left\lceil \frac{31}{200} \right\rceil 3 + \left\lceil \frac{31}{100} \right\rceil 3 + \\
& \left\lceil \frac{31 + (200 - 24)}{200} \right\rceil 13 + \left\lceil \frac{31 + (100 - 37)}{100} \right\rceil 13 = 92 \\
w_5^2 =& 31 + 1 + \left\lceil \frac{92}{300} \right\rceil 3 + \left\lceil \frac{92}{200} \right\rceil 3 + \left\lceil \frac{92}{100} \right\rceil 3 + \\
& \left\lceil \frac{92 + (200 - 24)}{200} \right\rceil 13 + \left\lceil \frac{92 + (100 - 37)}{100} \right\rceil 13 = 92
\end{aligned}
$$

$R_5 = 92$, since $w_5^2 = w_5^1 = 92$. The next task we consider is τ_6. τ_6 is the lowest priority task in the system and therefore receives no blocking. The response time is calculated

Name	*B*	*I*	D	*R*
Tasks				
τ_0	2	-	100	5
τ_1	2	τ_0	200	8
τ_2	2	τ_0, τ_1	300	11
τ_3	2	τ_0, τ_1, τ_2	100	23
τ_4	2	τ_0, τ_1, τ_2, τ_3	200	49
τ_5	1	τ_0, τ_1, τ_2, τ_3, τ_4	300	92
τ_6	0	τ_0, τ_1, τ_2, τ_3, τ_4, τ_5	400	127
Servers				
DS_0	2	τ_0, τ_1, τ_2	100	24
DS_1	2	τ_0, τ_1, τ_2, τ_3	200	37
PS_0	0	τ_0, τ_1, τ_2, τ_3, τ_4	300	149

Table 3.3: Summary of the response time analysis.

as follows:

$$
\begin{aligned}
w_6^0 =& 31 \\
w_6^1 =& 31 + \left\lceil \frac{31}{300} \right\rceil 3 + \left\lceil \frac{31}{200} \right\rceil 3 + \left\lceil \frac{31}{100} \right\rceil 3 + \\
& \left\lceil \frac{31 + (200 - 24)}{200} \right\rceil 13 + \left\lceil \frac{31 + (100 - 37)}{100} \right\rceil 13 + \left\lceil \frac{31}{300} \right\rceil 32 = 124 \\
w_6^2 =& 31 + \left\lceil \frac{124}{300} \right\rceil 3 + \left\lceil \frac{124}{200} \right\rceil 3 + \left\lceil \frac{124}{100} \right\rceil 3 + \\
& \left\lceil \frac{124 + (200 - 24)}{200} \right\rceil 13 + \left\lceil \frac{124 + (100 - 37)}{100} \right\rceil 13 + \left\lceil \frac{124}{300} \right\rceil 32 = 127 \\
w_6^3 =& 31 + \left\lceil \frac{127}{300} \right\rceil 3 + \left\lceil \frac{127}{200} \right\rceil 3 + \left\lceil \frac{127}{100} \right\rceil 3 + \\
& \left\lceil \frac{127 + (200 - 24)}{200} \right\rceil 13 + \left\lceil \frac{127 + (100 - 37)}{100} \right\rceil 13 + \left\lceil \frac{127}{300} \right\rceil 32 = 127
\end{aligned}
$$

$R_6 = 127$, since $w_6^3 = w_6^2$. Both response times for τ_5 and τ_6 are less than their periods, therefore they are guaranteed to meet their deadlines.

System Schedulability

According to the response time analysis performed in this section, the example system was deemed as schedulable. Table 3.3 summarises the results of the schedulability analysis. Specifically, for each task and server in the system the calculated blocking (B), interference (I) and response times (R) are listed. Comparing all response times with the deadline (D) of each task, all response times are shorter than the deadlines, therefore all tasks are schedulable.

3.9 Summary

In this chapter we present the proposed system architecture for a hypervisor system that supports mapping many partitions onto a single core. We present the proposed memory configuration, to enforce spatial protection and support the operations required by the scheduling model. We then focus on detailing the scheduling model, which is motivated by the modularity of AUTOSAR-based automotive OSs.

Our approach uses a priority ordering, which is aimed to reduce the latency of handling event-triggered events, while having a high level of processor utilisation. Temporal protection is enforced using execution servers, which guarantee partitions with a certain amount of the CPU time over a fixed period. Two types of servers are used to minimise the latency of handling event triggered tasks, allow for high utilisation and enforce temporal protection. Specifically, event-triggered (sporadic) tasks execute within high priority deferrable servers, whereas time-triggered (periodic) tasks execute using lower priority periodic servers.

In the next chapter we evaluate our architecture using a case study that was obtained through our collaboration with ETAS Ltd. A partial hypervisor implementation and a realistic application were used to obtain the necessary time measurements to realise the proposed model. The realised model is then used to determine the system's maximum utilisation, given a priority ordering, both using the response time analysis of this chapter and via simulation.

CHAPTER 4

Case Study: Engine Controller

The hypothesis of this research project, as stated in Chapter 1, is that virtualisation can be used in the automotive industry for the integration of multiple control units on a single processor while retaining the real-time properties of the system. The use of HV technology as a virtualisation technique also offers the benefit for alleviating some of the challenges identified in Section 1.1. In Chapter 2 we reviewed relevant work in the literature as well as existing HVs. We then reviewed RTA-HV, the HV developed by ETAS Ltd, motivating the development of the system model described in Chapter 3.

In this chapter we evaluate the proposed scheduling model in terms of utilisation and the tightness of the schedulability analysis using a case study that was derived from application code provided by ETAS Ltd.

First, we present an overview of the hardware platform (ARM1176-JZF-S) that was used to obtain time measurements of the application tasks. The application tasks, which were provided by the industrial sponsor, were ported to execute on the aforementioned hardware platform and a measurement-based timing analysis was performed. A partial hypervisor was implemented in order to obtain timing information for the associated overheads. The timing information was used to generate scenarios for a custom simulation. The results from the simulation runs were analysed and discussed in terms of the architectural decisions of Chapter 3.

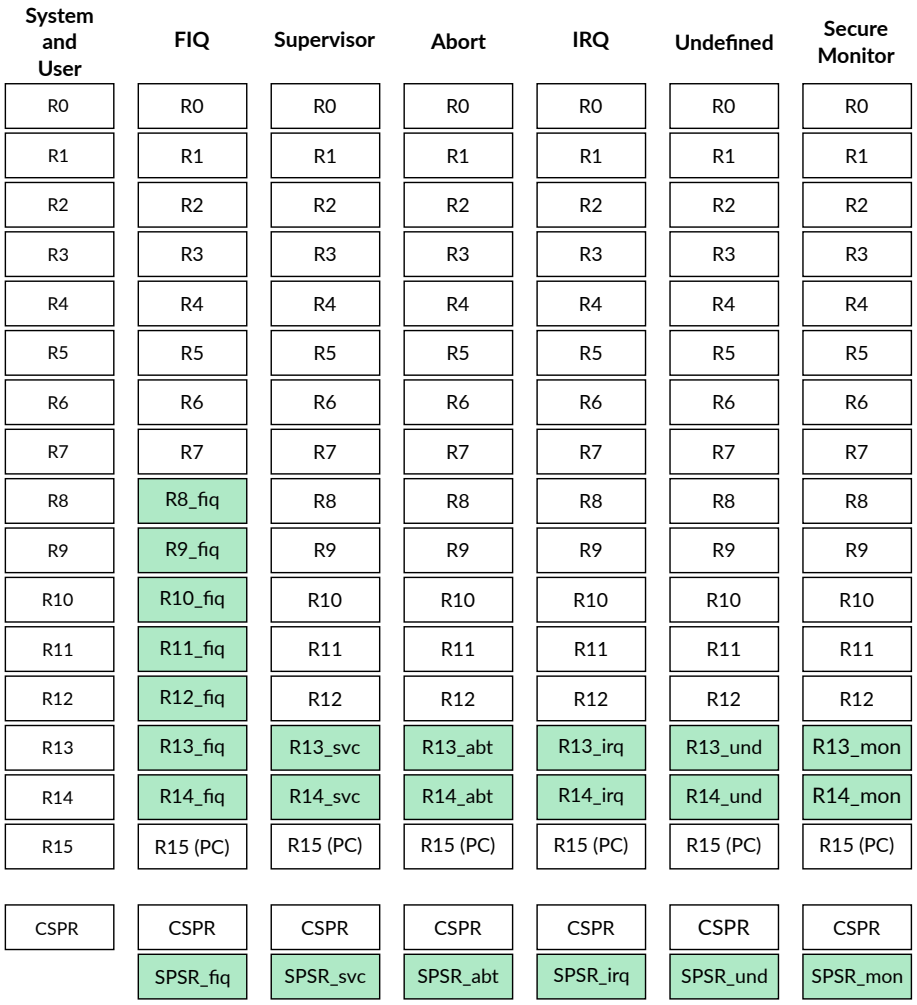


Figure 4.1: ARM1176JZF-S core registers.

4.1 Hardware Platform Characteristics

The experimental evaluation for the proposed system model was performed using a custom simulator, which is detailed in Section 4.3, using measurements that were obtained using an ARM1176JZF-S processor [6]. ARM1176JZF-S is a 32-bit processor based on the ARMv6 architecture using a system control coprocessor, CP15, which is used to read status information and to configure the processor's functions such as interrupts, the memory management unit (MMU) and performance monitoring.

4.1.1 Operating Modes and Core Registers

The ARM1176JZF-S processor has eight modes of execution [6]:

- **User:** the mode of execution that is conventionally used by user applications. In

the context of the proposed system, the partitions will execute in User mode.

- **System:** a privileged form of User mode, typically used by the OS. This mode can be used by the hypervisor to access the partitions' execution context.
- **FIQ:** is used when handling fast interrupts.
- **Supervisor:** is a protected privileged mode.
- **Abort:** is entered in the event of a prefetch or data abort.
- **IRQ:** is used when handling interrupts.
- **Undefined:** is entered when an instruction results in an undefined exception.
- **Secure Monitor:** is a mode that is part of TrustZone, which is a set of security extensions that were first introduced in ARM1176JZF-S.

Figure 4.1 lists the core registers of ARM1176JZF-S that are accessible by each mode. Each mode has 16 general purpose registers (`R0 - R15`) and a status register (`CSPR`). `SPSR` is a program status register that contains the processor's status before a mode change that was caused by an exception. FIQ has 8 banked registers, whereas the rest of the privileged modes, excluding System mode, have 3. The banked registers allow for faster exception handling since in some cases saving the banked register values can be avoided.

4.1.2 Memory Management

Memory protection in ARM1176JZF-S is implemented using an MMU based on the ARMv6 architecture [6]. The ARMv6 MMU supports virtual to physical address translation, therefore simplifying the linking process. The support of address translation allows partitions to be linked according to a virtual address space, instead modifying the linker script for each partition separately to avoid clashes. The hypervisor needs only to be aware of the physical memory address of the partitions' entry points. This also makes the rearranging of the memory layout of partitions simpler, since all the required modifications are part of the hypervisor's configuration.

Another feature of the MMU is the support of up to 16 memory domains, therefore allowing up to 15 partitions per processor. In the ARMv6 architecture a domain is a collection of memory regions. Each memory region has its own access rights, therefore allowing a partition to potentially create a custom configuration of the memory

regions within its domain. This has the advantage of potentially reducing the migration overheads since changes in the partitions' memory setup can be avoided.

4.1.3 Vectored Interrupts

ARM1176JZF-S supports the use of vectored interrupts with the addition of the ARM PrimeCell Vectored Interrupt Controller (PL192) [5]. PL192 supports up to 32 vectored interrupt lines.

Each interrupt line can be enabled/disabled, have its handler's address and priority specified programmatically. Enabling/disabling certain interrupt lines is a requirement of the proposed model, allowing masking the release of higher priority tasks when their server's capacity is depleted. Vectored interrupts can also be taken advantage for lower latency, which is a key driver in the development of the proposed model.

The rationale behind the approach followed for forwarding and returning from interrupts is to minimise the required changes to the partition code. Therefore, the environment provided to the partition by the hypervisor after an interrupt is forwarded must reflect the hardware's behaviour.

On arrival of an interrupt the processor saves its state on the `SPSR`, the next instruction is preserved in the link register, `R14`. The processor then enters its relevant interrupt state (`FIQ` or `IRQ`, depending on the type of interrupt), and the address of the appropriate exception vector is set as the program counter, `R15`. In the case of a partition interrupt, the hypervisor first sets a watchdog timer to ensure that the partition does not exceed its server's capacity.

The execution context and state information required to support preemption is saved. Acknowledgement of the interrupt is signaled to PL192, interrupts of higher priorities are disabled. As identified in Section 3.2, the time required to forward a partition interrupt is C^{cs1}. Finally, the partition associated with the interrupt is allowed to execute for C time units.

After servicing an interrupt, or completing the execution of the corresponding sporadic task's main body, partitions signal completion to the hypervisor. The hypervisor dismisses the watchdog timer associated with the interrupt and updates the remaining server capacity, taking into account the time required to complete the remainder of this routine. The execution context and state information is then restored. This procedure takes C^{cs2} time units.

4.2 Case Study

The proposed approach was heavily motivated by the requirements of the automotive industry and takes into account implementation overheads. In this section a case study is formed to evaluate the approach proposed in Chapter 3.

4.2.1 Application Description

The case study evaluation was performed using ECU application code that was provided by ETAS Ltd. The application code consisted of a set of AUTOSAR TASKs of a Mercedes-Benz M160 engine controller. The functionality of the application code includes controlling the air flow, idle speed, and fuel injection.

Each TASK has a unique period and a set of sub-tasks that are to be executed at that period. The provided taskset was split into two partitions by an expert with respect to the individual task functionality. Each task was then classified as sporadic or periodic, with respect to their real-time requirements. The resulting partition configuration is shown in Table 4.1.

4.2.2 Task Measurement

In this section we describe the methodology followed for obtaining the timing information for application tasks and hypervisor overheads in the system. The resulting timing information for all tasks and hypervisor overheads is then presented.

Application Tasks

The next step for the case study is to analyse the provided ECU code, in order to be fit to the proposed model. The source contained only application code, without the underlying OS. The minimum information that is required for the proposed model is the task execution times and the period, which was elicited from the naming convention used for TASK names. The execution times of the tasks were obtained using a measurement-based approach.

The first part of the timing analysis was to study the provided code. Upon inspection, the code was primarily linear with minimal branching. Numeric calculations and variable conversions were the primary operations performed by the code. The input/output of the tasks was made by reading/writing to external variable, each with a

Partition	*Task Name*	*T (ms)*	*Type*
0	τ_0	1	Sporadic
	τ_1	1	Sporadic
	τ_2	1	Sporadic
	τ_3	1	Sporadic
	τ_4	1	Sporadic
	τ_5	10	Periodic
	τ_6	10	Periodic
	τ_7	10	Periodic
	τ_8	10	Periodic
	τ_9	10	Periodic
	τ_{10}	10	Periodic
	τ_{11}	20	Periodic
	τ_{12}	20	Periodic
	τ_{13}	100	Periodic
	τ_{14}	100	Periodic
	τ_{15}	100	Periodic
	τ_{16}	100	Periodic
	τ_{17}	100	Periodic
	τ_{18}	100	Periodic
	τ_{19}	100	Periodic
	τ_{20}	100	Periodic
	τ_{21}	100	Periodic
	τ_{22}	100	Periodic
	τ_{23}	100	Periodic
1	τ_{24}	10	Periodic
	τ_{25}	10	Periodic
	τ_{26}	10	Periodic
	τ_{27}	50	Periodic
	τ_{28}	100	Periodic
	τ_{29}	100	Periodic
	τ_{30}	100	Periodic
	τ_{31}	1000	Periodic

Table 4.1: Automotive engine controller taskset.

clearly defined range of valid values.

Measuring the task execution times required modifications to the provided code, since the OS code was not available. Specifically the code was modified with definitions of all the external variables and structures. The defined variables were initialised using valid values elicited from the coding convention used for the comments accompanying the definition of the external variables. A mechanism for running the tasks was also implemented, therefore making the application code runnable on operating systems capable of running GCC, such as Windows and Linux.

The chosen hardware platform to execute the modified application code on was a Raspberry PI Model B. This hardware platform was chosen for the following reasons:

- **Linux support:** The Linux environment support eliminates the need for porting the application code to run on a bare-metal environment. Additionally, the filesystem used by Linux can be used to log the observed execution times rather than requiring more complicated mechanisms of tracing the execution of the application code like streaming over GPIO or JTAG.

- **Compiler toolchain:** Typically, Linux distribution have access to a GCC compiler and a `make` build system. The modified application code can therefore be compiled and run on both the Raspberry PI and a host environment for testing, without the need to modify the source code or build configuration.

- **ARM1176JZF-S:** Raspberry PI features the same processor that was considered by the industrial sponsor for porting RTA-HV at the time. The application tasks primarily performed numeric calculations, without the use of OS calls. The timing measurements of the application code would therefore be obtained using the same instruction set and timings as if bare-metal version.

The main concern about using Linux for obtaining time measurements is the interference from periodic kernel-related interrupts, which can cause spikes in the time measurements. As shown in Figure 4.2, the use of a real-time kernel, such as the one bundled with Emlid, has significantly lower and more predictable preemption latency. Occasional spikes in the observed execution times can also be treated as outliers via statistic analysis. The measurements were therefore taken on a Raspberry Pi Model B running Emlid, which is a Debian-based Linux distribution with a real-time kernel.

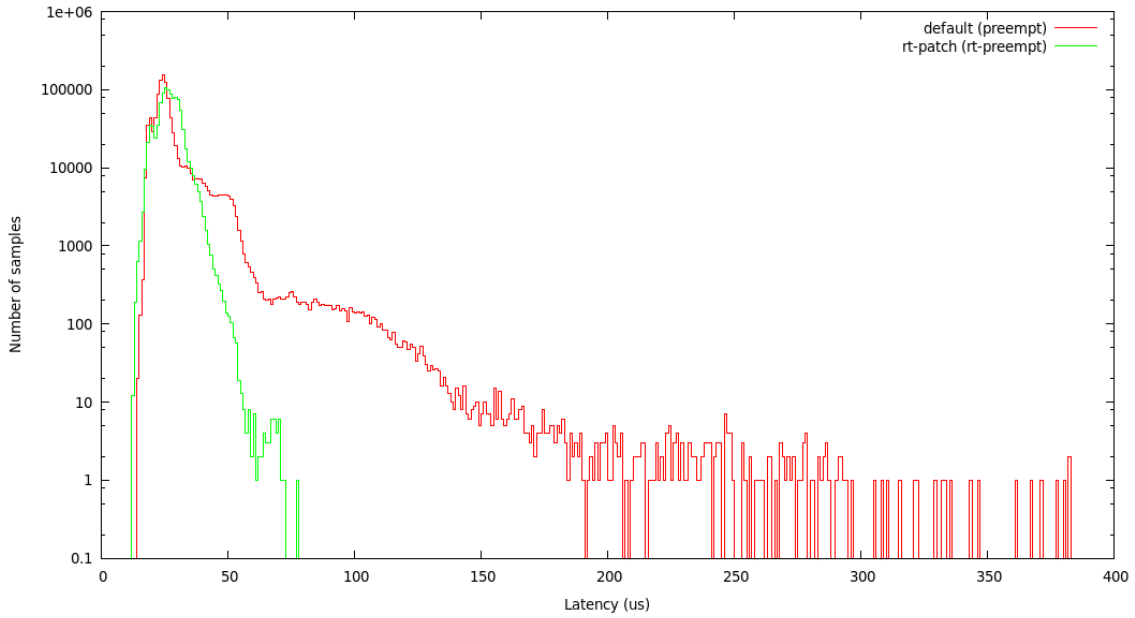


Figure 4.2: Comparison of preemption latency between the default kernel and a real-time patched kernel [35].

Executing and measuring a task's execution time was not possible as the execution times of the tasks were short, exceeding the the capabilities of the high resolution timer that was provided by the kernel. Therefore, to measure the task execution times, each task was executed 1 million times, taking an average execution time. This process was repeated 500 times per task, therefore having a sample of 500 estimated execution times per task. As previously identified, the execution path of the tasks was mostly linear. It is therefore assumed that the observed average-case execution times are representative of the worst-case execution times.

Table 4.2 summarises the sampled execution times. From the table, there is a small difference between mean and median as well as a small standard deviation of the execution times. This confirms that there is little variation between each measured execution time, as expected from the minimal branching that was observed during the inspection of the code. The data of Figure 4.3 represents the relationship between WCET and periods of all tasks taskset. Analysis using Pearson's r showed no correlation between the WCET and the period of tasks, $r(30) = -0.0622$, $p = 0.735$ (two-tailed test). Box plots of the execution times for all application tasks are listed in Appendix A.

Note: After obtaining all application task time measurements and conducting the experiments detailed in Chapters 4 and 6 a secondary timing analysis was performed. For

Task	Min	Max	Mean	Median	Std
τ_0	516.06	518.49	516.93	516.87	0.42
τ_1	3351.14	3641.27	3371.46	3353.75	64.65
τ_2	955.60	959.29	956.59	956.36	0.74
τ_3	188.55	189.35	188.91	188.86	0.16
τ_4	612.29	615.18	613.41	613.30	0.52
τ_5	221.49	222.28	221.85	221.78	0.16
τ_6	444.86	447.13	445.49	445.45	0.24
τ_7	123.99	124.65	124.23	124.18	0.13
τ_8	361.33	362.95	361.90	361.78	0.31
τ_9	361.36	363.60	362.08	361.95	0.37
τ_{10}	497.33	499.70	498.18	498.14	0.45
τ_{11}	361.41	363.16	361.99	361.86	0.34
τ_{12}	420.06	424.44	420.90	420.88	0.38
τ_{13}	361.39	363.38	361.97	361.84	0.34
τ_{14}	419.97	422.01	420.84	420.83	0.34
τ_{15}	1035.55	1039.08	1036.62	1036.49	0.59
τ_{16}	248.28	249.26	248.70	248.63	0.18
τ_{17}	1084.03	1088.36	1084.98	1084.87	0.63
τ_{18}	2523.01	2537.88	2526.73	2525.13	2.97
τ_{19}	361.39	363.18	362.03	361.91	0.32
τ_{20}	338.45	340.07	339.01	338.91	0.27
τ_{21}	372.68	374.80	373.27	373.19	0.32
τ_{22}	343.02	344.00	343.52	343.45	0.19
τ_{23}	1325.86	1330.68	1327.37	1327.15	0.89
τ_{24}	480.63	483.01	481.40	481.38	0.40
τ_{25}	460.15	462.09	460.82	460.79	0.24
τ_{26}	173.33	174.15	173.63	173.58	0.14
τ_{27}	203.72	204.47	204.04	203.98	0.16
τ_{28}	492.54	493.55	493.15	493.14	0.21
τ_{29}	505.74	507.78	506.46	506.44	0.24
τ_{30}	2351.58	2373.03	2356.22	2354.42	3.73
τ_{31}	755.56	758.30	756.43	756.47	0.26

Table 4.2: Descriptive statistics for task execution times in *ns*.

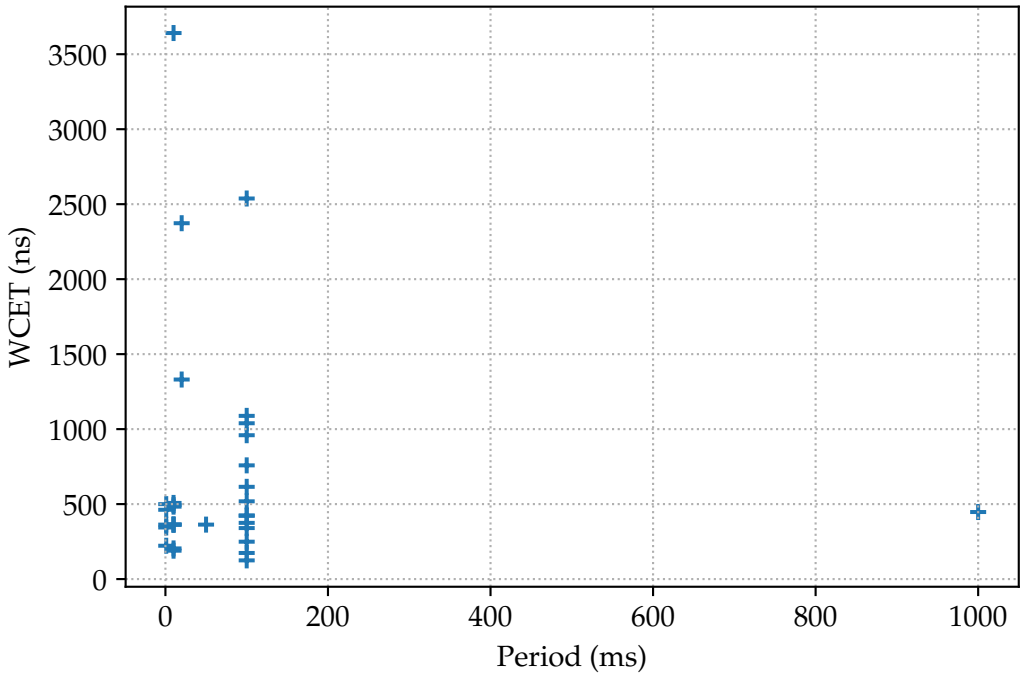


Figure 4.3: Relationship between period and WCET.

the secondary timing analysis the OS kernel was patched to allow for the use of the System Performance Monitor (SPM) by disabling the system validation registers. The SPM provides a cycle counter (co-processor `CP15`, register `c15`, operational register `c12`), which was used to obtain cycle-accurate execution times of the application tasks. The measurements taken using the cycle-accurate counter were within a 5% error margin. For the purpose of constructing a representative taskset and given the time constraints of this project, the original measurements were used.

Hypervisor Overheads

To calculate the hypervisor overheads, a partial hypervisor implementation was built. The hypervisor overheads in the system are responsible for the replenishment of the server capacity and handling forwarding and returning from interrupts.

Figure 4.4 outlines the operations performed by the hypervisor after an event triggers the fire of an interrupt. When an interrupt is triggered that needs to be handled by a partition the hypervisor's ISR is entered in **IRQ** mode. The first step is to change the processor's mode to **System**. In **System** the hypervisor has access to the registers used by the partition, which are saved to the corresponding VRB. The execution context is pushed on the hypervisor's stack. An acknowledge signal is then sent to the interrupt

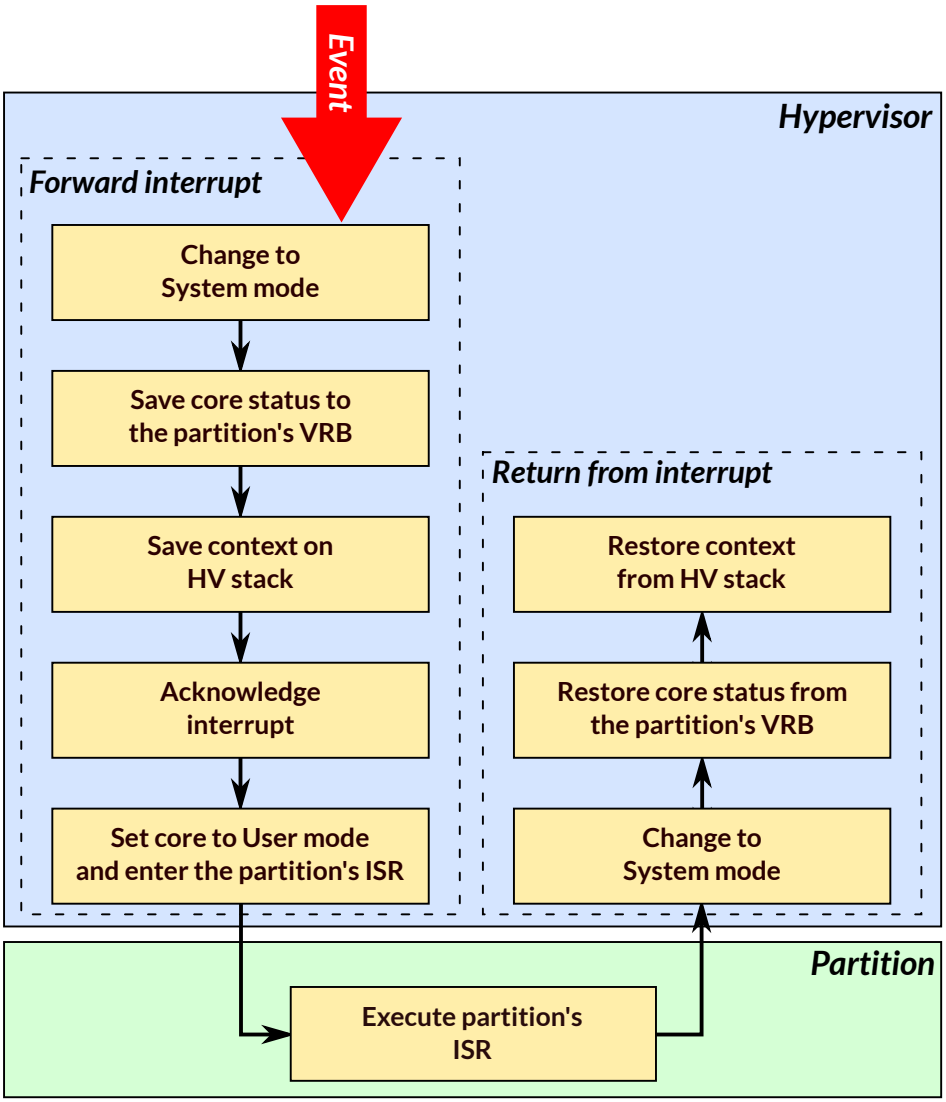


Figure 4.4: ARM1176JZF-S Hypervisor Overhead Routines.

controller and the partition is allowed to execute its ISR in **User** mode.

After the partition finishes the execution of the task released in response to the event it issues a *return from exception* (RFE) hypervisor call. The hypervisor then changes the core's execution mode to **System**, and reads the context from the hypervisor stack. The partition registers are restored from the corresponding VRB and the system resumes execution by restoring the context; the context is popped from the hypervisor's stack.

The replenishment of a server's capacity combines operations from both forward and return from interrupt routines. To replenish a server's capacity the context and status information is saved and the interrupt corresponding to the server's replenishment timer is acknowledged. The server's capacity is then set to the appropriate value and all interrupt lines related to the server are unmasked. The context is then restored and the

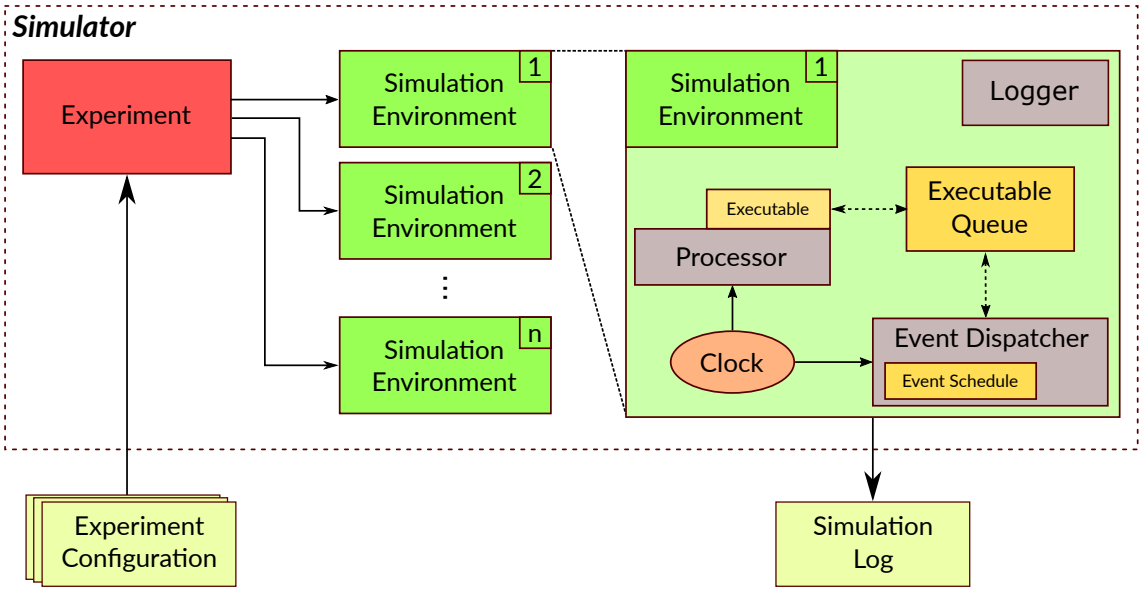


Figure 4.5: Simulator abstract architecture overview.

system proceeds to execute or resume execution of the highest priority entity.

Specialised versions of the described hypervisor routines are generated for every interrupt line and server at the system configuration service. This is common practice in the automotive industry to avoid having dynamic code, which can hinder the performance and timing predictability of the system. The resulting components are therefore short and linear, therefore making it feasible to accurately calculate their WCET using static analysis. Specifically, the WCET of the developed hypervisor components was calculated using the maximum cycles required for the ARM1176JZF-S instructions to execute [6].

Table 4.3 summarises the hypervisor overheads:

Hypervisor Overhead	WCET (*ns*)
Forward interrupt	363
Return from interrupt	139
Replenish server capacity	553

Table 4.3: Hypervisor overheads WCET.

4.3 Simulator Implementation

A Java-based simulator was developed to allow the evaluation of the approach proposed in Chapter 3. The simulator's abstract architecture is shown in Figure 4.5. The motivation behind the developed simulator's architecture is to take advantage of parallelism

by creating multiple isolated simulation environments, each executed within a separate thread.

4.3.1 Simulator Overview

The main drivers of the simulator's design and development were to accurately model the behaviour of the architecture described in Chapter 3 and the ability to take advantage of parallelism. Within the system boundary of the simulator an experiment is defined as a set of simulation environments, constructed according to the experiment configuration. The experiment configuration files are provided as an input to the simulator. These include the tasks in the system, the execution servers and the length of the simulation. Each simulation generates a log with the execution summary of each released job in the system. Each simulation environment is an independent entity, allowing the parallel execution of simulations on multi-core, taking advantage of Java's concurrency programming features. All experiments were executed on the York Advanced Research Computing Cluster (YARCC), which is a compute cluster with 840 cores spread across 40 nodes.

4.3.2 Main Simulator Structures

A simulation environment has a system clock and an arbitrary number of timed entities whose state is updated with every clock tick. The timed entities implemented for the purposes of the experiments required for the evaluation of our approach are the processor, an event dispatcher and a logger.

Timed Entities

Timed entities are used as an abstract way of defining all elements in the system whose state depends on the progression of time. All timed entities have a common interface that allows them to advance their state by one time unit.

The *processor* is defined as a timed entity that maintains a queue of *executables* and a currently running executable. With each time step the processor first updates the currently running executable with the highest priority executable that is eligible to run. The current executable's state is then progressed one time step. In the case where there is no executable that is eligible to run the processor is idle until the next time unit.

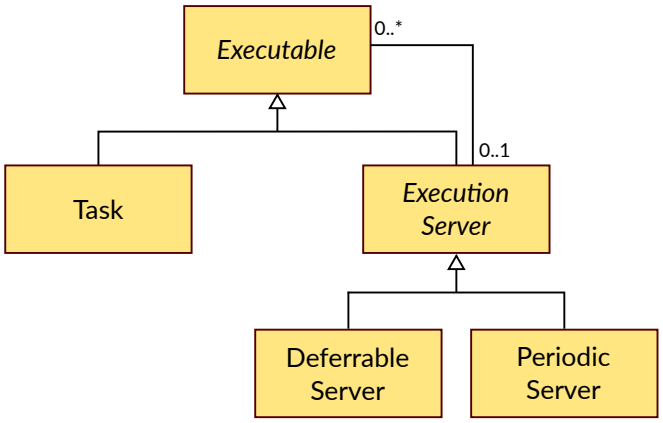


Figure 4.6: Class diagram for executables.

The *event dispatcher* is a timed entity consisting of a queue of events that is used as a mechanism for injecting events into the system. Each event is defined by a set of parameters: absolute release time, an optional period, a template executable entity and the task queue the executable is injected to. An event is released when its absolute release time matches the value of the clock. During an event release, the executable template is used to construct an instance of the executable, which is then added to the appropriate queue. If the event is periodic, a new event is scheduled for the next absolute release time. Being a timed entity, the event dispatcher's state is updated with every tick of the simulation environment's clock. The state of the event dispatcher is strictly updated before updating the processor's state to avoid race conditions,

Executables

Executables within the context of the developed simulator are timed entities that are part of the realisation of the scheduling model. As summarised in Figure 4.6, executables refer to tasks and execution servers. A task can either execute directly on the processor (hypervisor tasks), or within an execution server (partition tasks). An execution server can either be a deferrable server (sporadic tasks) or a periodic server (periodic tasks).

To reflect the desired behaviour of the preemption rules defined in Sections 3.2 and 3.5, each executable is composed by a sequence of regions with their own priority and whether it is preemptive or not. This inherently supports the non-preemptive rule for the sporadic task and periodic server overheads, as well as the capability to raise a tasks priority as dictated by IPCP.

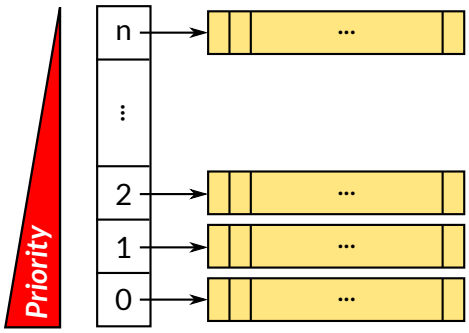


Figure 4.7: Simulator priority queue data structure.

Execution Servers

Execution servers in the simulator have a similar functionality to the processor. Each server has its own currently active executable; nested servers are supported by design, however in the context of our model, the executables associated with execution servers are tasks. With each time step, if a server is the currently executable on the processor, it progresses one time unit, causing its currently active executable to also progress by one time unit.

The capacity of the execution servers is depleted with each clock tick, as described in Section 2.4.1. Specifically, the periodic servers' capacity is depleted with every clock tick if they are the currently running executable. The capacity of a deferrable server is depleted only if it has a currently active executable, otherwise it enters a "not ready" state. Both deferrable and periodic servers enter a "not ready" state, if their capacity is depleted, where they remain until their next replenishment.

Priority Queue

Figure 4.7 summarises the data structure used for the implementation of the priority queues in the system. The priority queue is an array of length $n + 1$, each entry corresponding to a priority level. Each entry contains a queue of executables in order to support cases where a task's execution extends into the next period, therefore having in essence two ready tasks at the same priority level.

Tasks at the same priority level are scheduled using a first-in first-out approach. Specifically, executables that are preempted are placed in front of the queue of their respective priority level. New executable releases are placed at the end of the queue.

4.4 Experiment

The simulator described in Section 4.3 was used to investigate the tightness of the response time analysis of Section 3.7 and the robustness of the system in terms of propagating the effects of deadline misses.

4.4.1 Methodology

To evaluate the tightness of the analysis we compare the maximum utilisation achieved until the analysis indicates a deadline miss with the utilisation achieved until the first deadline miss in the simulation.

Server Parameters

The server parameters were selected in such a way to minimize gaps where the servers have no capacity. The server period used is the minimum of the periods of all tasks within the server as shown in Equation (4.1).

$$T_s = \min_{\forall i} M_{i,s} T_i \tag{4.1}$$

Given the server's period the capacity required to service all tasks is given by Equation (4.2). Specifically, the server capacity is set as the sum of all task execution times over a window of length T_s.

$$C_s = \sum_{\forall i} M_{i,s} \left\lceil \frac{T_s}{T_i} \right\rceil C_i^{tot} \tag{4.2}$$

Priority Assignment

Given a system configuration with task and server definitions, we first assign priorities. Due to the complexity of the model, the priority assignment is performed following a divide and conquer approach. First, the priority space is divided into three priority bands, as described in Section 3.4: hypervisor, sporadic and periodic. Hypervisor tasks are assigned priorities using rate monotonic at the hypervisor priority band. At the periodic priority band, periodic servers are allocated priorities using Audsley's algorithm [8], without taking into account the periodic tasks.

After assigning the priorities to the hypervisor tasks and periodic servers, we assign priorities to the sporadic tasks in a similar manner to the periodic servers. Lastly, we allocate priorities to periodic tasks. The periodic tasks are assigned local priorities within their server. Therefore, starting from the lowest priority periodic server, its serviced tasks are assigned priorities using Audsley's algorithm, without taking into account periodic tasks that execute in different servers.

Generation of Simulator Configurations

Given the priority assignment, the response time analysis that was formulated in Section 3.7 is used to determine whether the system is schedulable. The processor speed is decreased progressively by incrementing the execution times of all tasks until the analysis indicates that the system is schedulable. This assumes that task execution time increases linearly as the processor's speed is decreased. With each iteration, new priorities are assigned and the system is checked for schedulability.

When the system is deemed unschedulable by the analysis the priority ordering is retained and system configurations are generated while scaling down the processor speed until the utilisation reaches 100%. The collection of system configurations is used as input to the simulator described in Section 4.3, along with the additional experiment parameters required to form the experiment configuration.

This procedure is followed to generate experiment configurations for two different server-task mapping. The first mapping was listed in Table 4.1, which corresponds to a two-partition system. The second mapping was devised in order to investigate the propagation of the effects of deadline misses across partitions. Specifically, we further divided the tasks into servers with respect to their period. Tasks within different partitions that have the same period were kept in separate servers. This resulted into a nine-partition system using 1 deferrable server and 8 periodic servers.

Both systems were simulated for a full period $1000ms$. One period was determined as sufficient for the purpose of this experiment, since all tasks are released simultaneously at the start of the simulation. In the worst-case simulations, all tasks execute for WCET, since this is the worst-case scenario. Moreover, always executing for the WCET, removes non-determinism therefore each scenario is executed once. The average-case simulations were repeated five times.

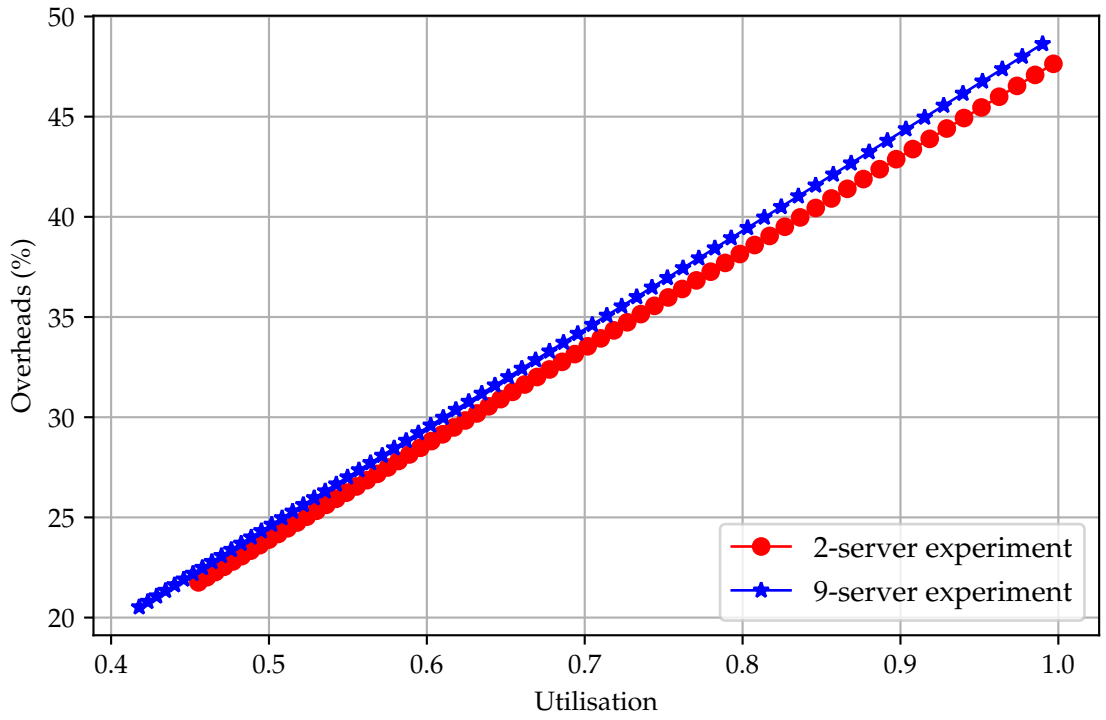


Figure 4.8: Hypervisor overheads with respect to the processor utilisation.

4.4.2 Results

Worst Case

Figure 4.8 shows the relationship between the total utilisation and the utilisation associated with overheads. In both systems, overheads increase linearly with respect to the total system utilisation. The additional periodic servers in the nine-server experiment has introduced a small increase in overheads, which suggests that the use of deferrable servers for sporadic tasks introduces the majority of the overheads in the system.

Figure 4.9 summarises the deadline misses with respect to the system utilisation according to the simulator results. The first deadline miss of the two-partition system occurs at 82% utilisation. The tasks that first miss their deadlines execute within the lowest priority periodic server with short periods with medium-low priority level. Subsequently, all tasks in the lowest priority server missed their deadline when the system reached 88% utilisation.

At 89% utilisation, tasks in the higher priority periodic server started to miss their deadlines. Additionally, the lowest priority sporadic task suffered from deadline misses due to blocking. At 95% utilisation all periodic tasks and lower priority sporadic tasks

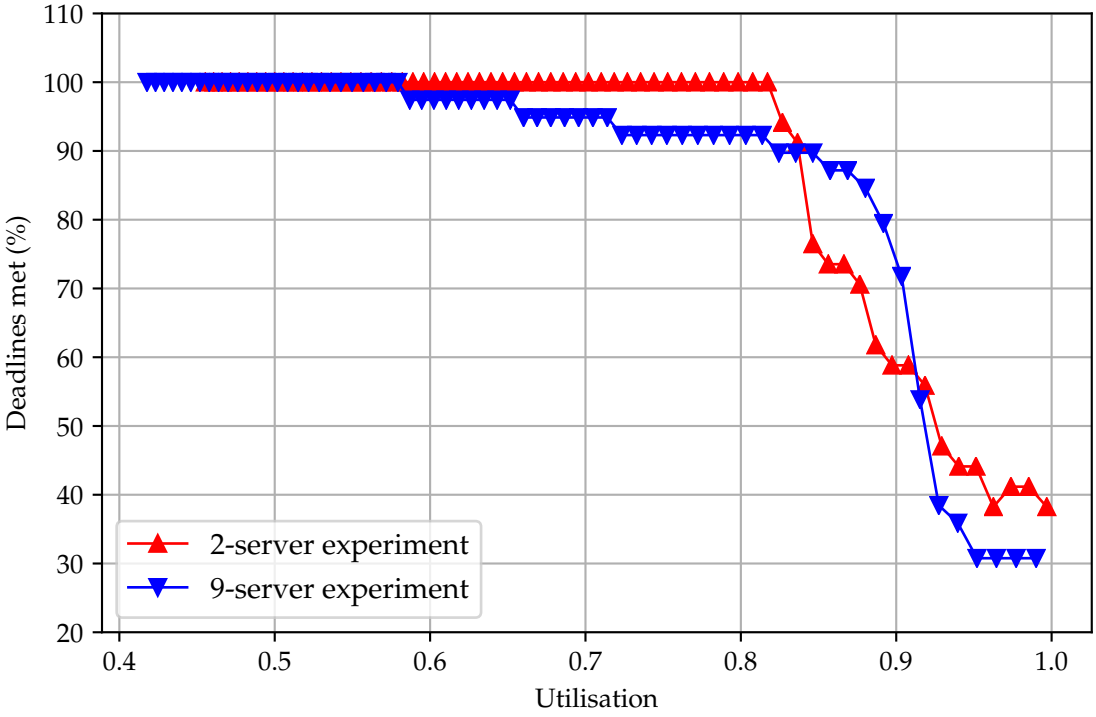


Figure 4.9: Experiment results summary.

suffered from deadline misses. All deadlines were met by hypervisor tasks.

The nine-server configuration has additional hypervisor tasks are responsible with the replenishment of the server capacity. The hypervisor tasks significantly contributed to the interference received by the sporadic tasks, causing the first deadline miss at 59% utilisation. The additional of servers also increased the frequency with which tasks were blocked. At 86% utilisation tasks in lower priority servers begin to miss their deadlines.

Average Case

The simulation results of the previous section show the behaviour of the system in the worst case, where sporadic tasks are always released at their minimum interarrival time. Most commonly, this is not the case, therefore we also examine the behaviour of the system for the case where sporadic tasks do not manifest as periodic.

The simulations were repeated for both systems described above, after configuring the simulator to release sporadic tasks with varying interarrival times. Specifically, the interarrival times of sporadic tasks were taken using a Weibull distribution ($k = 2, \lambda = 1$). The choice of parameters and probability distribution was based on the work done by Maxim et al. [75]. Both sporadic and periodic tasks were simulated using estimated

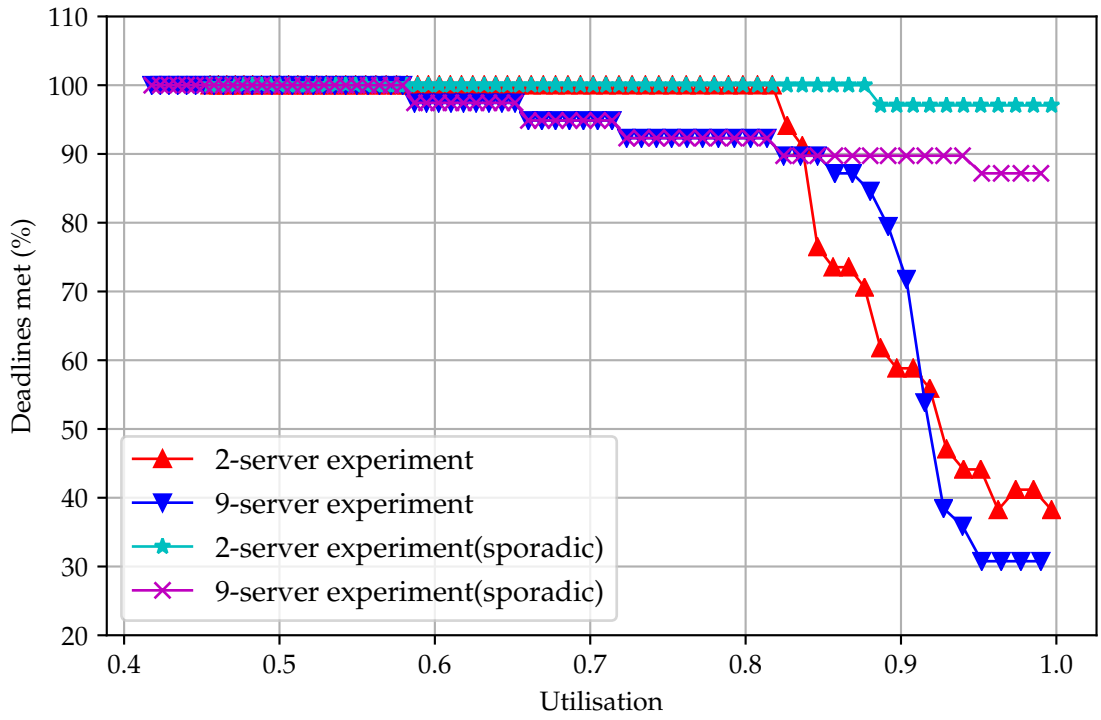


Figure 4.10: Comparison summary of average case vs worst case.

average-case execution times. Inspired by the work done by Hansen et al. [49], the execution time for each release of a task was calculated using a Gumbel distribution ($\mu = 0.8$, $\sigma = 0.125$). The results are summarised in Figure 4.10. Note that the analysis of Section 4.2.2 showed that the WCET and the AVET were very close. This is often not the case, therefore the tasks were simulated using additional variation to better investigate the impact of average case behaviour compared to the worst case.

In the two-server simulation runs, the first deadline miss was observed at 88.6% utilisation. Increasing the utilisation did not cause any further deadline misses. The task that missed its deadline was the lowest priority sporadic task. The experiment results indicate that the use of limited capacity execution servers provided sufficient temporal isolation, as per the requirements of Section 3.1.

The first deadline miss in the nine-server simulation run was observed at 59% utilisation. Additional deadline misses were observed at 66%, 72%, 82% and 95%. Similar to the two-server configuration, the tasks with deadline misses were aperiodic, which further demonstrates FFI between the different partitions in the system as the deadline misses of higher priority tasks did not cause the starvation of lower priority tasks.

Analysis vs Simulations

The simulation results indicated that the first observed deadline miss occurred at a significantly higher utilisation level than the analysis. Specifically, the response time analysis indicated that the system was not schedulable after slowing down the clock speed by a factor of 84 for the two-partition system and 68 for the nine-partition system. The slow-down factors translate to utilisation of 55% and 45% respectively for each system configuration. Compared to the worst-case simulation results, the first observed deadline misses were at 82% utilisation for the two-server configuration and 59% for the nine-partition configuration.

The response time analysis represents the critical instance, which is the worst case behaviour of the system. The difference between the analysis and simulation results is attributed to two factors: release jitter of tasks and blocking.

In the critical instance, the release jitter of tasks, as shown in Section 3.7.2, is defined as $R_s - T_s$. The release is accounted for in the response time analysis for all tasks that execute within deferrable servers and all unbound tasks that execute within periodic servers. As a result of the server parameter selection approach followed, the release jitter was not exhibited in the simulations.

The blocking factor in the response time analysis was defined as the longest non-preemptive region of lower priority tasks and servers. The non-preemptive regions in the system, which as shown in the timing analysis of Section 4.2.2, have large WCET with respect to the main body of the application tasks of the system. This leads to high blocking, therefore making the analysis to achieve low utilisation.

4.5 Evaluation of Architectural Design

In this section we evaluate the architecture proposed in Chapter 3. Specifically, we consider the requirements of Section 3.1 and their fulfilment by the proposed architecture based on the case study performed in this chapter.

A requirement of the proposed system architecture, as discussed in Section 3.1, is the minimization of the response times of sporadic tasks. This was achieved by the priority space that was introduced in Section 3.4. The proposed priority space places hypervisor tasks at the highest priority space, sporadic (or event-driven) tasks are placed at a priority level that is strictly lower than hypervisor tasks but higher than all periodic

(time-driven) tasks. This ensures that all event-driven tasks are serviced as fast as possible, provided that there is enough server capacity and the CPU is not used by the hypervisor.

The primary requirement of the proposed architecture is to provide sufficient temporal isolation between the partitions. The simulation results showed that deadline misses of tasks within an execution server had did not cause tasks in other servers to miss their deadlines. Additionally, the use of execution servers provided a mechanism to guard against starvation of the periodic tasks from the higher-priority sporadic tasks. This was demonstrated in the worst-case simulation of the nine-partition configuration.

The use of deferrable servers for servicing sporadic tasks allowed for lower response times for the serviced tasks. Specifically, in the average case simulations, as the sporadic tasks have varying interarrival times the deferrable servers are able to service the tasks as they arrive, without having their capacity expended while idle. Periodic servers allow for no jitter in bound tasks, which is a contributing factor to the calculation of the response times of tasks in the critical instance.

4.6 Summary

In this chapter we evaluate the proposed system model using a case study provided by ETAS Ltd. Realistic ECU application code was analysed in order to extract timing information to use for realising the model proposed in Chapter 3. The hypervisor overheads were calculated using a partial implementation of a hypervisor. The timing information was used to simulate the system's behaviour using a two and nine server configuration, varying the processor's speed with each run.

The analysis demonstrated that the case study will deliver relatively low utilisation due to the high blocking introduced by the hypervisor overheads and the constraints in the priority assignment due to the partitioning of the priority space of Section 3.4. The majority of the overheads are mainly attributed to the forwarding of interrupts that are necessary for handling sporadic tasks. The hypervisor overheads however are highly dependent on hardware support and the configuration of the system.

In the next chapter the proposed model is extended to support multiple levels of criticality. The mixed-criticality model features two levels of system degradation.

CHAPTER 5

Extension to Mixed-Criticality

The introduction of safety in the automotive industry with the ISO26262 standard on functional safety of road vehicles [54] gave rise to concerns regarding the integration of components with different ASIL in the same physical ECU. AUTOSAR enables the integration of SWC from different vendors, requiring modifications to the configuration of the RTE and BSW [79]. From a safety-critical perspective AUTOSAR lacks the required separation mechanisms. Specifically a failure in an AUTOSAR SWC typically results in an ECU reset. In the case where all SWCs are of the same ASIL this is acceptable, however in the case of mixed-criticality this could potentially allow a low criticality component to interfere with a higher criticality one. This directly violates *freedom from interference* (FFI), as dictated by ISO 26262-6:2011 Annex D [54].

A possible way of achieving FFI is to ensure that components of different criticality levels are located on separate physical ECUs. Failures in lower criticality components would therefore be isolated and not propagate to other ECUs of higher criticality. This approach could potentially have little to no benefit on the number of physical ECUs, which, as identified in Section 1.1, is one of the main drivers to the high development costs for software and hardware in vehicles. The use of a hypervisor can provide the necessary isolation between its partitions to allow the integration of multiple ECU images on a single physical ECU, while enforcing FFI.

This chapter details an extension to the model described in Chapter 3, allowing components of three criticality levels, LO, MI and HI, to execute on the same hardware using a hypervisor. In the context of the proposed model, criticality is a property of

the partitions. Specifically all tasks in a partition have the same criticality level of the partition.

The proposed mixed-criticality model features two modes of degradation and takes advantage of the low utilisation that was introduced by the requirement for low sporadic task latency in order to provide the additional capacity required by high-criticality tasks to meet their deadlines. In the first degraded mode all sporadic tasks are migrated in periodic servers, providing additional capacity due to the lower overheads and higher utilisation. Tasks are only dropped during the second degraded mode, where only HI criticality partitions are allowed to execute.

5.1 Mixed Criticality Task Model

The proposed mixed-criticality model supports three levels of criticality, LO, MI, HI, with $HI > MI > LO$. The level of criticality is used as a property of the partitions, enforcing that all tasks within a partition are of the same level of criticality. The task model is defined in a similar manner as the model proposed by Vestal [103]. A task τ_i is defined by the tuple $(\vec{C_i^{cs1}}, \vec{C}_i, \vec{C_i^{cs2}}, \vec{T}_i, \vec{P}_i, \mathcal{L}_i)$, where:

- $\vec{C_i^{cs1}}$: A vector containing the WCET of the implementation overheads before the execution of the task's main body for each criticality level, $C_i^{cs1}(LO)$, $C_i^{cs1}(MI)$ and $C_i^{cs1}(HI)$.
- $\vec{C}_i$: A vector containing the WCET of the task's main body at each criticality level. $C_i(LO) \leq C_i(MI) \leq C_i(HI)$.
- $\vec{C_i^{cs2}}$: The WCET of the overheads after the execution of the task's main body for each criticality level, $C_i^{cs2}(LO)$, $C_i^{cs2}(MI)$ and $C_i^{cs2}(HI)$.
- $\vec{T}_i$: The period of the task at each level of criticality, $T(LO) \geq T(MI) \geq T(HI)$.
- $\vec{P}_i$: The priority level of the task at each criticality level, $P_i(LO)$, $P_i(MI)$, $P_i(HI)$.
- $\mathcal{L}_i$: The criticality level of the task (ie. LO, MI, HI).

In the mixed-criticality task model, the preemption rules are as specified in Section 3.2. The main difference between the single criticality and mixed-criticality model is the ability of tasks to change from sporadic to periodic, depending on which server they use to execute.

5.2 Mixed Criticality Execution Servers

Consider a system composed of a set of k partitions, $p_0 ... p_{k-1}$. Similarly to the single-criticality model, the CPU time used by each partition is managed using execution servers. Each partition can have one deferrable sever to handle its sporadic tasks and a periodic server for its periodic tasks.

An execution server s_i is defined by the tuple $(\vec{C_i^{cs1}}, \vec{C}_i, \vec{C_i^{cs2}}, \vec{T}_i, \vec{P}_i, \mathcal{L}_i)$, where:

- $\vec{C_i^{cs1}}$: A vector containing the context switching overheads required by a periodic server prior to its execution. For deferrable servers, the context switch overheads are incorporated in the sporadic task definition, therefore $C^{cs1} = 0$.
- $\vec{C}_i$: A vector containing the server's capacity for each criticality level.
- $\vec{C_i^{cs2}}$: A vector containing the context switching overheads after a periodic server uses up its capacity. Similarly with $\vec{C_i^{cs1}}$, for deferrable servers this is part of the sporadic task definition, and is therefore set to 0.
- $\vec{T}_i$: The replenishment period of the execution server for each criticality level.
- $\vec{P}_i$: The server's priority for each criticality level. This is defined as the maximum of the priorities of all serviced tasks.
- $\mathcal{L}_i$ The criticality level of the server (ie. LO, MI, HI). Execution servers within the same partition are of the same criticality level.

The association matrix is defined as a vector, $\vec{M}_{i,s}$, allowing a different task-server mapping for each criticality level. This is used to take advantage of the latency-utilisation trade-off of the model described in Chapter 3, by allowing different task-server mappings are required for each criticality level.

5.3 Execution Modes

Figure 5.1 summarises the mode transitions performed by the system to support multiple levels of criticality. A requirement of implementing the mixed-criticality model is to monitor the execution time of all partition tasks.

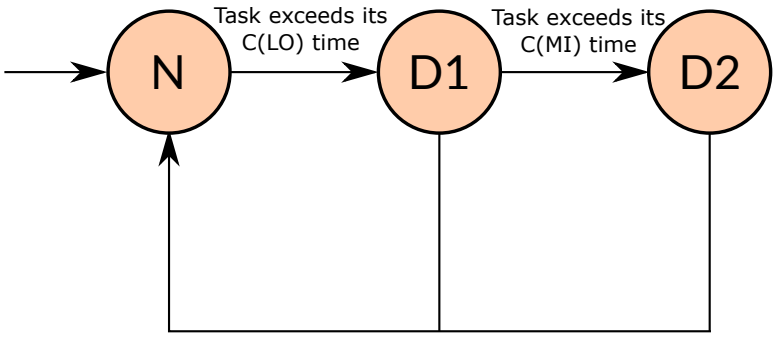


Figure 5.1: State transitions for the mixed-criticality model.

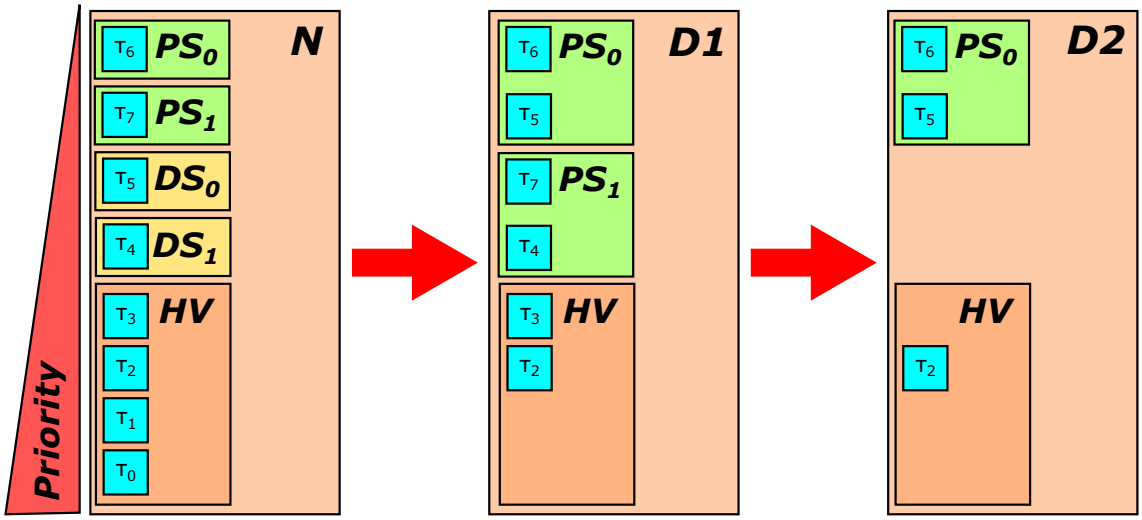


Figure 5.2: Example task priorities in the mixed-criticality model.

5.3.1 Normal Execution Mode (N)

The initial mode the system executes in is referred to as normal, or **N**. While in **N** mode, the scheduling approach described in Chapter 3 is used. Specifically, each partition is associated with a deferrable server and a periodic server. For every server in the system a hypervisor task is required in order to replenish its capacity at the start of its period. All partition tasks are associated with the corresponding partition's servers.

Figure 5.2 examples a simple two-partition system, p_0 and p_1, each having an sporadic and a periodic task. p_0 is a HI criticality partition, whereas p_1's criticality is level MI. All partition tasks are handled by deferrable and periodic servers, as shown in Figure 5.2 and Equation (5.1). The servers' capacity is replenished by the hypervisor tasks

τ_0, τ_1, τ_2 and τ_3.

$$M(LO) = \begin{array}{c} \\ \tau_0 \\ \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \\ \tau_6 \\ \tau_7 \end{array} \begin{array}{c} \begin{array}{ccccc} HV & DS_0 & DS_1 & PS_0 & PS_1 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{5.1}$$

The execution time of the partition tasks are monitored by the hypervisor. In the case where a partition task exceeds its *LO* criticality WCET a mode change is triggered (**N**→**D1**).

5.3.2 First Degraded Execution Mode (D1)

During the mode change to **D1** all sporadic tasks are migrated to their corresponding partition's periodic servers. Since the deferrable servers are no longer in use, the hypervisor tasks responsible for replenishing their capacity are dropped. Equation (5.2) and Figure 5.2 demonstrate the resulting task allocation after the transition to **D1**.

$$M(MI) = \begin{array}{c} \\ \tau_0 \\ \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \\ \tau_6 \\ \tau_7 \end{array} \begin{array}{c} \begin{array}{ccccc} HV & DS_0 & DS_1 & PS_0 & PS_1 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{5.2}$$

As identified in Section 2.4.1, periodic servers are superior to deferrable servers in terms of utilisation [30]. Moreover, as it was identified in Section 4.4.2, deferrable servers have considerable overheads. With the migration to periodic servers, the overheads as-

sociated with using deferrable servers are used as additional capacity to provide tasks with enough capacity to execute at the MI criticality level. The additional capacity available to the periodic server associated with a partition p_n during the **N**→**D1** transition is given by Equation (5.3).

$$\begin{aligned} C^{+}_{N\rightarrow D1}\left(T_{PS_n}(MI)\right) = & \left\lfloor \frac{T_{PS_n}(MI)}{T_{DS_n}(LO)} \right\rfloor C^{rep} + \\ & \sum_{j \bullet M_{j,DS_n}(LO)=1} \left\lfloor \frac{T_{PS_n}(MI)}{T_j(LO)} \right\rfloor \left(C_j^{cs1}(LO) + C_j^{cs2}(LO) \right) - \\ & \sum_{k \bullet M_{k,DS_n}(LO)=1} \left\lfloor \frac{T_{PS_n}(MI)}{T_k(MI)} \right\rfloor \left(C_k^{cs1}(MI) + C_k^{cs2}(MI) \right) \end{aligned} \tag{5.3}$$

If no tasks are pending to execute and an idle tick is detected, the system reverts to the normal execution mode, **N**. In the case where a task executes for more than its MI WCET a mode change is triggered and the system executes in the second degraded mode **D2** (**D1**→**D2**).

5.3.3 Second Degraded Execution Mode (D2)

The second degraded mode, **D2**, is used as a last resort in order to ensure that HI criticality tasks are able to meet their deadlines. In the **D2** mode lower criticality partitions are dropped, providing the additional capacity required for tasks to execute for their HI criticality WCET.

$$M(MI) = \begin{array}{c} \\ \tau_0 \\ \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ \tau_5 \\ \tau_6 \\ \tau_7 \end{array} \begin{array}{c} \begin{array}{ccccc} HV & DS_0 & DS_1 & PS_0 & PS_1 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \tag{5.4}$$

As shown in Equation (5.4), p_1 is of MI criticality, therefore the periodic server, PS_1,

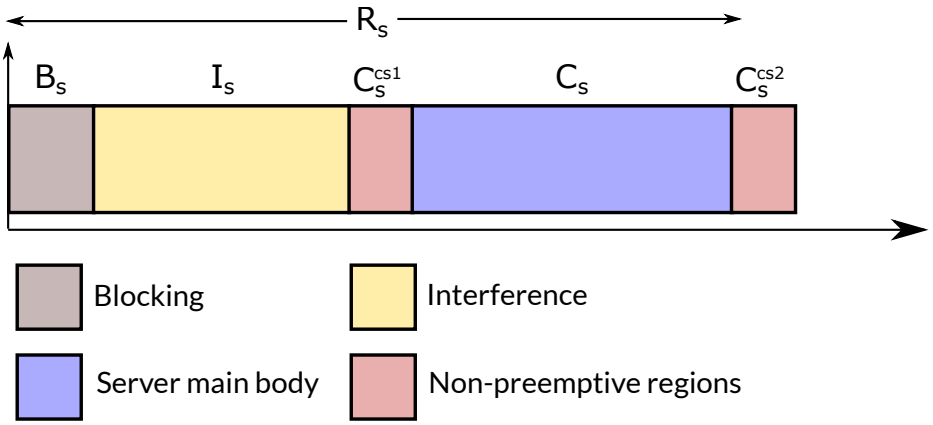


Figure 5.3: Server critical instance.

which is responsible for servicing its tasks is no longer allowed to execute. With PS_1 not allowed to execute, τ_3, which is responsible for replenishing its capacity, is also dropped.

The additional capacity available to the HI criticality partitions given a window of length w is given by Equation (5.5).

$$C^{+}_{D1 \to D2}(w) = \sum_{L_j \leq MI} \left\lfloor \frac{w}{T_j(MI)} \right\rfloor \left(C_i^{cs1}(MI) + C_i(MI) + C_i^{cs2}(MI) \right) \tag{5.5}$$

Similar to **D1**, the system is able to revert to the **N** mode if no task is pending for execution and an idle tick is detected.

5.4 Response Time Analysis

The response time analysis for the mixed criticality level is performed using the approach followed by Adaptive Mixed-Criticality (AMC) [12]. The system is schedulable if all task deadlines are met at each execution mode, **N**, **D1** and **D2**, and during the mode changes **N**→**D1**, **D1**→**D2**, **D1**→**N** and **D2**→**N**.

5.4.1 Server Schedulability

As with the single criticality model, the first part of the analysis is the calculation of the server response times. For the server response time analysis servers are treated as regular tasks with interference from higher priority hypervisor tasks and higher priority servers. The critical instance of an execution server s at citicality level ℓ, where $\ell \in \{LO, MI, HI\}$, is shown in Figure 5.3.

Symbol	*Description*
$\vec{C_i^{cs1}}$	A vector containing the time required by the overheads before the execution of the main body of a task or server for each criticality level.
$\vec{C_i}$	A vector containing the time required by the main body of a task or the capacity of a server for each criticality level.
$\vec{C_i^{cs2}}$	A vector containing the time required by the overheads after the execution of the main body of a task or server for each criticality level.
$\vec{T_i}$	A vector containing the period of a task or server at each criticality level.
$\vec{P_i}$	A vector containing the priority of a task or server at each criticality level.
$C_i^{cs1}(\ell)$	Time required by the overheads before the execution of the main body of a task or server at criticality level ℓ.
$C_i(\ell)$	Execution time required by the main body of a task or the capacity of a server at criticality level ℓ.
$C_i^{cs2}(\ell)$	Time required by the overheads after the execution of the main body of a task or server at criticality level ℓ.
$T_i(\ell)$	The period of a task or server at criticality level ℓ.
$P_i(\ell)$	The priority of a task or server at criticality level ℓ.
$\mathcal{L}_i$	The criticality level of a task or server.
$J_i(\ell)$	Release jitter at criticality level ℓ.
B_i	Blocking received by tasks of priority lower than P_i.
$M_{i,s}(\ell)$	Returns 1 if the task tau_i executes using the server s at criticality level ℓ.
R_i	The response time of a task or server.
$L_i^s(w)$	The load on a server s at the priority level P_i over the length of a window w.
$I_i^\ell(w)$	The interference received by a task or server at priority level P_i when the system executes at criticality level ℓ.
$\mathrm{lp}(i)$	The set of tasks or servers of lower priority than P_i.
$\mathrm{hp}(i)$	The set of tasks or servers of higher priority than P_i.
$async$	The set of sporadic tasks.
$sync$	The set of periodic tasks.
hv	The set of hypervisor tasks.
PS	The set of periodic servers.
DS	The set of deferrable servers.
$C^+_{a\to b}(w)$	The additional capacity received over a window w by after the mode change **a**→**b**.
C^{rep}	The time required to replenish a server's capacity.

Table 5.1: Table of symbols.

$$B_s^{cs}(\ell) = \max\left\{\max(C_j^{cs1}(\ell), C_j^{cs2}(\ell)) | j \in \text{lp}(s,\ell) \wedge j \in async\right\} \cup \left\{\max(C_j^{cs1}(\ell), C_j^{cs2}(\ell)) | j \in \text{lp}(s,\ell) \wedge j \in PS\right\} \quad (5.6)$$

First we consider the blocking factor of a server s at criticality level ℓ. As shown in Equation (5.6) a server can be blocked by lower priority sporadic tasks or lower priority periodic servers. The response time of a server excludes the $cs2$ region, which can potentially be pushed to the next busy period [105]. Therefore the total blocking of a server s at criticality level ℓ is defined as:

$$B_s(\ell) = \max\left(C_i^{cs2}(\ell), B_i^{cs}(\ell)\right) \quad (5.7)$$

The interference received by a server is dependent on hypervisor tasks and servers with a higher priority ceiling. Given a window of length w, the interference received by a server s at criticality level ℓ is given by:

$$I_s^{\ell}(w) = \sum_{j \in \text{hp}(s)} \left\lceil \frac{w + J_j(\ell)}{T_j(\ell)} \right\rceil \left(C_j^{cs1}(\ell) + C_j(\ell) + C_j^{cs2}(\ell)\right) \quad (5.8)$$

The response time of a server s at criticality level ℓ is given by the recurrence relation:

$$R_s^{\ell} = C_i^{cs1}(\ell) + C_i(\ell) + \max\left(C_i^{cs2}(\ell), B_i^{cs}(\ell)\right) + \quad (5.9)$$

$$\sum_{j \in \text{hp}(s)} \left\lceil \frac{R_s^{\ell} + J_j(\ell)}{T_j(\ell)} \right\rceil \left(C_j^{cs1}(\ell) + C_j(\ell) + C_j^{cs2}(\ell)\right) \quad (5.10)$$

5.4.2 Task Response Times During Normal Mode

In the normal execution mode **N**, the response times of tasks are calculated by adapting the response time analysis of Section 3.7 to use the LO values of the vectors defined in Sections 5.1 and 5.2. To calculate the response time of a task τ_i that executes using server s we first need to calculate the server load. The load of the server s at priority level $P_i(LO)$ given a window of length w is given by:

$$L_i^s(w) = C_i^{cs1}(LO) + C_i(LO) + C_i^{cs2}(LO) + \tag{5.11}$$

$$\sum_{j \in \text{hp}(i)} M_{j,s}(LO) \left\lceil \frac{w + J_j(LO)}{T_j(LO)} \right\rceil \left(C_j^{cs1}(LO) + C_j(LO) + C_j^{cs2}(LO) \right) \tag{5.12}$$

The gaps where server capacity is not available for servicing tasks is given by the expression in Equation (5.13).

$$\left(\left\lceil \frac{L_i^s(R_i^{LO})}{C_s(LO)} \right\rceil - 1 \right) (T_s(LO) - C_s(LO)) \tag{5.13}$$

A partition task can be blocked during the execution of the non-preemptive regions of sporadic tasks and periodic servers:

$$B_i^{cs} = \max \left\{ \max \left(C_j^{cs1}(LO), C_j^{cs2}(LO) \right) | P_j(LO) \leq P_i(LO) \wedge j \in \mathit{async} \right\} \cup$$
$$\left\{ \max \left(C_j^{cs1}(LO), C_j^{cs2}(LO) \right) | P_j(LO) \leq P_i(LO) \wedge j \in PS \right\} \tag{5.14}$$

Therefore, the blocking factor for partition tasks is given by Equation (5.15):

$$B_i = \max \left(C_i^{cs2}(LO), B_i^{cs} \right) \tag{5.15}$$

The interference received by partition tasks varied depending on the type of server they use. In the **N** mode, both deferrable and periodic servers are used, therefore requiring partition tasks may receive interference from different sources. Equation (5.16) shows the interference received by tasks using deferrable servers. Specifically, a task within a deferrable server receives interference from higher priority sporadic tasks and hypervisor tasks.

$$I_i(w) = \sum_{P_j(LO) \geq P_i(LO)} \left\lceil \frac{w + J_j(LO)}{T_j(LO)} \right\rceil \left(C_j^{cs1}(LO) + C_j(LO) + C_j^{cs2}(LO) \right) \tag{5.16}$$

In the case of tasks within periodic servers there are four sources of interference: hypervisor tasks, sporadic tasks, periodic servers of higher priority and higher priority tasks using the same execution server:

$$I_i(w) = \sum_{j \in hv} \left\lceil \frac{w}{T_j(LO)} \right\rceil \left(C_j^{cs1}(LO) + C_j(LO) + C_j^{cs2}(LO) \right) + \tag{5.17}$$

$$\sum_{k \in async} \left\lceil \frac{w + J_k(LO)}{T_k(LO)} \right\rceil \left(C_k^{cs1}(LO) + C_k(LO) + C_k^{cs2}(LO) \right) + \tag{5.18}$$

$$\sum_{m \in \text{hpPS}(s)} \left\lceil \frac{w + J_m(LO)}{T_m(LO)} \right\rceil \left(C_m^{cs1}(LO) + C_m(LO) + C_m^{cs2}(LO) \right) + \tag{5.19}$$

$$\sum_{P_l(LO) \geq P_i(LO)} M_{l,s}(LO) \left\lceil \frac{w + J_l(LO)}{T_l(LO)} \right\rceil \left(C_l^{cs1}(LO) + C_l(LO) + C_l^{cs2}(LO) \right) \tag{5.20}$$

The response time of partition tasks for the **N** mode is given by:

$$R_i^{LO} = C_i^{cs1}(LO) + C_i(LO) + C_s^{cs1}(LO) + \tag{5.21}$$

$$\left(\left\lceil \frac{L_i^s(R_i^{LO})}{C_s(LO)} \right\rceil - 1 \right) (T_s(LO) - C_s(LO)) + B_i + I_i(R_i^{LO}) \tag{5.22}$$

5.4.3 Task Response Times During Degraded Modes

In the degraded modes **D1** and **D2** we use the MI and HI values for the task parameters respectively. The load on a server s running at criticality level ℓ at priority level $P_i(\ell)$ is given by:

$$L_i^s(w) = C_i^{cs1}(\ell) + C_i(\ell) + C_i^{cs2}(\ell) + \sum_{j \in \text{hp}(i)} M_{j,s}(\ell) \left\lceil \frac{w + J_j(\ell)}{T_j(\ell)} \right\rceil \left(C_j^{cs1}(\ell) + C_j(\ell) + C_j^{cs2}(\ell) \right) \tag{5.23}$$

Similarly with the normal execution mode, the gaps where server capacity is not available over a window of length w is given by:

$$\left(\left\lceil \frac{L_i^s(w)}{C_s(\ell)} \right\rceil - 1 \right) (T_s(\ell) - C_s(\ell)) \tag{5.24}$$

In the degraded modes deferrable servers are not used, therefore the blocking is received from the non preemptive regions of lower priority periodic servers.

$$B_i^{cs} = \max \left\{ \left(C_j^{cs1}(\ell), C_j^{cs2}(\ell) \right) | P_j(\ell) \leq P_i(\ell) \wedge j \in PS \right\} \tag{5.25}$$

$$B_i = \max\left(C_i^{cs2}(\ell), B_i^{cs}\right) \tag{5.26}$$

The interference received by partition tasks in the degraded modes over a window of length w is calculated using Equation (5.27). The identified sources of interference are hypervisor tasks, higher priority servers and higher priority tasks that execute within the same server.

$$\begin{aligned} I_i(w) = &\sum_{j \in hv} \left\lceil \frac{w}{T_j(\ell)} \right\rceil \left(C_j^{cs1}(\ell) + C_j(\ell) + C_j^{cs2}(\ell)\right) + \\ &\sum_{m \in \text{hpPS}(s)} \left\lceil \frac{w + J_m(\ell)}{T_m(\ell)} \right\rceil \left(C_m^{cs1}(\ell) + C_m(\ell) + C_m^{cs2}(\ell)\right) + \\ &\sum_{P_l(\ell) \geq P_i(\ell)} M_{l,s}(\ell) \left\lceil \frac{w + J_l(\ell)}{T_l(\ell)} \right\rceil \left(C_l^{cs1}(\ell) + C_l(\ell) + C_l^{cs2}(\ell)\right) \end{aligned} \tag{5.27}$$

The response time of partition tasks during the degraded modes of execution, **D1** and **D2** is given by:

$$R_i^\ell = C_i^{cs1}(\ell) + C_i(\ell) + C_s^{cs1}(\ell) + \left(\left\lceil \frac{L_i^s(R_i^\ell)}{C_s(\ell)} \right\rceil - 1 \right) (T_s(\ell) - C_s(\ell)) + B_i + I_i(R_i^\ell) \tag{5.28}$$

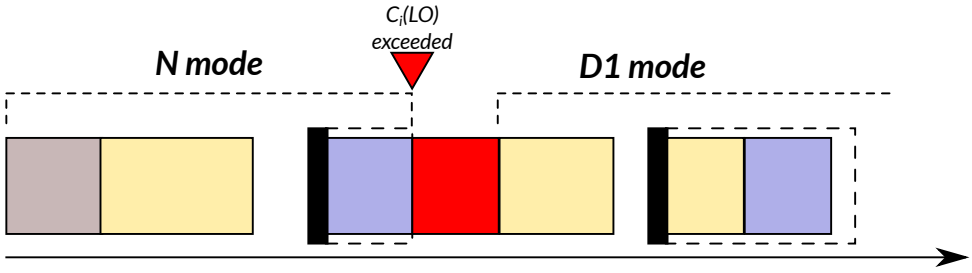
5.4.4 RTA During Mode Changes

In this section we produce the response time analysis, to determine whether the system is schedulable during mode changes. We calculate the response time of a task τ_i that uses server s while executing in **N** mode and s' during degraded execution. We identify three types of mode changes in the system: normal to first degraded (**N**→**D1**), first degraded to second degraded (**D1**→**D2**) and degraded to normal (**D1**→**N**, **D2**→**N**).

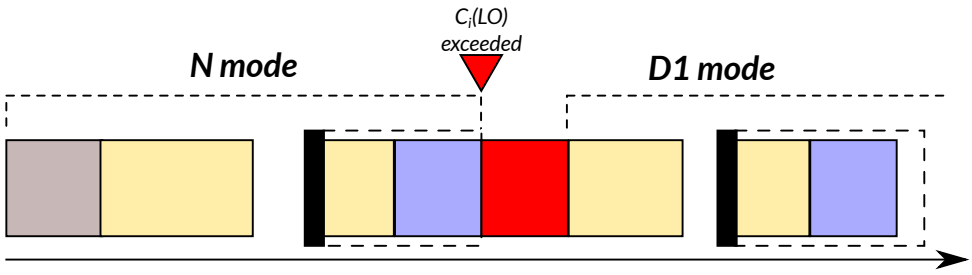
Transition from Normal to First Degraded

The critical instance during the **N**→**D1** mode change is shown in Figure 5.4 for sporadic and periodic tasks. In both task types the task τ_i receives blocking from lower priority tasks and servers. The blocking factor, B_i is given by:

$$B_i = \max\left(C_i^{cs2}(LO), B_i^{cs}\right) \tag{5.29}$$



(a) N→D1 transition for sporadic tasks.



(b) N→D1 transition for periodic tasks.

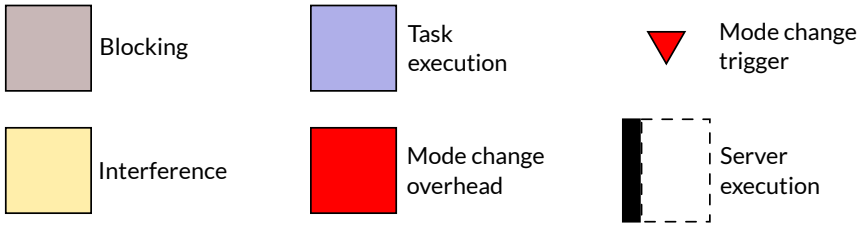


Figure 5.4: Critical instance for the **N→D1** mode change.

where

$$B_i^{cs} = \max\left\{\max\left(C_j^{cs1}(LO), C_j^{cs2}(LO)\right) | P_j(LO) \leq P_i(LO) \wedge j \in async\right\} \cup \left\{\max\left(C_j^{cs1}(LO), C_j^{cs2}(LO)\right) | P_j(LO) \leq P_i(LO) \wedge j \in PS\right\} \quad (5.30)$$

To calculate the first potential gap where there is no remaining server capacity we first require to calculate the load on the server. The load on the server while the system executes in **N** mode is given by:

$$L_i^s(R_i^{LO}) = C_i^{cs1}(LO) + C_i(LO) + C_i^{cs2}(LO) + \sum_{P_j(LO) \geq P_i(LO)} M_{j,s}(LO) \left\lceil \frac{R_i^{LO} + J_j(LO)}{T_j(LO)} \right\rceil \left(C_j^{cs1}(LO) + C_j(LO) + C_j^{cs2}(LO)\right) \quad (5.31)$$

The gap where there is no capacity to service task τ_i while the system executes in **N** mode is given by the expression of Equation (5.32).

$$\left(\left\lceil \frac{L_i^s(R_i^{LO})}{C_s(LO)} \right\rceil - 1 \right) (T_s(LO) - C_s(LO)) \tag{5.32}$$

As shown in Equation (5.33), sporadic tasks receive interference from hypervisor tasks and sporadic tasks with higher priority.

$$I_i^{LO}(R_i^{LO}) = \sum_{P_j(LO) \geq P_i(LO)} \left\lceil \frac{R_i^{LO} + J_j(LO)}{T_j(LO)} \right\rceil \left(C_j^{cs1}(LO) + C_j(LO) + C_j^{cs2}(LO) \right) \tag{5.33}$$

The sources of interference for periodic tasks are hypervisor tasks, sporadic tasks, higher priority periodic servers and higher priority tasks residing in the same server, s.

$$\begin{aligned} I_i^{LO}(R_i^{LO}) = & \sum_{j \in hv} \left\lceil \frac{R_i^{LO}}{T_j(LO)} \right\rceil \left(C_j^{cs1}(LO) + C_j(LO) + C_j^{cs2}(LO) \right) + \\ & \sum_{k \in async} \left\lceil \frac{R_i^{LO} + J_k(LO)}{T_k(LO)} \right\rceil \left(C_k^{cs1}(LO) + C_k(LO) + C_k^{cs2}(LO) \right) + \\ & \sum_{m \in \text{hpPS}(s)} \left\lceil \frac{R_i^{LO} + J_m(LO)}{T_m(LO)} \right\rceil \left(C_m^{cs1}(LO) + C_m(LO) + C_m^{cs2}(LO) \right) + \\ & \sum_{P_l(LO) \geq P_i(LO)} M_{l,s}(LO) \left\lceil \frac{R_i^{LO} + J_l(LO)}{T_l(LO)} \right\rceil \left(C_l^{cs1}(LO) + C_l(LO) + C_l^{cs2}(LO) \right) \end{aligned} \tag{5.34}$$

Transitioning from mode **N** to **D1** has a WCET of $C_{N \to D1}$ time units. Following the transition, we use the MI values for all task and server parameters. Since all sporadic tasks are incorporated into periodic servers it is necessary to account for the higher priority periodic servers and higher priority tasks within the same server. The interference received by τ_i after the transition to **D1** is given by:

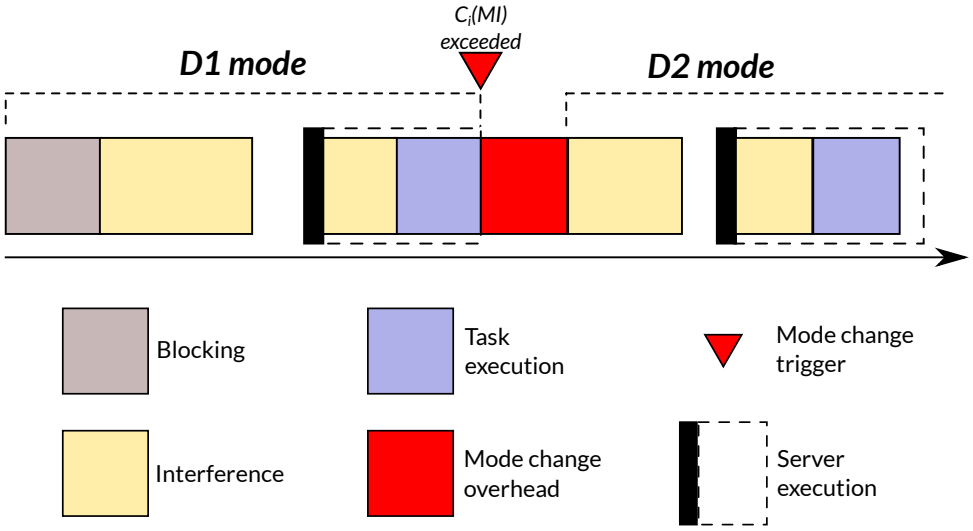


Figure 5.5: Critical instance during the **D1**→**D2** mode change.

$$
\begin{aligned}
I_i^{MI}(R_i^*) = &\sum_{j \in hv} \left\lceil \frac{R_i^*}{T_j(LO)} \right\rceil \left(C_j^{cs1}(LO) + C_j(LO) + C_j^{cs2}(LO) \right) + \\
&\sum_{k \in \text{hpPS}(s)} \left\lceil \frac{R_i^* + J_k(MI)}{T_k(MI)} \right\rceil \left(C_k^{cs1}(MI) + C_k(MI) + C_K^{cs2}(MI) \right) + \\
&\sum_{P_l(LO) \geq P_i(LO)} M_{l,s}(LO) \left\lceil \frac{R_i^* + J_l(LO)}{T_l(LO)} \right\rceil \left(C_l^{cs1}(LO) + C_l(LO) + C_l^{cs2}(LO) \right)
\end{aligned} \tag{5.35}
$$

The resulting response time during a **N**→**D1** mode switch is given by Equation (5.37).

$$
\begin{aligned}
R_i^* = &C_i^{cs1}(LO) + C_i(LO) + C_s^{cs1}(LO) + \qquad (5.36) \\
&\left(\left\lceil \frac{L_i^s(R_i^{LO})}{C_s(LO)} \right\rceil - 1 \right) (T_s(LO) - C_s(LO)) + \\
&\max \left(C_i^{cs2}(LO), B_i^{cs} \right) + I_i^{LO}(R_i^{LO}) + C_{N \to D1} + \\
&\left(\left\lceil \frac{L_i^{s'}(R_i^*)}{C_{s'}(MI)} \right\rceil - 1 \right) (T_{s'}(MI) - C_{s'}(MI)) + I_i^{MI}(R_i^*) \qquad (5.37)
\end{aligned}
$$

Transition from First Degraded to Second Degraded

The critical instance during a mode switch from **D1** to **D2** is shown in Figure 5.5. During the switch all partition tasks execute using periodic servers. The blocking received by

partition tasks during while the system executes in **D1** is defined as the maximum between the longest non-preemptive region of lower priority servers and $C_i^{cs2}(MI)$.

$$B_i = \max\left(C_i^{cs2}(MI), B_i^{cs}\right) \tag{5.38}$$

where

$$B_i^{cs} = \max\left\{\max\left(C_j^{cs1}(MI), C_j^{cs2}(MI)\right) | P_j(MI) \leq P_i(MI) \wedge j \in PS\right\} \tag{5.39}$$

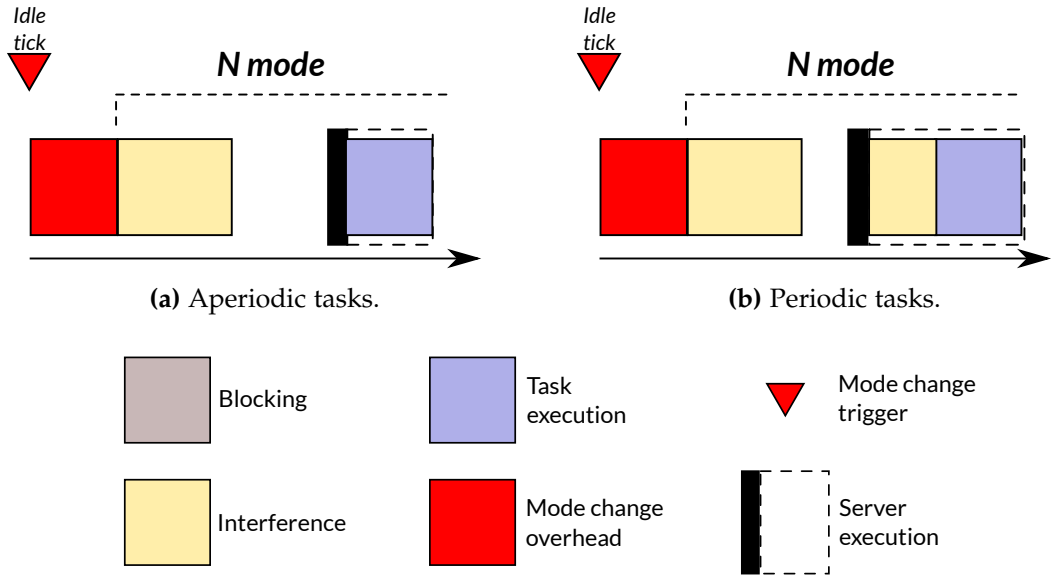
While executing in degraded modes, task τ_i uses server s'. The gap, where s' has no capacity to service τ_i during the execution in the **D1** mode is given by:

$$\left(\left\lceil \frac{L_i^{s'}(R_i^{MI})}{C_{s'}(MI)} \right\rceil - 1\right)\left(T_{s'}(MI) - C_{s'}(MI)\right) \tag{5.40}$$

Prior the mode change, τ_i receives interference from hypervisor tasks and higher priority servers and higher priority tasks that execute within s'. The interference during the execution in **D1** is given by Equation (5.41).

$$\begin{aligned} I_i^{MI}(R_i^{MI}) = & \sum_{j \in hv} \left\lceil \frac{R_i^{MI}}{T_j(MI)} \right\rceil \left(C_j^{cs1}(MI) + C_j(MI) + C_j^{cs2}(MI)\right) + \\ & \sum_{k \in \text{hpPS}(s)} \left\lceil \frac{R_i^{MI} + J_k(MI)}{T_k(MI)} \right\rceil \left(C_k^{cs1}(MI) + C_k(MI) + C_K^{cs2}(MI)\right) + \\ & \sum_{P_l(MI) \geq P_i(MI)} M_{l,s}(MI) \left\lceil \frac{R_i^{MI} + J_l(MI)}{T_l(MI)} \right\rceil \left(C_l^{cs1}(MI) + C_l(MI) + C_l^{cs2}(MI)\right) \end{aligned} \tag{5.41}$$

After the mode switch is triggered, the system spends $C_{D1 \to D2}$ time units. After the mode change τ_i receives interference from the HI criticality tasks in the system, when there is sufficient server capacity available.



(a) Aperiodic tasks. **(b)** Periodic tasks.

Figure 5.6: Critical instance during the degraded to normal mode transitions.

$$I_i^{HI}(R_i^{HI}) = \sum_{j \in hv} \left\lceil \frac{R_i^{HI}}{T_j(HI)} \right\rceil \left(C_j^{cs1}(HI) + C_j(HI) + C_j^{cs2}(HI) \right) + \sum_{k \in \text{hpPS}(s)} \left\lceil \frac{R_i^{HI} + J_k(HI)}{T_k(HI)} \right\rceil \left(C_k^{cs1}(HI) + C_k(HI) + C_K^{cs2}(HI) \right) + \sum_{P_l(HI) \geq P_i(HI)} M_{l,s}(HI) \left\lceil \frac{R_i^{HI} + J_l(HI)}{T_l(HI)} \right\rceil \left(C_l^{cs1}(HI) + C_l(HI) + C_l^{cs2}(HI) \right) \tag{5.42}$$

The response time of a task τ_i using server s' during the **D1**→**D2** mode switch is given by:

$$R_i^* = C_i^{cs1}(MI) + C_i(MI) + C_s^{cs1}(MI) + \left(\left\lceil \frac{L_i^{s'}(R_i^{MI})}{C_{s'}(MI)} \right\rceil - 1 \right) (T_{s'}(MI) - C_{s'}(MI)) + \max \left(C_i^{cs2}(MI), B_i^{cs} \right) + I_i^{MI}(R_i^{MI}) + C_{D1 \to D2} + \left(\left\lceil \frac{L_i^{s'}(R_i^*)}{C_{s'}(HI)} \right\rceil - 1 \right) (T_{s'}(HI) - C_{s'}(HI)) + I_i^{HI}(R_i^*) \tag{5.43}$$

Transition from Degraded States to Normal

As shown in Figure 5.6, the transition from the degraded modes to normal is triggered by an idle tick after all released tasks have finished executing. For the critical instance

we assume that all tasks in the system are released immediately after the mode change operation starts executing. During the **D1**→**N** and **D2**→**N** mode changes, the response time of task τ_i is calculated following a similar approach to the one described in Section 5.4.2. The key difference in the response time of τ_i is that instead of the blocking it suffers from the interference of the mode switch overhead, $C_{D\rightarrow N}$. The response time of τ_i during the **D1**→**N** and **D2**→**N** mode changes is given by:

$$\begin{aligned} R_i^* =& C_i^{cs1}(LO) + C_i(LO) + C_s^{cs1}(LO) + \\ & \left(\left\lceil \frac{L_i^s(R_i^*)}{C_s(LO)} \right\rceil - 1 \right) (T_s(LO) - C_s(LO)) + C_{D\rightarrow N} + I_i(R_i^*) \end{aligned} \tag{5.44}$$

5.5 Summary

In this chapter we extend the system model to support multiple levels of criticality. In the proposed model three criticality levels are supported: *LO*, *MI* and *HI*. The system can execute in one of three modes: **N**, **D1** and **D2**.

A **N**→**D1** mode switch is triggered if a task exceeds its *LO* criticality WCET. In the first degraded mode, **D1**, tasks within deferrable servers are migrated to periodic servers, therefore taking advantage of the reduced overheads to extend the periodic servers capacity. This execution mode takes advantage of the latency-utilisation trade-off, as well as the higher overheads that result from using high priority deferrable servers during the normal execution mode, **N**. In **D1** all tasks are scheduled, using the additional capacity available after the task migrations to satisfy their *MI* timing requirements.

Another mode switch (**D1**→**D2**) is triggered if a task executing while the system is in **D1** mode exceeds its *MI* WCET. The **D2** mode follows a more aggressive approach, allowing only *HI* criticality tasks to execute. The system is able to return to the normal execution mode, **N**, if an idle tick is detected and no task is pending for execution. The additional capacity available to the *HI* criticality partitions was calculated.

A response time analysis for the mixed criticality model was produced. Specifically, the analysis consists of calculating the response times of all partition tasks during the three modes of execution, **N**, **D1** and **D2**, and all mode changes, **N**→**D1**, **D1**→**D2**, **D1**→**N** and **D2**→**N**.

CHAPTER 6

Case Study: Mixed-criticality Engine Controller

One of the industrial use cases identified in Section 1.2 is supporting the integration of partitions of different criticality levels in a single physical ECU. This use case was realised in Chapter 5 by extending the scheduling model of Chapter 3 to support three levels of criticality using two modes of degradation. In the first degraded mode, **D1**, all event-triggered tasks are migrated from deferrable servers to their partitions' periodic serves, exploiting the additional available capacity that results from the reduced hypervisor overheads. In the second deferrable mode, **D2**, only partitions of HI criticality are allowed to execute.

In this chapter, we evaluate the scheduling approach of Chapter 5 by forming a case study using ECU application code that was provided by ETAS Ltd. The ECU application code used is the same as the one used in Chapter 4.

6.1 Server Parameter Selection

The task parameters and task-server mappings vary per criticality level, therefore the server period and capacity are calculated for each criticality level. Algorithm 1 details the method used for selecting server parameters. The approach followed focuses on having no gaps where server capacity is not available using the lowest possible replenishing period.

Algorithm 1: Server parameter selection for mixed criticality.

Input: ℓ - current criticality level
Data: *servers* - list of all servers
tasks - list of all tasks
MIN_P - minimum priority level
Result: The capacity and replenishment period of all servers are set using the minimum possible replenishment period.

```
1  Function SetServerParameters (ℓ)
2      foreach s ∈ servers do
3          periods ← [];
           /* Construct a list with the unique periods of all tasks
              serviced by server s                                    */
4          foreach t ∈ s.tasks[ℓ] do
5              if t.T(ℓ) ∉ periods then
6                  Append t.T(ℓ) to periods;
7          sort periods in ascending order;
8          done ← False;
           /* Periods are tried in ascending order, since they were
              sorted.                                                 */
9          foreach t' ∈ periods do
               /* The minimum required capacity given a period t' is given
                  as the total load on the server s.                  */
10             c' ← L(s, MIN_P, t', ℓ);
11             if c' < t' then
12                 s.T(ℓ) ← t';
13                 s.C(ℓ) ← c';
14                 done ← True;
15                 break loop;
16         if not done then
17             return FAIL;
18     return SUCCESS;
```

The parameters of a server s are selected by first constructing a list containing all the unique periods of the tasks serviced by s at criticality level ℓ. For each unique period t', starting from the shortest to the longest, the required server capacity, c', to service all the tasks within s is calculated as the load on the server given a window length t' at the minimum priority level, MIN_P. If the period and capacity values are valid, ie. $c' < t'$, they are assigned to the server.

6.2 Priority Assignment

Algorithm 2: Priority assignment algorithm.

Input: ℓ - current criticality level
Data: *servers* - list of all servers
ps - list of all periodic servers
hvtasks - list of all hypervisor tasks
dstasks - list of all tasks mapped to periodic servers

```
1  Function AssignPriorities(l)
2      p ← MAX_P;
3      sort hvtasks in ascending order by T(ℓ);
       /* Assign priorities to hypervisor tasks.                    */
4      foreach t ∈ hvtasks do
5          t.P(ℓ) ← p;
6          p ← decrement (p);
7      sort dstasks in ascending order by T(ℓ);
       /* Assign priorities to tasks using deferrable servers.      */
8      foreach t ∈ dstasks do
9          t.P(ℓ) ← p;
10         p ← decrement (p);
       /* For simplicity we assign tasks within periodic servers global
          priorities.                                               */
11     sort ps in ascending order by T(ℓ);
12     foreach s ∈ ps(ℓ) do
13         sort s.tasks(ℓ) in ascending order by T(ℓ);
14         foreach t ∈ s.tasks(ℓ) do
15             t.P(ℓ) ← p;
16             p ← decrement (p);
17     foreach s ∈ servers do
18         s.P(ℓ) ← ceilP (s.tasks(ℓ));     /* ceiling priority of tasks in s */
```

The priority assignment for all tasks and servers in the system follows Algorithm 2. The approach of Algorithm 2 assigns priorities based on a rate-monotonic approach, by assigning higher priorities to tasks with shorter periods. In the case where two tasks

have the same period, the one with the shortest WCET is assigned a higher priority. The algorithm follows a divide and conquer approach in order to produce a priority ordering that complies with the system model's priority space. Priorities are assigned from highest to lowest, given the current criticality level, ℓ.

A counter, p, is initialised at the highest priority level. The list of hypervisor tasks is sorted with respect to their periods and WCETs. The sorted list gives the priority ranking of the hypervisor tasks from highest to lowest. Iterating through the sorted hypervisor task list, priorities at the current criticality level (ℓ) are assigned the value of p. The priority level represented by p is decreased with each iteration, therefore all hypervisor tasks have unique consecutive priorities at the highest priority band.

Similar to hypervisor tasks, the list of tasks using deferrable servers at criticality level ℓ is sorted. All tasks in the sorted list are assigned unique priorities from high to low. The next step of priority assignment is tasks mapped on periodic servers. Unlike deferrable servers, periodic servers have no overlapping priorities. The list of periodic servers is therefore sorted with respect to their replenishment period and capacity at the current criticality level, ℓ.

For all periodic servers, starting from the highest priority to the lowest, tasks are assigned priorities. Specifically for a periodic server s the list of all its tasks at the current criticality level is sorted. The order of tasks in the sorted list is then used as a ranking for assigning priorities.

The last part of the priority assignment is setting the execution server priorities. As stated in Section 5.2, the priority level of an execution server at a criticality level ℓ is defined as the ceiling priority of all the tasks it services when the system executes at the current criticality level.

6.3 Sensitivity Analysis

Algorithm 3 details the algorithm used to perform sensitivity analysis to the mixed criticality model of Chapter 5. The purpose of the algorithm is to provide the maximum task WCET for each criticality level resulting in a schedulable system.

The input of the algorithm is a configuration file that contains all the base information required to construct a system definition at *LO* criticality. Specifically, the configuration file contains a list of application tasks. Each task is defined by its name,

Algorithm 3: Algorithm for scaling up the WCET of a system configuration.

Input: $conf$ - base system configuration file
Data: $STEP$ - the granularity level for scaling up the task WCETs
Result: A marginally schedulable system definition that follows the base configuration.

```
1  system ← parse (conf);
2  foreach ℓ ∈ {LO, MI, HI} do
3      min_scale ← 1;
4      max_scale ← 1/getUtilisation (system);
5      scale ← (max_scale − min_scale)/2;
6      schedulable ← false;
7      while (max_scale − min_scale) > 2 ∗ STEP do
8          system.setScaleWCET (scale, ℓ);
9          SetServerParameters (ℓ);
10         system.AssignPriorities (ℓ);
11         schedulable ←SetServerParameters (ℓ) ∧ isSchedulable (system, ℓ)
12         if schedulable then
13             min_scale ← scale;
14             scale ← (max_scale − min_scale)/2;
15         else
16             max_scale ← scale;
17             scale ← (max_scale − min_scale)/2;
18     if ¬schedulable then
19         scale ← scale − STEP;
20         system.setScaleWCET (scale, ℓ);
21         SetServerParameters (ℓ);
22         system.AssignPriorities (ℓ);
```

LO-criticality WCET (C^{cs1}, C, C^{cs2}), period, type (periodic or sporadic), criticality level (*LO*, *MI* or *HI*) and partition name. The application task parameters in conjunction with pre-set hypervisor overheads provides sufficient information to generate a representation of the resulting system.

The sensitivity analysis of Algorithm 3 follows a bisection-based approach to search for the maximum scaling factor that can be applied to the WCETs while keeping the system schedulable. The first step of the sensitivity analysis is parsing the configuration file and the initialisation of *min_scale*, *max_scale* and *scale*. This generates a system description with base parameters. *min_scale* then is initialised to 1 (ie. no scaling), *max_scale* is initialised to the scaling required to get a utilisation of 1 in the system, *scale* is defined as the mid-point between *min_scale* and *max_scale*. Additionally, the *schedulable* flag is initialised to $false$. For each criticality level ℓ, follow a bisection approach until

the difference between *min_scale* and *max_scale* is less than $2 * STEP$. Every iteration updates the *schedulable* flag, to indicate whether the system was schedulable or not. If the difference of *min_scale* and *max_scale* is less than $2 * STEP$ and the system was not schedulable during the last iteration, *scale* is decreased by $STEP$ and the scaling is applied to the system configuration. When the algorithm terminates, *scale* contains the maximum scaling factor and *system* contains the system configuration scaled up until just schedulable.

6.4 Taskset and Overhead Characteristics

This section details the characteristics of the case study formed using the application code provided by ETAS Ltd that was also used for Chapter 4.

6.4.1 Mixed-criticality Taskset

Table 6.1 lists the characteristics of the tasks used to evaluate the mixed-criticality extension of the system model. The WCET of the listed tasks is the maximum observed execution time, as obtained by the timing analysis performed in Section 4.2.2. These were used as a base for the *LO*-criticality WCET. The WCET values used to evaluate the proposed approach were obtained as described in Algorithm 3. The tasks retain their periodic/sporadic classification, since these properties are determined by the timing characteristics.

All tasks in the classified with respect to their criticality level. The classification was performed by an expert with respect to the functionality of the tasks. Specifically, tasks with functions that are key for the operation of an engine, such as fuel pump control and manifold pressure monitoring, are classified as HI criticality.

Tasks that are less susceptible to causing damage to the engine in case of failure, such as diagnostics or battery voltage monitoring, are classified as MI criticality. MI tasks are only dropped in **D2** mode, which is used as a last resort to allow the system to safely recover or reboot. Therefore, tasks with relaxed timing requirements from a functional prospective, such as coolant temperature monitoring (the rate of change of coolant temperature, assuming no sensor malfunction, is relatively slow), are also classified as MI criticality.

Task	WCET (*ns*)	Period (*ms*)	Criticality	Type
τ_0	518	100	HI	Periodic
τ_1	3641	10	HI	Periodic
τ_2	959	100	HI	Periodic
τ_3	189	10	MI	Periodic
τ_4	615	100	HI	Periodic
τ_5	222	1	HI	Sporadic
τ_6	447	1000	MI	Periodic
τ_7	125	100	MI	Periodic
τ_8	363	10	HI	Periodic
τ_9	364	10	HI	Periodic
τ_{10}	500	1	HI	Sporadic
τ_{11}	363	1	HI	Sporadic
τ_{12}	424	100	HI	Periodic
τ_{13}	363	10	MI	Periodic
τ_{14}	422	100	HI	Periodic
τ_{15}	1039	100	HI	Periodic
τ_{16}	249	100	HI	Periodic
τ_{17}	1088	100	HI	Periodic
τ_{18}	2538	100	MI	Periodic
τ_{19}	363	50	MI	Periodic
τ_{20}	340	100	MI	Periodic
τ_{21}	375	100	HI	Periodic
τ_{22}	344	1	HI	Sporadic
τ_{23}	1331	20	HI	Periodic
τ_{24}	483	10	MI	Periodic
τ_{25}	462	1	HI	Sporadic
τ_{26}	174	100	HI	Periodic
τ_{27}	204	10	HI	Periodic
τ_{28}	494	10	HI	Periodic
τ_{29}	508	10	HI	Periodic
τ_{30}	2373	20	HI	Periodic
τ_{31}	758	100	HI	Periodic

Table 6.1: Mixed-criticality application taskset characteristics.

6.4.2 Hypervisor Overheads

The identified hypervisor overheads for the mixed-criticality extension of the scheduling model are shown in Table 6.2. The interrupt forward and return overheads remain unchanged with the mixed criticality model. The cost of replenishing a server's capacity is also unchanged, however the replenishment period is set equal the corresponding server's period for each criticality level. In the case where no tasks are serviced by a server at a criticality level, the server capacity replenishment cost is set to 0.

Hypervisor Overhead	**WCET (*ns*)**
Forward interrupt	363
Return from interrupt	139
Replenish server capacity	553
Mode change	645

Table 6.2: Hypervisor overheads for the mixed-criticality model.

Changing criticality modes is a new source of hypervisor overheads that arises from the proposed mixed-criticality model. During a mode switch the hypervisor performs the necessary operations for migrating tasks between servers. The current system model assumes a single-core, therefore no task state information is required to be moved. The main operations performed during a mode change is a reconfiguration of the interrupt controller to support the new task priority levels and entry points. Similar to the other hypervisor overheads, the measurement was obtained via static analysis of a partial implementation.

6.5 Hypervisor System Configurations

In this section we present three different task configurations that were used for the evaluation of the mixed-criticality model of Chapter 5.

6.5.1 2-partition Configuration

The first configuration that was used for the evaluation was the two-partition setup that is shown in Table 6.3. In this configuration, the tasks are divided into two partitions with respect to their criticality level. Specifically p_0 is a *HI* criticality partition, whereas the criticality level of p_1 is *MI*. The highest priority band is occupied by the server replenishment tasks $p0\_ds\_rep$, $p0\_ps\_rep$ and $p1\_ps\_rep$. All asynrchronous tasks in

Task ID	LO		MI		HI	
	Server	Priority	Server	Priority	Server	Priority
p0_ds_rep	HV	1	HV	N/A	N/A	N/A
p0_ps_rep	HV	2	HV	0	HV	0
p1_ps_rep	HV	3	HV	1	N/A	N/A
τ_0	p0_ps	30	p0_ps	28	p0_ps	19
τ_1	p0_ps	22	p0_ps	20	p0_ps	11
τ_2	p0_ps	33	p0_ps	31	p0_ps	22
τ_3	p1_ps	9	p1_ps	2	N/A	N/A
τ_4	p0_ps	31	p0_ps	29	p0_ps	20
τ_5	p0_ds	4	p0_ps	10	p0_ps	1
τ_6	p1_ps	16	p1_ps	9	N/A	N/A
τ_7	p1_ps	13	p1_ps	6	N/A	N/A
τ_8	p0_ps	18	p0_ps	16	p0_ps	7
τ_9	p0_ps	19	p0_ps	17	p0_ps	8
τ_{10}	p0_ds	8	p0_ps	14	p0_ps	5
τ_{11}	p0_ds	6	p0_ps	12	p0_ps	3
τ_{12}	p0_ps	29	p0_ps	27	p0_ps	18
τ_{13}	p1_ps	10	p1_ps	3	N/A	N/A
τ_{14}	p0_ps	28	p0_ps	26	p0_ps	17
τ_{15}	p0_ps	34	p0_ps	32	p0_ps	23
τ_{16}	p0_ps	26	p0_ps	24	p0_ps	15
τ_{17}	p0_ps	35	p0_ps	33	p0_ps	24
τ_{18}	p1_ps	15	p1_ps	8	N/A	N/A
τ_{19}	p1_ps	12	p1_ps	5	N/A	N/A
τ_{20}	p1_ps	14	p1_ps	7	N/A	N/A
τ_{21}	p0_ps	27	p0_ps	25	p0_ps	16
τ_{22}	p0_ds	5	p0_ps	11	p0_ps	2
τ_{23}	p0_ps	23	p0_ps	21	p0_ps	12
τ_{24}	p1_ps	11	p1_ps	4	N/A	N/A
τ_{25}	p0_ds	7	p0_ps	13	p0_ps	4
τ_{26}	p0_ps	25	p0_ps	23	p0_ps	14
τ_{27}	p0_ps	17	p0_ps	15	p0_ps	6
τ_{28}	p0_ps	20	p0_ps	18	p0_ps	9
τ_{29}	p0_ps	21	p0_ps	19	p0_ps	10
τ_{30}	p0_ps	24	p0_ps	22	p0_ps	13
τ_{31}	p0_ps	32	p0_ps	30	p0_ps	21

Table 6.3: 2-partition system configuration.

the application used for the case study are classified as HI criticality. The deferrable server of $p1$ is not used, therefore the replenishment task $p1\_ds\_rep$ was omitted. From a scheduling perspective, replenishment tasks for deferrable servers are considered as LO criticality, since no deferrable servers are used in the degraded modes.

6.5.2 3-partition Configuration

The 3-partition configuration was composed to compare the performance of the model in the case where partitions were classified in terms of criticality and timing requirements. All periodic tasks are mapped to partitions p_0 and p_1. The sporadic tasks that were part of p_0 in the 2-partition configuration of Section 6.5.1 are now isolated in a separate partition, p_2. The criticality level of the tasks composing partition p_1 is MI, whereas p_0 and p_1 are HI criticality.

6.5.3 8-partition Configuration

In the 8-partition configuration tasks were partitioned with respect to their periods and criticality level, as shown in Table 6.5. p_0 is the only partition using a deferrable server, since it is composed of all the sporadic tasks in the system. The MI criticality partitions are p_2, p_4, p_6 and p_7. The partitions composed of HI criticality tasks are p_0, p_1, p_3 and p_5.

6.6 Experiment

In this section we provide an overview of the implementation used to perform the sensitivity analysis, as shown in Sections 6.1, 6.2 and 6.3. The hypervisor configurations of Section 6.5 were then used as input data to produce the results necessary to evaluate the mixed-criticality model.

6.6.1 Implementation

Figure 6.1 summarises the process followed to produce a system configuration that is marginally schedulable, given a base system specification. The base system specification is a file containing definitions for all application tasks in the system. Each task definition is consisted of the task name, WCET (C^{cs1}, C and C^{cs2}), period (T), criticality level (ℓ),

Task ID	LO		MI		HI	
	Server	Priority	Server	Priority	Server	Priority
p0_ps_rep	HV	4	HV	1	HV	1
p1_ps_rep	HV	5	HV	2	N/A	N/A
p2_ds_rep	HV	3	N/A	N/A	N/A	N/A
p2_ps_rep	N/A	N/A	HV	0	HV	0
τ_0	p0_ps	32	p0_ps	29	p0_ps	20
τ_1	p0_ps	24	p0_ps	21	p0_ps	12
τ_2	p0_ps	35	p0_ps	32	p0_ps	23
τ_3	p1_ps	11	p1_ps	8	N/A	N/A
τ_4	p0_ps	33	p0_ps	30	p0_ps	21
τ_5	p2_ds	6	p2_ps	3	p2_ps	2
τ_6	p1_ps	18	p1_ps	15	N/A	N/A
τ_7	p1_ps	15	p1_ps	12	N/A	N/A
τ_8	p0_ps	20	p0_ps	17	p0_ps	8
τ_9	p0_ps	21	p0_ps	18	p0_ps	9
τ_{10}	p2_ds	10	p2_ps	7	p2_ps	6
τ_{11}	p2_ds	8	p2_ps	5	p2_ps	4
τ_{12}	p0_ps	30	p0_ps	27	p0_ps	18
τ_{13}	p1_ps	12	p1_ps	9	N/A	0
τ_{14}	p0_ps	31	p0_ps	28	p0_ps	19
τ_{15}	p0_ps	36	p0_ps	33	p0_ps	24
τ_{16}	p0_ps	28	p0_ps	25	p0_ps	16
τ_{17}	p0_ps	37	p0_ps	34	p0_ps	25
τ_{18}	p1_ps	17	p1_ps	14	N/A	N/A
τ_{19}	p1_ps	14	p1_ps	11	N/A	N/A
τ_{20}	p1_ps	16	p1_ps	13	N/A	N/A
τ_{21}	p0_ps	29	p0_ps	26	p0_ps	17
τ_{22}	p2_ds	7	p2_ps	4	p2_ps	3
τ_{23}	p0_ps	25	p0_ps	22	p0_ps	13
τ_{24}	p1_ps	13	p1_ps	10	N/A	N/A
τ_{25}	p2_ds	9	p2_ps	6	p2_ps	5
τ_{26}	p0_ps	27	p0_ps	24	p0_ps	15
τ_{27}	p0_ps	19	p0_ps	16	p0_ps	7
τ_{28}	p0_ps	22	p0_ps	19	p0_ps	10
τ_{29}	p0_ps	23	p0_ps	20	p0_ps	11
τ_{30}	p0_ps	26	p0_ps	23	p0_ps	14
τ_{31}	p0_ps	34	p0_ps	31	p0_ps	22

Table 6.4: 3-partition system configuration.

Task ID	LO		MI		HI	
	Server	Priority	Server	Priority	Server	Priority
p0_ds_rep	HV	8	N/A	N/A	N/A	N/A
p0_ps_rep	N/A	N/A	HV	0	HV	0
p1_ps_rep	HV	9	HV	1	HV	1
p2_ps_rep	HV	10	HV	2	N/A	N/A
p3_ps_rep	HV	11	HV	3	HV	2
p4_ps_rep	HV	12	HV	4	N/A	N/A
p5_ps_rep	HV	13	HV	5	HV	3
p6_ps_rep	HV	14	HV	6	N/A	N/A
p7_ps_rep	HV	15	HV	7	N/A	N/A
τ_0	p5_ps	41	p5_ps	33	p5_ps	22
τ_1	p1_ps	29	p1_ps	21	p1_ps	14
τ_2	p5_ps	44	p5_ps	36	p5_ps	25
τ_3	p2_ps	21	p2_ps	13	N/A	N/A
τ_4	p5_ps	42	p5_ps	34	p5_ps	23
τ_5	p0_ds	16	p0_ps	8	p0_ps	4
τ_6	p7_ps	47	p7_ps	39	N/A	N/A
τ_7	p6_ps	33	p6_ps	25	N/A	N/A
τ_8	p1_ps	25	p1_ps	17	p1_ps	10
τ_9	p1_ps	26	p1_ps	18	p1_ps	11
τ_{10}	p0_ds	20	p0_ps	12	p0_ps	8
τ_{11}	p0_ds	18	p0_ps	10	p0_ps	6
τ_{12}	p5_ps	39	p5_ps	31	p5_ps	20
τ_{13}	p2_ps	22	p2_ps	14	N/A	N/A
τ_{14}	p5_ps	40	p5_ps	32	p5_ps	21
τ_{15}	p5_ps	45	p5_ps	37	p5_ps	26
τ_{16}	p5_ps	37	p5_ps	29	p5_ps	18
τ_{17}	p5_ps	46	p5_ps	38	p5_ps	27
τ_{18}	p6_ps	35	p6_ps	27	N/A	N/A
τ_{19}	p4_ps	32	p4_ps	24	N/A	N/A
τ_{20}	p6_ps	34	p6_ps	26	N/A	N/A
τ_{21}	p5_ps	38	p5_ps	30	p5_ps	19
τ_{22}	p0_ds	17	p0_ps	9	p0_ps	5
τ_{23}	p3_ps	30	p3_ps	22	p3_ps	15
τ_{24}	p2_ps	23	p2_ps	15	N/A	N/A
τ_{25}	p0_ds	19	p0_ps	11	p0_ps	7
τ_{26}	p5_ps	36	p5_ps	28	p5_ps	17
τ_{27}	p1_ps	24	p1_ps	16	p1_ps	9
τ_{28}	p1_ps	27	p1_ps	19	p1_ps	12
τ_{29}	p1_ps	28	p1_ps	20	p1_ps	13
τ_{30}	p3_ps	31	p3_ps	23	p3_ps	16
τ_{31}	p5_ps	43	p5_ps	35	p5_ps	24

Table 6.5: 8-partition system configuration.

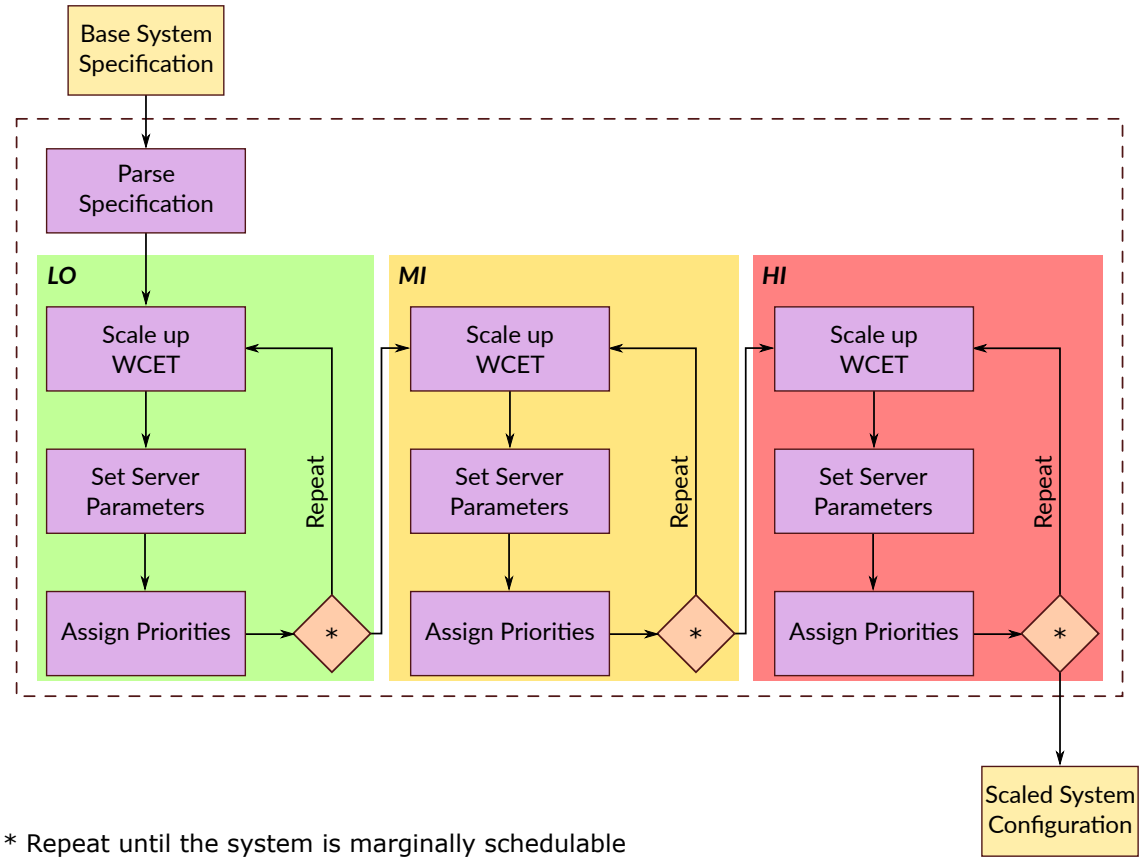


Figure 6.1: Implementation for the mixed-criticality model evaluation.

type (periodic or sporadic) and the identifier of the partition it is part of. All system overheads are provided as parameters.

For each application task a check is performed whether its partition has already been defined. If the partition is undefined, a new partition is added to the system with the provided identifier. Each partition has a deferrable server and a periodic server associated with it. A task entity is created using the specification of the current task, which is then mapped to the appropriate servers for each criticality level. For example, a HI criticality sporadic task is mapped on the deferrable server of its partition at criticality level LO and the periodic server at MI and HI. Whereas, a periodic task would be mapped on the periodic server of its partition for all criticality levels.

The taskset specification in conjunction with the system overhead parameters are used to compose a base system configuration. The configuration is then processed as described in Algorithm 3, using a `STEP` value of 0.001. This results in a system configuration that is marginally schedulable at all criticality levels. Hypervisor code is assumed

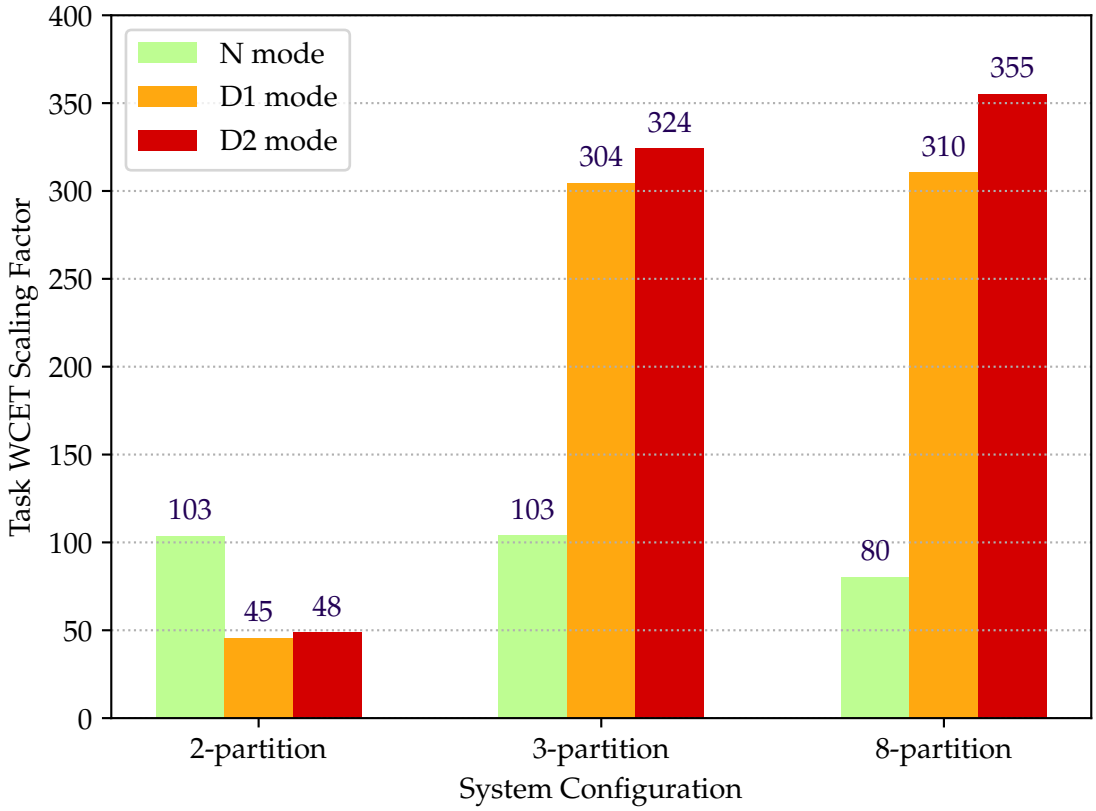


Figure 6.2: Application task WCET scaling with 2, 3 and 8-partition configurations.

to have highly predictable WCET. Therefore, the WCET for all operations performed by the hypervisor are not scaled up for the *MI* and *HI* criticality levels.

The implementation of the required analysis for the evaluation of the proposed mixed-criticality model was written using Oracle Java SE JDK 8. The analysis was run on a DELL XPS L501X laptop with an 8-core Intel i7 Q740 CPU, 6GB RAM running a 64-bit version of Microsoft Windows 10.

6.6.2 Results

In this section we discuss the results of the sensitivity analysis on the three identified system configurations.

WCET Scaling

Figure 6.2 shows the scaling that was achieved by each system configuration for all criticality levels. The 2-partition configuration while the system executes in **N** mode achieved a scaling factor of 103. At the degraded modes there is significant capacity

loss, which makes the proposed model ineffective for the 2-partition configuration. The reason for the poor performance in **D1** and **D2** is that the large variation of the temporal requirements of the application tasks of p_0.

As it was identified in Section 6.5.1, in the 2-partition configuration p_0 is consisted of both periodic and sporadic tasks. All sporadic tasks have a minimum interarrival time of $1ms$, whereas the periodic tasks of p_0 have periods of up to $100ms$. The server parameter selection method that was used was intended to eliminate gaps where no capacity was available. The capacity of the periodic server of p_0 at the degraded modes was set as the capacity required to service all its tasks within a $1ms$ interval. This resulted in significant waste of periodic server capacity, which caused the system to lose capacity after the mode switch to the first degraded mode, **D1**. Dropping p_1 in the second degraded mode, **D2**, provides a small amount of additional capacity, however it still performs poorly in comparison to the execution in **N** mode.

The 3-partition configuration achieves the same scaling as the 2-partition configuration while the system executes in **N** mode. After the mode switch to **D1** there is a significant increase of the WCET scaling, by a factor of 3. Assigning the sporadic tasks in a separate partition (p_2) eliminated the server parameter shortcoming of the 2-partition configuration. Specifically, in the degraded modes the sporadic tasks execute using the periodic server of p_2. All sporadic tasks in the system share the same minimum interarrival time of $1ms$. This allows for no wasted server capacity, therefore increasing the achieved scaling in the degraded modes. Switching to the second degraded mode, **D2**, there is a small increase in the scaling factor. The small increase is as expected, since the 95% of application task utilisation is used by HI criticality tasks.

The scaling achieved with the 8-partition configuration was 80, which is significantly lower than the 2 and 3-partition configurations. The lower scaling achieved for the 8-partition configuration was attributed to the large number of hypervisor tasks, which reside at the highest priority band in the system. Specifically, server capacity replenishment tasks are assigned strictly higher priorities than application tasks, therefore having a greater impact on the optimality of the priority assignment algorithm.

Another contributing factor to this is the context switching overheads of sporadic tasks. After the switch to the first degraded mode, **D1**, the achieved scaling is 310, which is the highest achieved in all three configurations. The use of solely periodic servers servicing tasks of the same period made the priority assignment algorithm very effective.

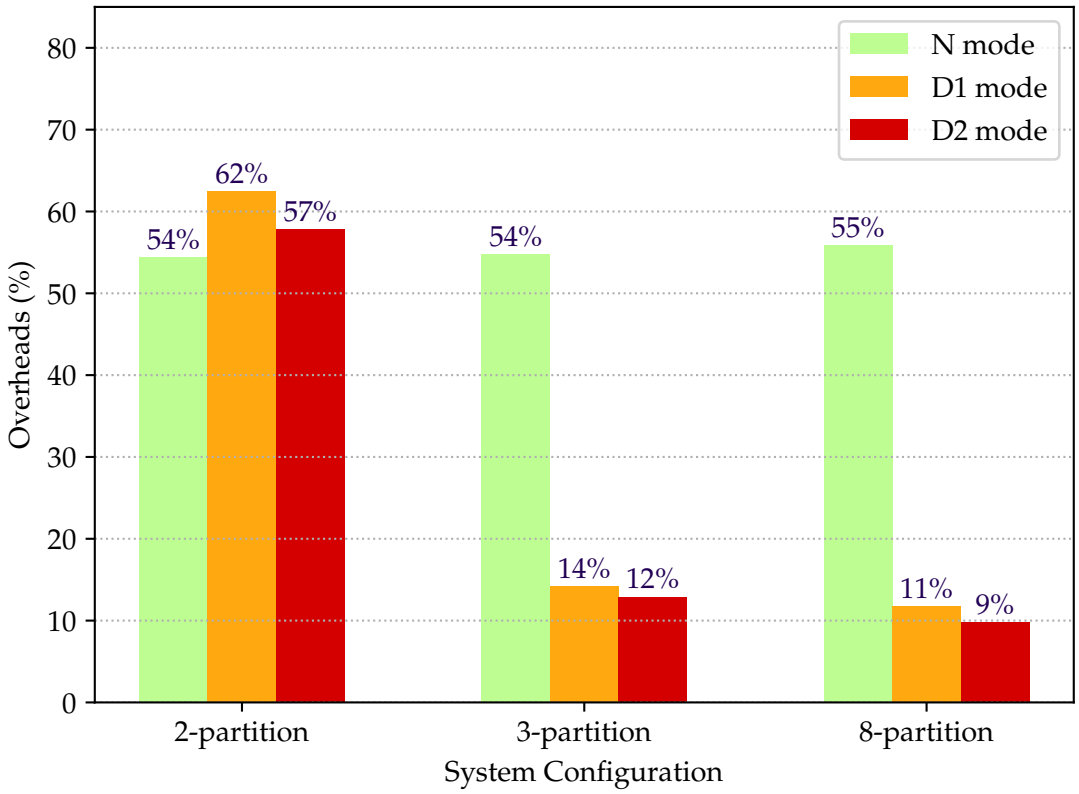


Figure 6.3: Ratio of hypervisor overheads to the system utilisation 2, 3 and 8-partition configurations.

Switching to **D2** results in more spare capacity than the 3-partition configuration, since additional capacity is freed due to the number of server replenishment tasks that are no longer in use, as shown in Table 6.5.

Hypervisor Overheads

Figure 6.3 shows the percentage of the system utilisation spent for hypervisor overheads for each system configuration; 54% of the utilisation of the 2 and 3-partition configurations is spent in hypervisor overheads. The majority of these overheads are attributed to handling sporadic tasks. The 8-partition configuration has 55% overheads, due to the additional number of execution servers.

The 2-partition configuration suffers from 62% and 57% overheads during the **D1** and **D2** modes respectively. Similarly to the poor scaling factor, the high overheads is also a result of the wasted server capacity due to the difference in the periods of the periodic and sporadic tasks of p_0. The capacity of the periodic server of p_0 in the

degraded modes has enough capacity to service all its tasks with each release. Most of this capacity is idled away, therefore increasing the overheads-to-utilisation ratio.

Separating the sporadic tasks in separate partitions resulted in a significant overhead decrease during the degraded modes, as shown in Figure 6.3 for the 3 and 8-partition configurations. The 8-partition configuration has slightly lower overheads with respect to the total utilisation than the 3-partition configuration since there is no wasted server capacity resulting from large periods variations of tasks within the same server.

6.7 Architectural Design Evaluation

Overall, the proposed mixed-criticality approach performs poorly in system configurations where the tasks within a single partition have significantly different temporal requirements. A high increase in the WCET scaling factor was observed when the sporadic tasks were migrated on a separate server instead of using the same periodic server as periodic tasks.

The poor performance of the 2-partition configuration is an indication where the use of polling servers may provide better performance under certain circumstances. The approach used for selecting server parameters was to find the smallest possible period for a server capacity that is equal to the load on the server over a window of length that is equal to the period. In the worst-case, this can result in a significant amount of unused capacity in cases where the serviced tasks have large variations in their periods. This is exacerbated in the average case.

The use of polling servers instead of periodic servers could therefore provide better overall performance when used by bound periodic tasks as polling servers relinquish their capacity when they are idle. The variation in the interarrival times of sporadic tasks can introduce instances where the server is idle, thus causing the server's capacity to be relinquished. Subsequent releases of sporadic tasks will therefore be blocked until the next replenishment of their server. Although the proposed mixed-criticality model takes advantage of the utilisation/performance trade-off, the use of polling servers to service sporadic tasks in degraded modes can provide lower performance than periodic servers. Therefore, the use of polling servers in degraded modes for servicing sporadic tasks can contradict the requirement for minimising the latency of sporadic tasks, as discussed in Section 3.1.

6.8 Summary

In this chapter the mixed criticality system model is evaluated by performing sensitivity analysis on three system configurations based on the application code provided by ETAS Ltd.

The methodology followed for performing the sensitivity analysis was detailed. Specifically, we present a server parameter selection approach that focuses on eliminating server gaps using short replenishment periods. A rate-monotonic based priority assignment was then defined, providing a systematic way of assigning priorities for all criticality levels, given the temporal characteristics of the tasks. The server parameter selection and priority assignment algorithms were used as part of the scaling up algorithm, which, given a base system configuration, scales up the WCET of all tasks in the system, producing a marginally schedulable system for each criticality level.

From the sensitivity analysis it was observed that the proposed approach was successful in providing additional capacity in the configurations where the sporadic tasks were isolated in separate partitions, during the execution in **D1** mode. Dropping MI criticality tasks in **D2** mode provided little improvement in terms of additional capacity, due to the taskset consisting of 95% HI criticality tasks. High variations in the task periods of one partition resulted in capacity loss due to high overheads and wasted server capacity.

CHAPTER 7

Case Study: Olympus Attitude and Orbital Control System

In this chapter we perform an experimental evaluation of the mixed-criticality extension of the proposed model, which was introduced in Chapter 5. The case study performed in this chapter was inspired from the real-time characteristics of the Olympus Attitude and Orbital Control System (AOCS) taskset, as presented in the paper by Burns et al. [22]. The focus of the experiments is the investigation of the response times achieved by the proposed mixed-criticality model when the simulated system experiences average-case behaviour in the three execution modes.

AOCS is a subsystem of the Olympus experimental communication satellite, which was in service from July 1989 [22] until August 1993 [39]. The AOCS is responsible for maintaining the satellite's position and orientation in the geostationary ring. The primary functionality of the AOCS is implemented by the CONTROLLER object, which is functionally decomposed to the following objects:

- **RECEIVE FROM BUS -** Used for accessing shared data from the bus.
- **CONTROL LAW -** Responsible for maintaining the satellite's position.
- **SENSOR -** Provides the CONTROL LAW object with required sensor information.
- **ACTUATORS -** Provides access to the satellite's actuators.

7.1 Experiment Setup

This section describes the experiment setup used to evaluate the proposed mixed-criticality model of Chapter 5. We first list the tasks that consist the Olympus AOCS taskset using the information from the case study by Burns et al. [22]. The information from the case study is also used to obtain representative hypervisor overheads for a hardware platform with a high level of virtualisation support. We then present two alternative partitioning configurations, which are used to investigate the impact of partitioning in the observed latency and response times.

7.1.1 AOCS Taskset and Hypervisor Overheads

The case study by Burns et al. [22] provides the real-time requirements of the tasks of the AOCS. The taskset is consisted of 10 periodic tasks and 7 sporadic tasks, as listed in Table 7.1. Each task is defined by its period, WCET, deadline and offset. The level of criticality of each task was assigned with respect to the functionality provided by each task. Specifically, task C5 (CONTROL_LAW) was treated as the primary task of the AOCS, therefore it is treated as a HI criticality task. Based on the object descriptions in [22], tasks that are directly related to C5 were also treated as HI criticality. Task $S2$ is identified as a soft real-time task, therefore it was assigned a LO criticality. The rest of the tasks in the system were assigned MI criticality. We then present the approach taken to simulate average-case behaviour in the simulated system.

For the hypervisor overheads, we assume that the target hardware platform has a high level of virtualisation support, which results in hpervisor overheads that are multiples of the time required to service an interrupt, as per the timing analysis performed by Burns et al. [22]. The hypervisor overheads that are used in this case study are especially selected to be minimal as the automotive case study of Chapters 4 and 6 investigate a taskset where the application has a relatively low utilisation compared to the hypervisor overheads.

Investigating the behaviour of the proposed model with minimal hypervisor overheads can provide additional insights with respect to the design choices of the architecture proposed in Chapter 5. The resulting hypervisor overheads are summarised in Table 7.2.

ID	Name	T	WCET	Criticality	D	Offset
Periodic Tasks						
C1	READ_BUS_IP	10	1.7639E+00	HI	10	0
C2	REAL_TIME_CLOCK	50	2.8248E-01	HI	90	0
C3	COMMAND_ACTUATORS	200	2.1265E+00	HI	140	50
C4	REQUEST_WHEEL_SPEEDS	200	1.4257E+00	MI	22	0
C5	CONTROL_LAW	200	5.2846E+01	HI	200	50
C6	PROCESS_DSS_DATA	1000	5.1562E+00	MI	400	200
C7	REQUEST_DSS_DATA	200	1.4257E+00	MI	17	150
C8	CALIBRATE_GYRO	1000	6.9140E+00	HI	900	200
C9	PROCESS_IRES_DATA	100	8.2206E+00	HI	50	500
C10	REQUEST_IRES_DATA	100	1.4257E+00	MI	24	0
Sporadic Tasks						
S1	TELEMETRY_RESPONSE	62.5	3.1930E+00	HI	30	0
S2	TELECOMMANDS	187	2.5006E+00	LO	187	0
S3	READ_YAW_GYRO	100	4.0749E+00	HI	50	0
S4	MESSAGES_HERE	50	1.3424E+00	MI	50	0
S5	TM_HERE	62.5	9.9160E-02	MI	62.5	0
S6	ZI_HERE	100	9.9160E-02	MI	100	0
S7	TC_HERE	187	9.9160E-02	MI	187	0

Note: The unit of measure of time used in the table is milliseconds (ms).

Table 7.1: AOCS Taskset Real-time Characteristics.

Hypervisor Overhead	WCET (ms)
Forward interrupt	0.0054
Return from interrupt	0.0054
Replenish server capacity	0.0108
Mode change	0.0108

Table 7.2: Hypervisor overheads for the mixed-criticality model.

7.1.2 Average-Case Behaviour Simulation

The primary requirement of the experiment setup is the investigation of the system's behaviour using average case manifestations of all task releases. This requires the introduction of randomisation in two areas in the system: sporadic task interarrival times and the task execution times. The necessary information for creating a probabilistic model of the average case behaviour of the system is not available in the original case study, therefore all randomisation was performed using examples from the literature. The simulations were run for the length of one hyper-period and repeated five times.

In Table 7.1, all tasks are associated with a period. In the case of sporadic tasks, the period is treated as a minimum interarrival time. To obtain the interarrival time between the each release of a given sporadic tasks, we employ the analysis performed

by Maxim et al. [75]. Specifically, we use a Weibull distribution ($k = 2$, $\lambda = 1$) to obtain a coefficient for each release of a given sporadic task. The coefficient is then multiplied with the minimum interarrival time resulting in the actual interarrival time. Coefficients with a value of ≤ 1 are set to 1.

The actual execution time of each task release is calculated using a Gumbel distribution ($\mu = 0.6$, $\sigma = 0.125$). Similar to the interarrival times, the actual execution time of each task release is calculated by multiplying the worst-case execution time of the given task with a value from the distribution. If the coefficient obtained from the Gumbel distribution is ≥ 1, it is set to 1.

7.1.3 Partitioning

From the analytical results on both the single-criticality model of Chapter 4 and the mixed-criticality model of Chapter 6, partitioning plays an important role in the performance of the proposed model. The results suggested that partitioning tasks primarily based on their periods rather than just the sporadic/periodic nature of tasks can significantly improve the schedulability of the system despite the increase of overheads that follows many-partition configurations.

The case study performed in this chapter examines a system with minimal hypervisor overheads. This can provide a clearer view on the impact of partitioning, since the effect of the hypervisor overheads will be significantly lower. We examine two system configurations: a three-partition configuration and a four-partition configuration. The partitioning for the two configurations is listed in Table 7.3. Tasks *C1* and *C7* have relatively short deadlines. To accommodate the requirement for a short response time of those tasks, they were assigned to the deferrable servers of their respective partitions instead of the periodic servers during the execution in **N** mode.

3-partition Configuration

As stated in Section 5.1, all tasks within a partition are constrained to have the same level of criticality. Therefore, the minimum number of partitions for the tasks in Table 7.1 is achieved by grouping all tasks of the same criticality level into a single partition. This results into three partitions.

3-partition Configuration			4-partition Configuration		
Partition	**Task**	**Period**	**Partition**	**Task**	**Period**
p0, L=LO	S2	187	p0, L=LO	S2	187
p1, L=MI	C10	100	p1, L=MI	C10	100
	C3	200		C3	200
	C4	200		C4	200
	C6	1000		C6	1000
	C7	200		C7	200
	S4	50		S4	50
	S5	62.5		S5	62.5
	S6	100		S6	100
	S7	187		S7	187
p2, L=HI	C1	10	p2, L=HI	C1	10
	C2	50	p3, L=HI	C2	50
	C5	200		C5	200
	C8	1000		C8	1000
	C9	100		C9	100
	S1	62.5		S1	62.5
	S3	100		S3	100
HV	p0_DS_rep	187	HV	p0_DS_rep	187
	p1_DS_rep	50		p1_DS_rep	50
	p1_PS_rep	100		p1_PS_rep	100
	p2_DS_rep	62.5		p2_DS_rep	62.5
	p2_PS_rep	100		p2_PS_rep	100
	N/A	N/A		p3_PS_rep	10

Table 7.3: Olympus AOCS partition configurations.

4-partition Configuration

From Table 7.1, the *HI* criticality periodic task C1 has the shortest period in the system. In the 4-partition configuration we introduce an additional partition that services only C1. An alternative partitioning scheme would the introduction of additional partitioning for tasks with significantly different temporal requirements in comparison to the other tasks in the same partition.

7.2 Results

In this section we discuss the results of the simulation runs that were described in Section 7.1. Boxplots of the response times of each task in each configuration is shown in Appendix B. Tables 7.4 and 7.5 summarise the observed response times of of each task the 3-partition and 4-partition configurations, respectively, for each execution mode (**N**, **D1**, **D2**). Descriptive statistics for the observed latency values in the system are shown

Task	N - LO			D1 - MI			D2 - HI		
ID	MAX	Mean	Variance	MAX	Mean	Variance	MAX	Mean	Variance
C1	4.262E-02	1.680E-03	2.444E-05	**1.111E+01**	2.681E+00	1.854E+01	1.620E-02	1.223E-03	1.377E-05
C2	2.332E+01	7.617E+00	5.793E+01	2.852E+01	1.870E+01	2.265E+01	1.628E+01	5.909E+00	2.249E+01
C3	2.748E+00	4.005E-01	2.282E-01	1.765E+01	1.454E+01	1.639E+00	4.047E+00	1.797E+00	6.944E-01
C4	7.281E+00	3.737E-01	1.177E+00	2.700E-02	1.985E-02	6.092E-06	*N/A*	*N/A*	*N/A*
C5	4.096E+01	2.288E+01	2.005E+02	6.261E+01	4.018E+01	1.320E+02	2.764E+01	1.152E+01	4.078E+01
C6	6.055E+00	2.785E+00	1.725E+00	3.334E+00	2.886E+00	8.075E-02	*N/A*	*N/A*	*N/A*
C7	3.453E-01	1.595E-02	4.532E-03	1.620E-02	1.283E-02	3.640E-06	*N/A*	*N/A*	*N/A*
C8	1.843E+01	1.488E+01	8.682E+00	2.685E+01	2.240E+01	8.023E+00	1.344E+01	1.041E+01	5.732E+00
C9	1.592E+01	9.768E+00	2.142E+00	1.789E+01	1.476E+01	2.328E+00	5.350E+00	1.646E+00	1.140E+00
C10	8.509E+00	9.333E-01	1.474E+00	1.443E+00	4.639E-01	2.174E-01	*N/A*	*N/A*	*N/A*
S1	1.735E+00	1.574E-01	1.735E-01	1.433E+01	3.684E+00	3.206E+01	1.777E+00	3.502E-01	3.199E-01
S2	5.714E+00	3.859E-01	1.169E+00	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
S3	4.094E+00	6.002E-01	9.685E-01	2.600E+01	2.041E+01	5.422E+00	1.287E+01	7.438E+00	6.024E+00
S4	2.748E+00	2.152E-01	2.915E-01	2.647E+00	9.498E-01	4.815E-01	*N/A*	*N/A*	*N/A*
S5	5.510E+00	1.554E-01	4.721E-01	4.068E+01	2.065E+01	1.962E+02	*N/A*	*N/A*	*N/A*
S6	5.573E+00	1.818E-01	6.971E-01	3.745E+00	2.426E+00	3.376E-01	*N/A*	*N/A*	*N/A*
S7	5.660E+00	3.858E-01	1.234E+00	4.096E+01	1.681E+01	1.812E+02	*N/A*	*N/A*	*N/A*

Table 7.4: Descriptive statistics of the observed latency Olympus AOCS tasks under the 3-partition configuration.

Task	N - LO			D1 - MI			D2 - HI		
ID	MAX	Mean	Variance	MAX	Mean	Variance	MAX	Mean	Variance
C1	5.709E+00	1.112E-02	6.091E-02	2.749E-02	1.064E-02	1.131E-05	2.375E-02	9.917E-03	5.270E-06
C2	1.823E+01	6.021E+00	3.675E+01	2.737E+01	1.779E+01	1.086E+01	1.184E+01	4.175E+00	6.434E+00
C3	3.060E+00	3.166E-01	3.415E-01	1.720E+01	1.529E+01	5.457E-01	4.697E+00	2.147E+00	3.811E-01
C4	3.225E+00	1.036E-01	2.143E-01	1.800E+00	1.793E+00	1.060E-05	*N/A*	*N/A*	*N/A*
C5	3.564E+01	5.872E+00	9.613E+01	4.629E+01	2.320E+01	6.817E+01	2.006E+01	7.518E+00	9.682E+00
C6	4.977E+00	1.636E+00	1.552E+00	3.909E+00	3.287E+00	1.058E-01	*N/A*	*N/A*	*N/A*
C7	5.204E-01	2.590E-02	1.072E-02	1.796E+00	1.788E+00	7.322E-06	*N/A*	*N/A*	*N/A*
C8	1.547E+01	1.212E+01	5.643E+00	2.346E+01	2.165E+01	5.363E+00	9.472E+00	5.699E+00	3.488E+00
C9	1.254E+01	8.965E+00	8.452E-01	1.929E+01	1.531E+01	9.981E-01	4.472E+00	2.025E+00	2.762E-01
C10	4.308E+00	3.993E-01	3.131E-01	3.217E+00	2.058E+00	1.145E-01	*N/A*	*N/A*	*N/A*
S1	9.439E-01	8.803E-02	4.854E-02	1.466E+01	4.200E+00	3.741E+01	1.793E+00	4.570E-01	6.101E-01
S2	2.694E+00	1.724E-01	2.687E-01	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
S3	1.943E+00	1.002E-01	1.304E-01	2.565E+01	1.871E+01	5.825E+00	1.096E+01	4.935E+00	2.719E+00
S4	1.651E+00	7.115E-02	6.208E-02	3.891E+00	2.289E+00	1.842E-01	*N/A*	*N/A*	*N/A*
S5	2.574E+00	1.061E-01	1.553E-01	2.905E+01	1.145E+01	1.234E+02	*N/A*	*N/A*	*N/A*
S6	2.615E+00	8.599E-02	1.963E-01	4.438E+00	3.046E+00	1.893E-01	*N/A*	*N/A*	*N/A*
S7	2.651E+00	8.623E-02	2.266E-01	3.815E+01	1.458E+01	1.592E+02	*N/A*	*N/A*	*N/A*

Table 7.5: Descriptive statistics of the observed latency Olympus AOCS tasks under the 4-partition configuration.

Configuration	Exec. Mode	Max	Mean	Variance	Skewness	Kurtosis
3-partition	**N (LO)**	40.96	2.56	38.86	3.20	11.56
4-partition		35.64	1.65	16.68	3.12	12.88
3-partition	**D1 (MI)**	62.61	7.59	109.35	1.65	2.71
4-partition		46.29	5.65	69.49	1.40	1.10
3-partition	**D2 (HI)**	27.64	1.75	13.77	2.69	8.28
4-partition		20.06	1.29	5.51	2.24	5.74

Table 7.6: Descriptive statistics of the observed latency for each mode of execution.

in Table 7.6.

During the execution of the 3-partition configuration in **N** mode, no deadline misses were observed. The maximum observed latency was $40.96ms$ by task C5, which was assigned the second lowest priority in the system. The mean of the observed latencies was $\mu = 2.56ms$ with a variance of $\sigma^2 = 38.86$. The kurtosis of the observed latency values was 11.56. In combination with the skewness (3.20), which indicates a high number of outliers. The 4-partition configuration in **N** execution mode also experiences no deadline misses. The maximum observed latency is $35.64ms$ with mean $\mu = 1.65ms$ and variance $\sigma^2 = 16.68$. The skewness (3.12) and kurtosis (12.88) of the observed latencies are close to the corresponding 3-partition ones, indicating a similar shape in the distribution. Using a significance level of $\alpha = 0.05$, analysis of the latency times indicated that the 3-partition configuration exhibited significantly higher latency compared to the 4-partition configuration.

After a **N**→**D1** mode change, S2, which is the only *LO* criticality task in the system is dropped, whereas the remaining sporadic tasks are migrated to their corresponding partitions' periodic servers. The overall observed latencies in both configurations are increased. This is expected as the motivation of the mixed criticality system model of Chapter 5 and the results of the case study in Chapter 6, is the exploitation of the schedulability/performance trade-off.

The highest observed latency in the 3-partition configuration is $62.61ms$, with a mean of $\mu = 7.59ms$ and variance $\sigma^2 = 109.35$. The skewness (1.65) and kurtosis (2.71) during the execution **D1** mode indicating a significantly lower number of outliers compared to **N**. The 4-partition configuration exhibited a maximum latency of 46.29, mean $\mu = 5.65$ and variance $\sigma^2 = 69.49$. The skewness (1.40) and kurtosis (1.10) of the 4-partition configuration are lower than the respective 3-partition values. Specifically, the lower kurtosis in the 4-partition configuration indicates a lower number of outliers.

During the execution of the 3-partition configuration, the only task that exhibited deadline misses was C1. The deadline misses of C1 are attributed to the temporal requirements of C1 compared to the rest of the tasks of the same partition. The 4-partition configuration exhibited no deadline misses. This is consistent with the findings of Section 6.6.2, where it was indicated that a high level of variation in periods of tasks sharing a single execution server can have a negative effect in the optimality of the server parameters and priority assignment.

The transition **D1**→**D2** causes all tasks with criticality level of MI and LO to be abandoned. No deadline misses were observed in either configuration (3-partition and 4-partition). The maximum observed latency in the 3-partition configuration is $27.64ms$, whereas in the case of the 4-partition configuration the maximum observed latency is $20.06ms$. The mean and variance of the observed latency in the 3-partition configuration was $\mu = 1.74ms$ and $\sigma^2 = 13.77$, whereas in the 4-partition configuration, $\mu = 1.29ms$ and $\sigma^2 = 5.51$.

7.3 Architectural Design Evaluation

The response time analysis of the original case study [22] indicated that at the worst case, all hard real-time tasks would meet their deadlines. $S2$, which is a soft-real time task would miss its deadline in the worst case. Response time analysis on the Olympus AOCS taskset using the model proposed in Chapter 5, indicated that the system is not schedulable in either configuration. The release jitter experienced from unbound tasks and the deferrable servers played a significant role in the interference when the system executes in the critical instance.

The proposed model of Chapter 5 was developed to take advantage of the trade-off between low latency and schedulability. In the experiment results of Section 7.2 it was shown that both configuration experienced overall lower response times during the execution in **N** mode compared to **D1**. Although the observed response times during the execution in **N** mode were lower, there were occasional spikes, which are represented by the calculated skewness and kurtosis values. The use of periodic servers in **D1** mode resulted in higher but more predictable response times. This was expected due to the nature of deferrable servers. This correlates with the rationale behind the development of the mixed-criticality extension to the system model (Chapter 5), as performance (ie.

low latency) is traded for more predictable response times.

As it was identified in Section 6.7, and the simulation runs performed in this Chapter using the 3-partition configuration, in cases where tasks with significantly different temporal requirements share the same periodic server, there is some degradation due to the relatively long replenishment period of the server. This raises the question whether polling servers would be a more appropriate option in place of periodic servers. Given the real-time characteristics of the Olympus AOCS taskset, and in particular the number of unbound tasks in the systems, the use of polling servers would potentially have a negative impact on the overall performance of the system. Specifically, given the taskset of this case study, the likelihood of a task waiting on the replenishment of its execution server's capacity is higher.

As was also observed in Chapters 4 and 6, temporal isolation, which is one of the key requirements identified in Section 3.1, is sufficiently enforced by the use of execution servers. Specifically, limiting the capacity of the execution servers provides an upper boundary on the expected interference from tasks that execute in different servers. Additionally, with the abandonment of *LO* criticality tasks in **D1** and *MI* criticality tasks in **D2**, we limit the impact of lower criticality tasks on higher criticality tasks.

Abandoning the execution of *MI* and *LO* criticality tasks was shown to be sufficient in both configurations for providing additional capacity to service *HI* criticality tasks. Execution in **D2** exhibits the shortest response times in both configurations, ensuring that no deadlines are missed. This further reinforces the rationale behind **D2** mode (Section 5.3.3), where if increasing the utilisation bound by the use of periodic servers is not sufficient, dropping lower criticality tasks can provide additional capacity to service all *HI* criticality tasks.

7.4 Summary

In this chapter we form a case study inspired from the Olympus Attitude and Orbital Control System (AOCS) [22]. The taskset of the Olympus AOCS is first analysed with respect to the functionality provided by each task. The tasks were then classified into the three levels of criticality, as supported by the model described in Chapter 5. The experimental setup was tailored to investigate the impact of partitioning in the system and the behaviour of the system in the average case, assuming a hardware platform

with a high level of virtualisation support.

Two partitioning approaches were considered: 1 partition per criticality level (3-partitions), further partitioning based on real-time characteristics of tasks (4-partitions). Simulation results of the average case behaviour indicated that the 3-partition configuration exhibited a higher level of skewness of the observed latencies compared to the 4-partition configuration. The 3-partition configuration exhibited deadline misses of a low priority task while executing in **D1** mode. This aligns with the findings of the sensitivity analysis of Chapter 6, where it was indicated that in cases where tasks with significantly different temporal characteristics can have a negative impact in the performance of the proposed model.

The 4-partition configuration, which was constructed based on temporal characteristics, as well as the level of criticality of each task, outperformed the 3-partition configuration in all modes of execution. The simulation results show that the mixed-criticality model takes advantage of the performance-predictability trade-off during the execution in the degraded modes (**D1** and **D2**), as the number of outliers in the observed latencies is significantly lower compared to the execution in the **N** mode.

CHAPTER 8

Conclusion

This thesis is written in partial fulfilment of the requirements for the degree of Doctor of Engineering, EngD, therefore the undertaken research is highly motivated by the interests of the industrial sponsor of this project, ETAS Ltd. The work towards this thesis was evaluated by forming a case study using real ECU application code that was provided by ETAS Ltd.

In this chapter we discuss the contributions of the work done within this thesis in terms by addressing the research hypothesis. We then identify limitations and possibilities for future work to further investigate and build on the contributions of the work done towards this thesis.

8.1 Thesis Overview

Chapter 1 starts with a brief introduction on the challenges faced in the automotive domain. It then sets an industrial context based on the interests of the sponsoring organisation, which served as a motivation towards the undertaken research. The introduction closes by stating the thesis hypothesis, and providing an outline of the thesis.

Chapter 2 provides a review of the relevant literature. We briefly visit topics on timing analysis and real-time scheduling. A review on relevant work on mixed-criticality and hierarchical scheduling is provided. The focus of the review then shifts to an industry-oriented approach by reviewing existing hypervisor systems. The interests of the industrial sponsor are then visited, reviewing the current state of their work and

identifying the research gap that the work of this thesis focuses on.

Chapter 3 describes a proposed architectural design and a scheduling approach for a single-core hypervisor system that focuses on providing low latency for event-triggered application tasks. The proposed model incorporates the scheduling and hypervisor overheads. A response time analysis is provided for the proposed model.

Chapter 4 evaluates the approach proposed in Chapter 3 by means of a case study using real ECU code that was provided by ETAS Ltd. A short description of the hardware platform used is provided. The application task parameters are calculated by means of measurement-based analysis, whereas hypervisor overheads were calculated using static analysis. The evaluation was performed via simulation.

Chapter 5 extends the model proposed in Chapter 3, allowing for three criticality levels. The definition of tasks and servers is extended with vectors to support parameters for each criticality level. The execution modes and the transitions between them are described and a response time analysis is produced.

Chapter 6 evaluates the mixed-criticality model of Chapter 5 by forming a case study inspired from the automotive industry. The algorithms used for selecting server parameters, assigning task priorities and performing sensitivity analysis are listed. The sensitivity analysis was applied to three system configurations using the timing information of the application tasks from the timing analysis of Section 4.2.2.

Chapter 7 consists of a case study inspired from the Olympus AOCS. The results of the case study were obtained by simulating the average-case behaviour of the system. Statistical analysis was used to investigate the performance of the proposed model.

8.2 Summary of Contributions

In this section we summarise the contributions of the work done towards exploring the research hypothesis. The research hypothesis addressed by this thesis is:

> "The hypothesis of this research project is that virtualisation can be used in the automotive industry to combine the functionality of more than one ECUs on a single hardware platform, while being able to make guarantees about the real-time properties of the system. " (Section 1.3)

From the research hypothesis two key requirements are elicited:

(R1) The research is relevant and fills the identified research gap in the automotive industry, as articulated by ETAS Ltd.

(R2) The scheduling models developed by this research must be analysable to make guarantees about the real-time properties of the system.

The requirements from the research hypothesis are satisfied by the following contributions.

8.2.1 Development of a Hypervisor Scheduling Model

In Chapter 3 we present a scheduling approach for a hypervisor system. Traditional hypervisor scheduling systems assume no visibility of the partitions' execution **(R1)**. The proposed approach focuses on minimising the latency of event-driven tasks **(R1)**. The use of execution servers ensures temporal isolation between partitions **(R2)**. The proposed model allows for the incorporation of implementation overheads, making the response time analysis closer to the system's actual execution **(R1, R2)**. Empirical results show that although the schedulability analysis is pessimistic, timing faults are contained and not propagated, making the proposed model robust and reliable.

8.2.2 Mixed-criticality Model

Chapter 5 extends the model of Chapter 3 to support multiple levels of criticality **(R1)**. With the first degraded mode **D1**, the proposed mixed-criticality model takes advantage of the trade-off between schedulability and low latency for event-triggered tasks, increasing the capacity available to partitions. The mixed-criticality model was evaluated by means of sensitivity analysis using parameters derived from real ECU application tasks **(R1, R2)**. The analysis showed that partitions composed of tasks with similar temporal requirements are able to increase the capacity available by a factor of 3 **(R2)**. The additional capacity is made available without requiring the abandonment of any partition tasks, which makes it a novel approach. The second degraded mode, **D2**, provides additional capacity based on the criticality levels of the composing partitions **(R2)**.

8.3 Limitations and Future Work

In this section we identify the limitations of the work done and propose possible areas of future work.

8.3.1 Dependency of MC Model on Task Temporal Characteristics

The evaluation of the mixed-criticality model showed that the capacity after switching to the degraded modes is highly dependent on the temporal requirements of the tasks within each partition. Specifically, in the case where tasks with significant variability in their temporal requirements, capacity loss was observed. A possible way to remedy that is to separate the application tasks into partitions with respect to their temporal requirements. This is a potentially expensive job, since it is highly dependent on the availability of tooling that allows the reconfiguration of the AUTOSAR BSW. An alternative approach which is to be explored as future work is the conversion of deferrable servers into periodic servers during the first degraded mode. A third option that can be explored is the investigation of using polling servers in place of periodic servers for periodic tasks based on the timing characteristics of the taskset. Another consideration which was identified in Section 7 is taking into consideration the release time offset of tasks in the taskset, further guiding the development of the system model.

8.3.2 Support for Multi-core

The proposed models currently support single-core hardware platforms. Section 1.2 identifies the support for multi-core platforms as a use case for hypervisor systems as a means of reducing the porting costs of older ECU code. Supporting multi-core platforms opens interesting research questions, such as the viability of task/partition migration and inter-core communications.

8.3.3 Variability in Hardware

The proposed scheduling models for single and mixed-criticality systems were designed to be able to take into account implementation overheads. The empirical results were obtained using execution time and overhead parameters based on a ARM1176JZF-S processor. Further investigation on the impact of processor features, such as virtualisation support, on the hypervisor overheads. The hypervisor overheads were a key factor in the pessimism of the response-time analysis of Chapter 4 and the capacity loss of the 2-partition system configuration in Chapter 6.

8.3.4 Partition and Task Dependencies

The only support for dependencies between tasks and partitions in the proposed model is the use of resources. In Section 1.1 it was identified that one of the challenges in the automotive industry is the complexity due to cross-ECU communication. The lack of support for interaction between tasks and partitions is therefore a limitation of the model. Extending the model to support these interactions is a possible avenue for future work.

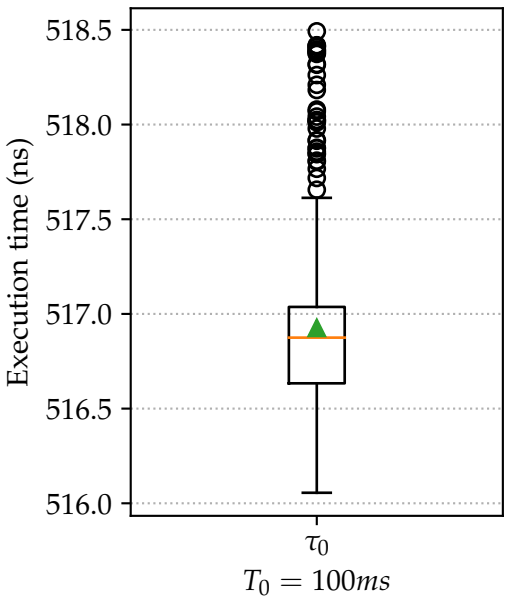
8.4 Closing Remarks

Overall, although there are significant topics of future work to be addressed, the work of this thesis has demonstrated that a flexible approach to virtualisation can address many of the current requirements of automotive software.

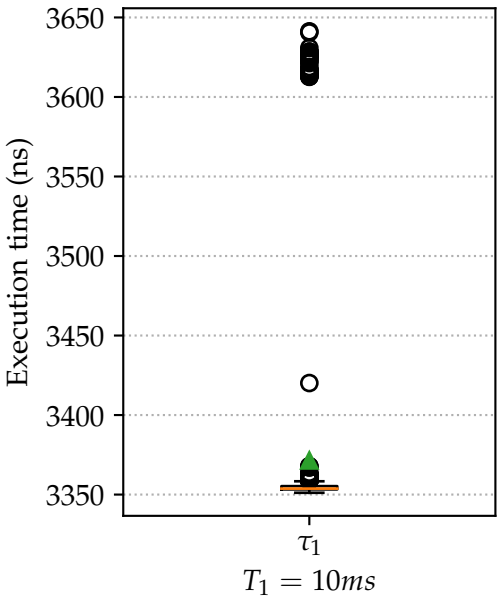
APPENDIX A

Application Task Execution Times

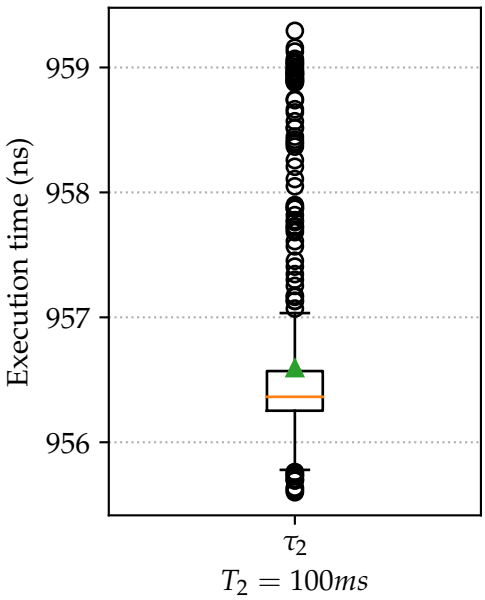
This appendix contains boxplots summarising the observed execution times during the timing analysis performed in Section 4.2.2.



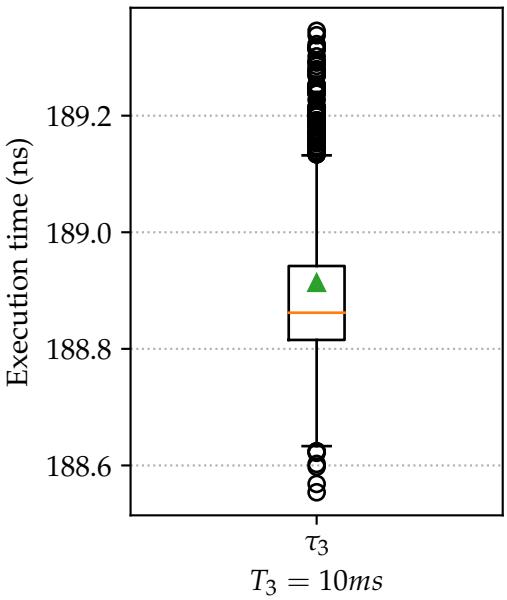
(a) τ_0 execution measurements.



(b) τ_1 execution measurements.

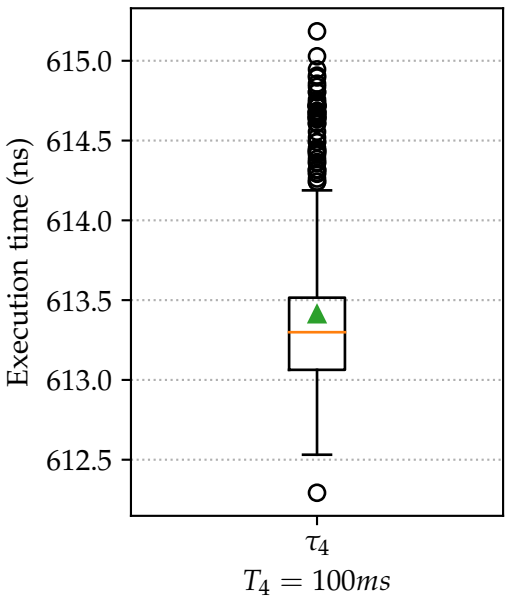


(c) τ_2 execution measurements.

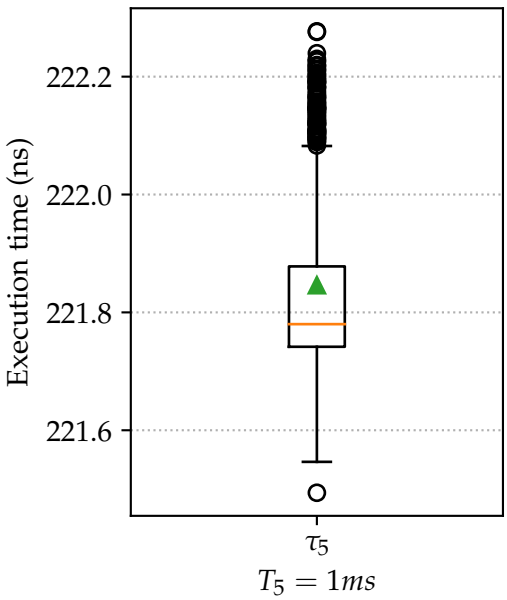


(d) τ_3 execution measurements.

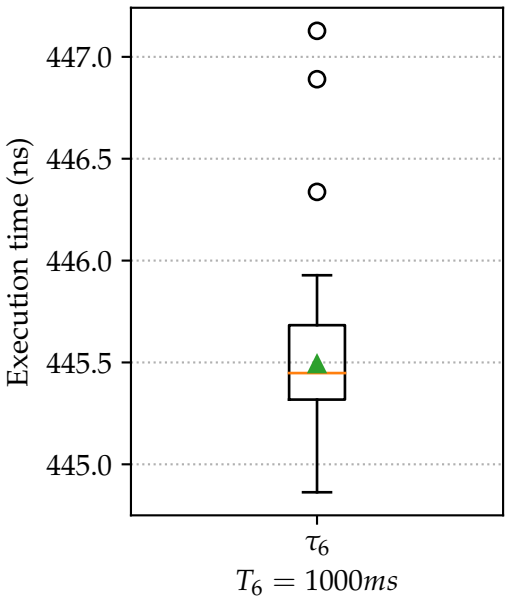
Figure A.1: Box plots with execution time measurements for τ_0 - τ_3.



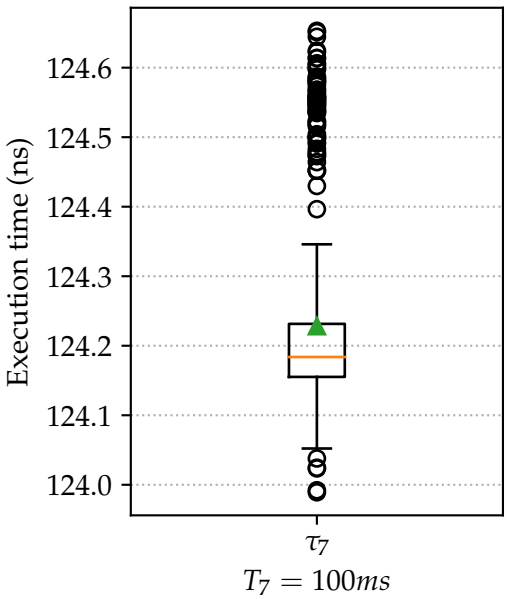
(a) τ_4 execution measurements.



(b) τ_5 execution measurements.

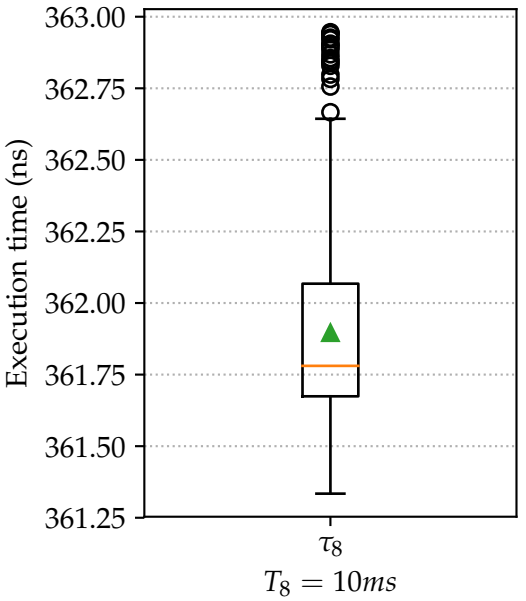


(c) τ_6 execution measurements.

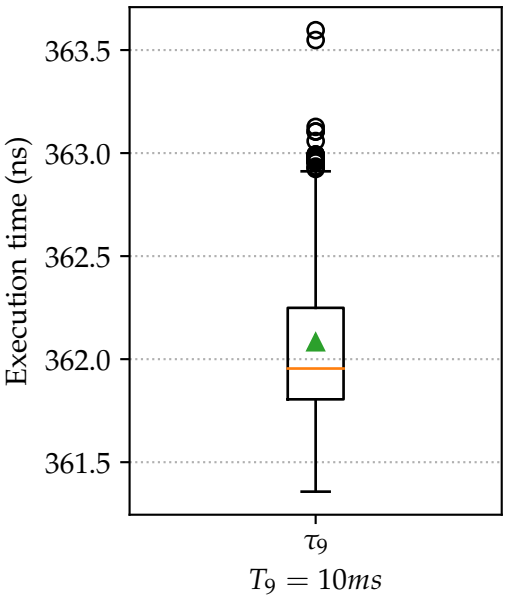


(d) τ_7 execution measurements.

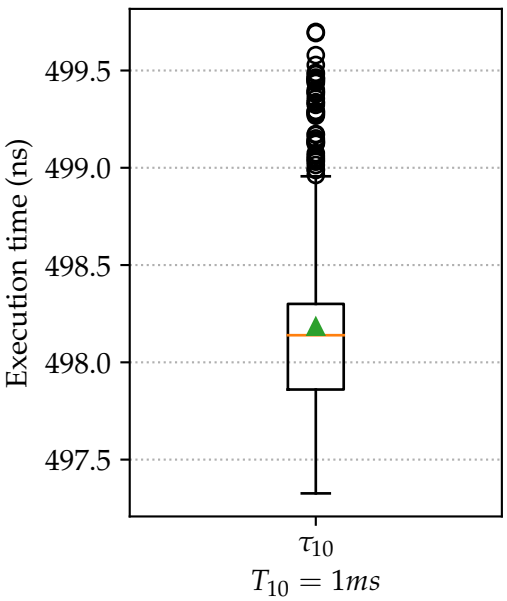
Figure A.2: Box plots with execution time measurements for τ_4 - τ_7.



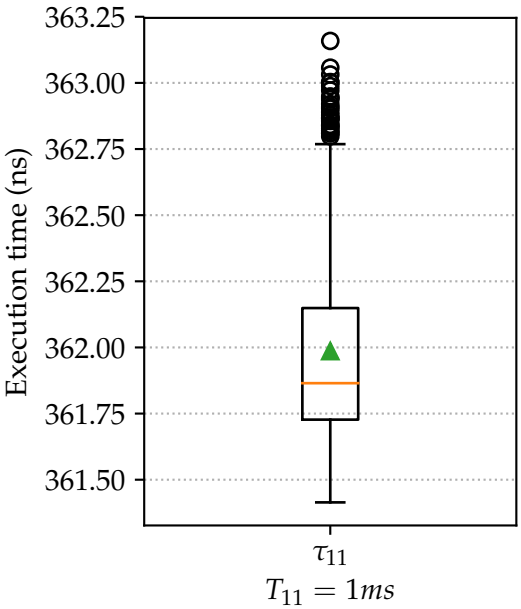
(a) τ_8 execution measurements.



(b) τ_9 execution measurements.

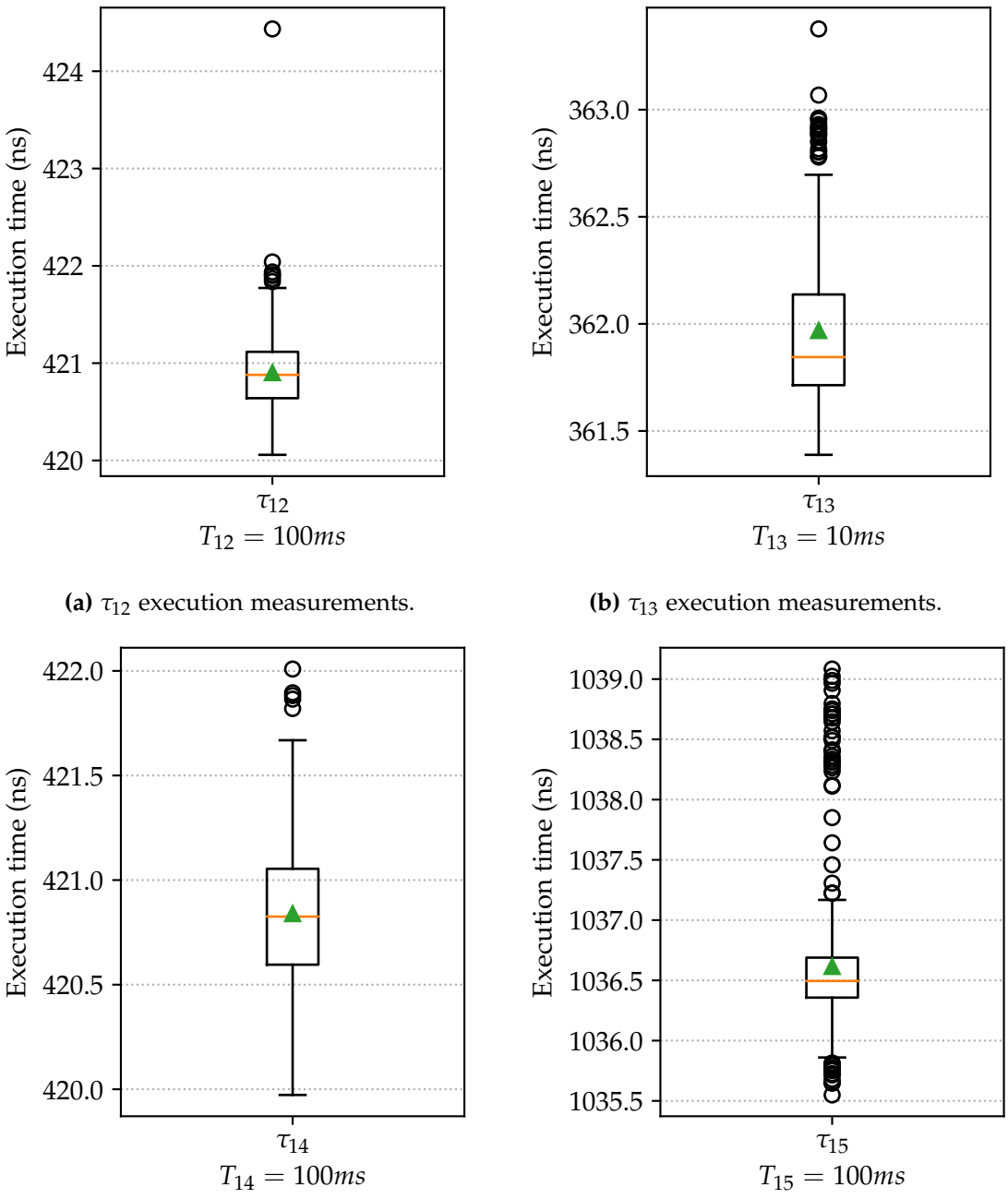


(c) τ_{10} execution measurements.



(d) τ_{11} execution measurements.

Figure A.3: Box plots with execution time measurements for τ_8 - τ_{11}.



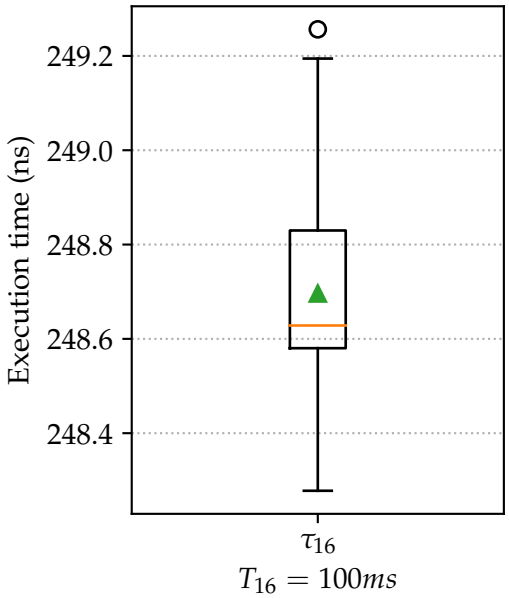
(a) τ_{12} execution measurements.

(b) τ_{13} execution measurements.

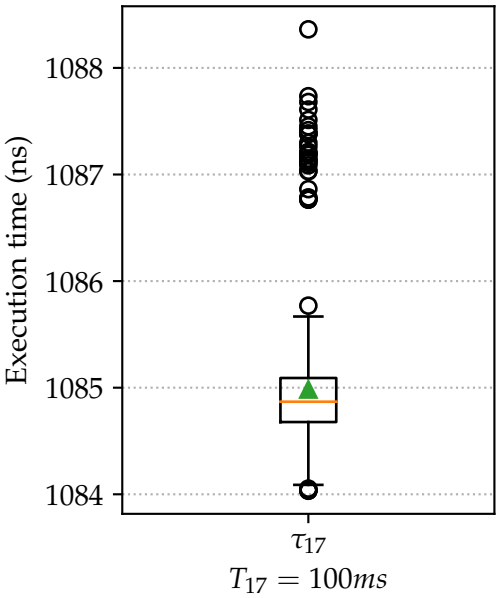
(c) τ_{14} execution measurements.

(d) τ_{15} execution measurements.

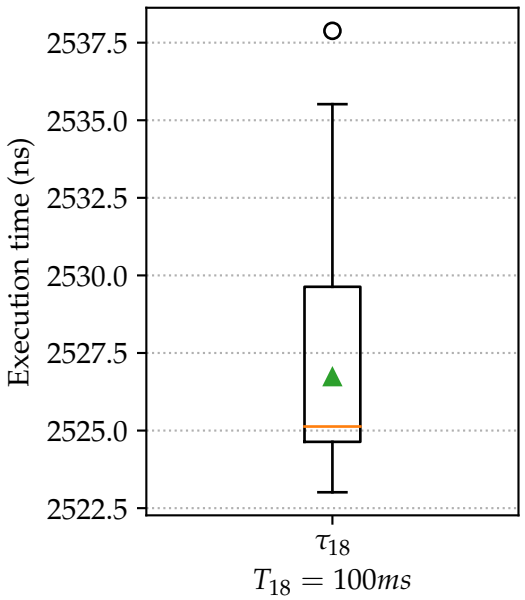
Figure A.4: Box plots with execution time measurements for τ_{12} - τ_{15}.



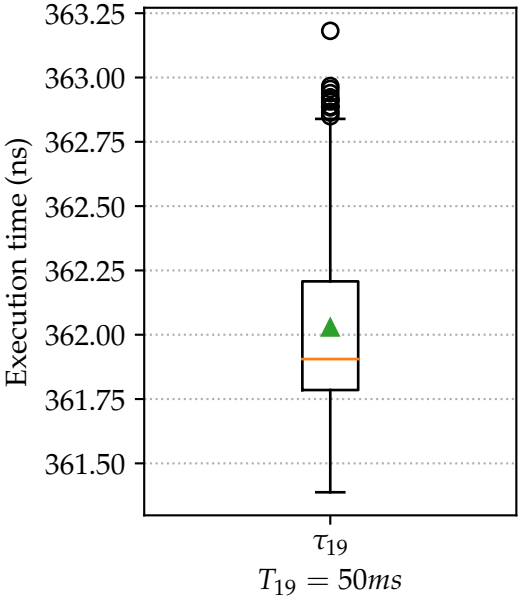
(a) τ_{16} execution measurements.



(b) τ_{17} execution measurements.

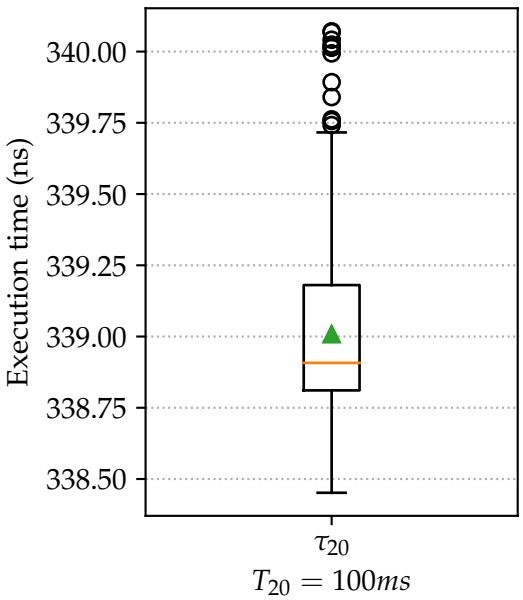


(c) τ_{18} execution measurements.

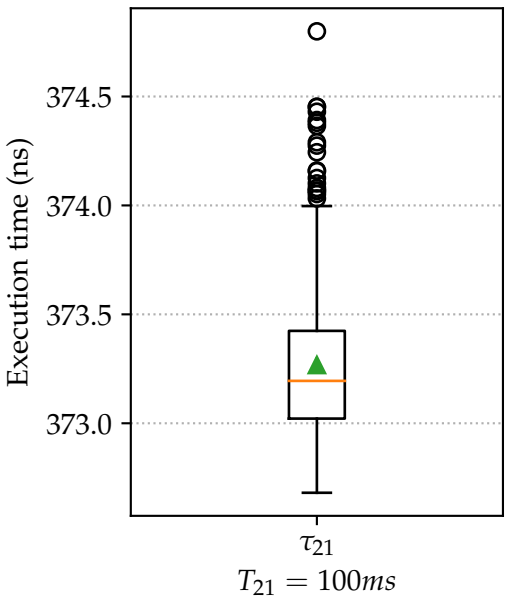


(d) τ_{19} execution measurements.

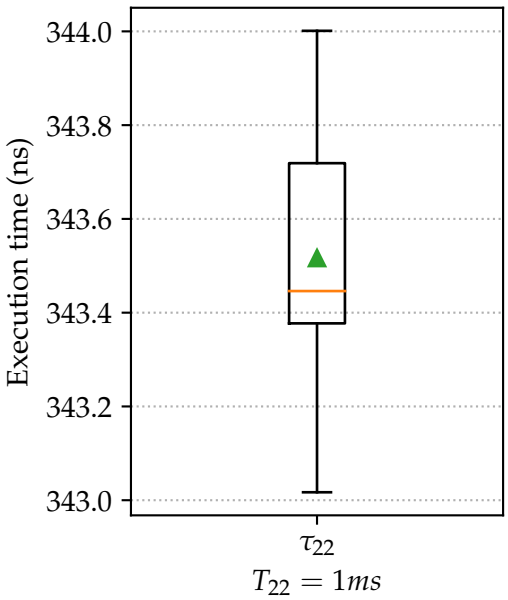
Figure A.5: Box plots with execution time measurements for τ_{16} - τ_{19}.



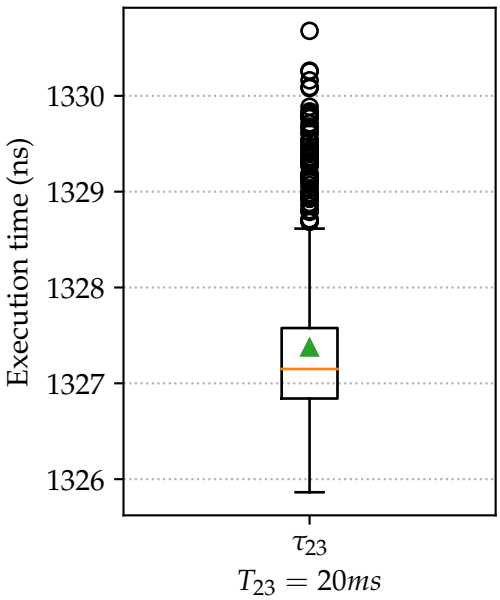
(a) τ_{20} execution measurements.



(b) τ_{21} execution measurements.

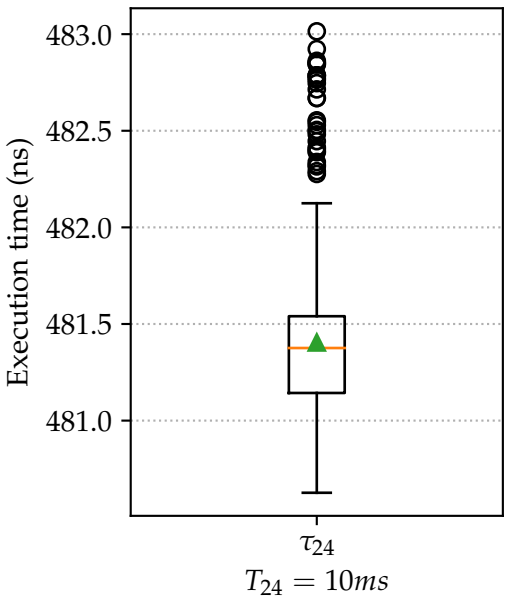


(c) τ_{22} execution measurements.

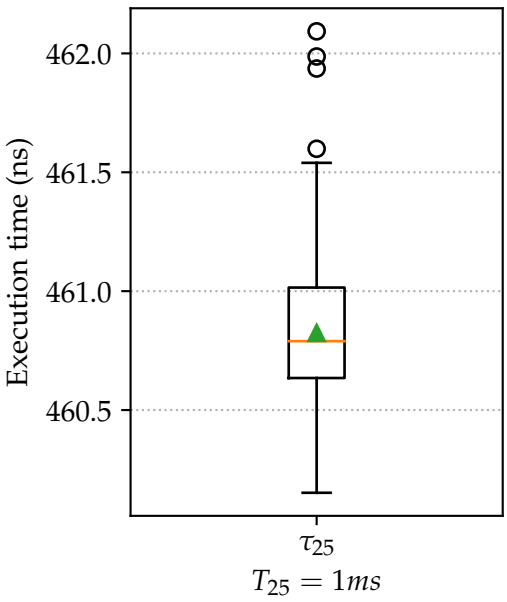


(d) τ_{23} execution measurements.

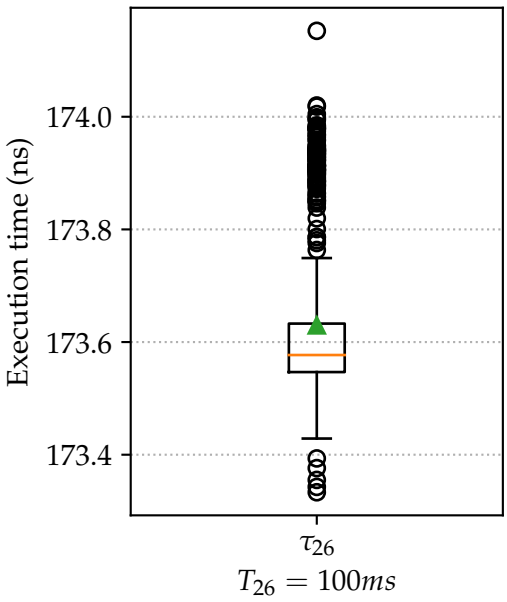
Figure A.6: Box plots with execution time measurements for τ_{20} - τ_{23}.



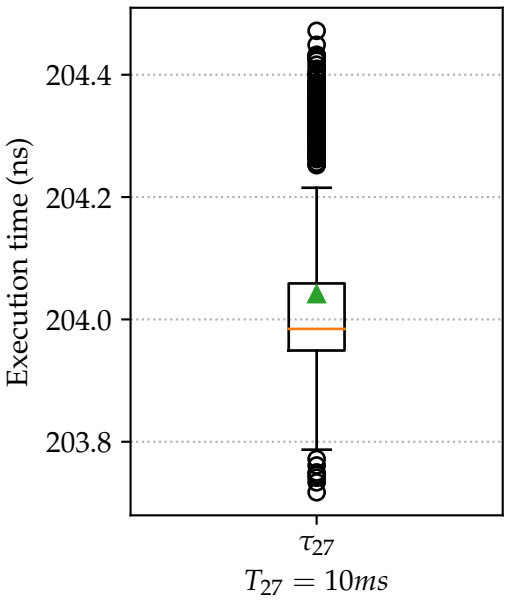
(a) τ_{24} execution measurements.



(b) τ_{25} execution measurements.

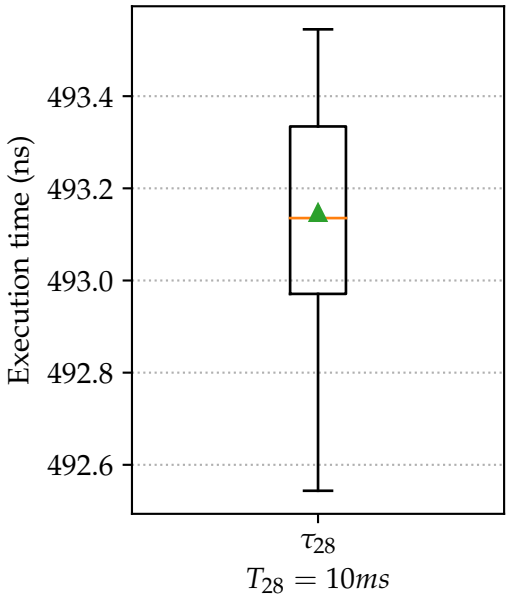


(c) τ_{26} execution measurements.

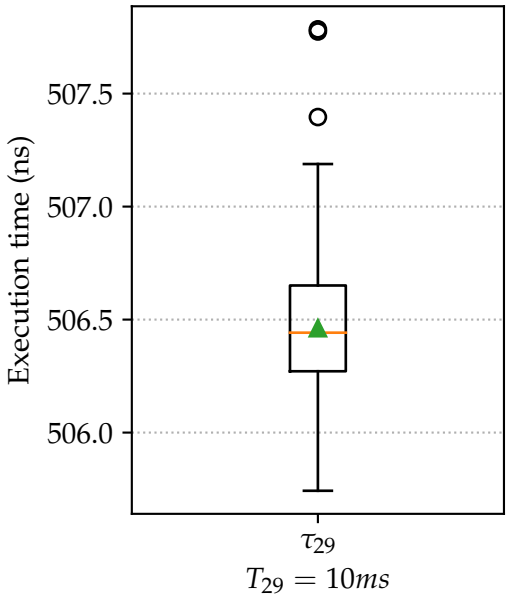


(d) τ_{27} execution measurements.

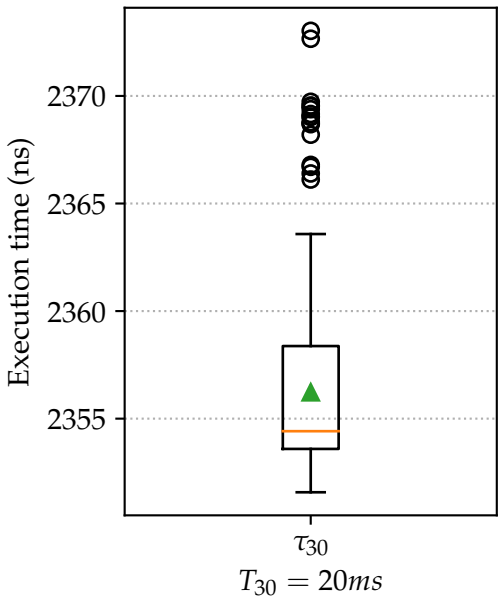
Figure A.7: Box plots with execution time measurements for τ_{24} - τ_{27}.



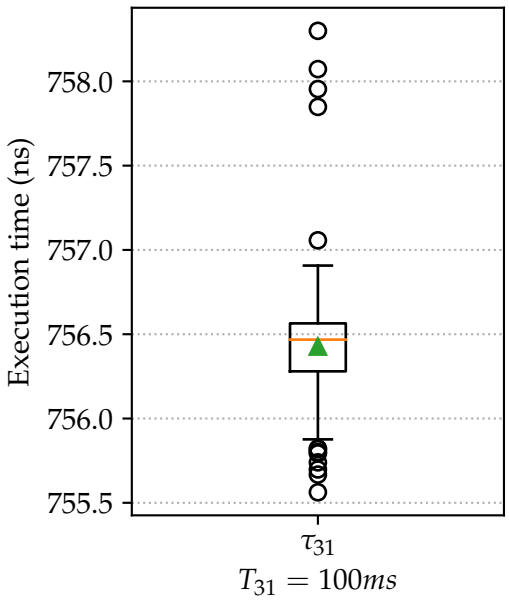
(a) τ_{28} execution measurements.



(b) τ_{29} execution measurements.



(c) τ_{30} execution measurements.



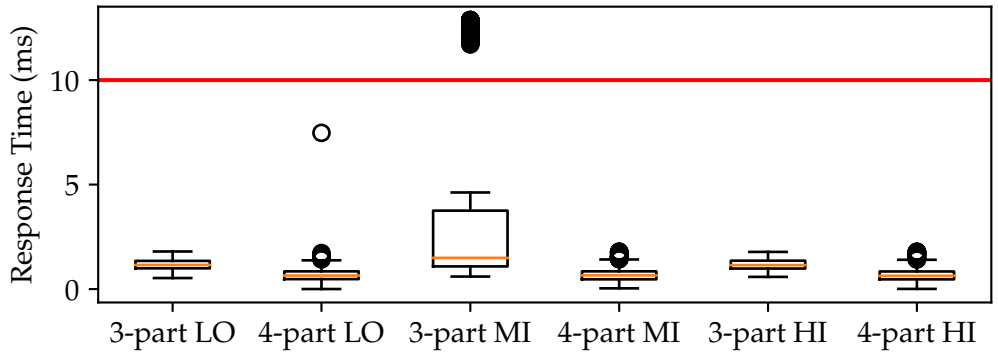
(d) τ_{31} execution measurements.

Figure A.8: Box plots with execution time measurements for τ_{28} - τ_{31}.

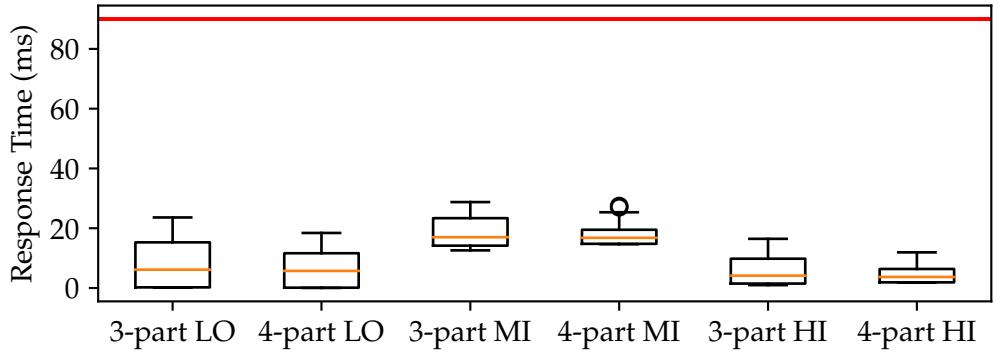
APPENDIX B

Olympus AOCS Case Study Response Times

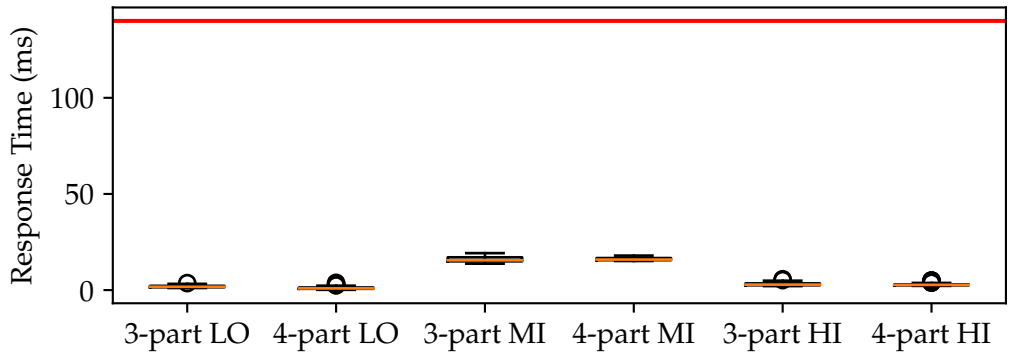
In this Appendix, we provide boxplots that summarise the observed response times of the Olympus AOCS taskset of Chapter 7. For each task we illustrate the response times during each mode of execution. The deadline of each task is marked by the horizontal line in the following plots.



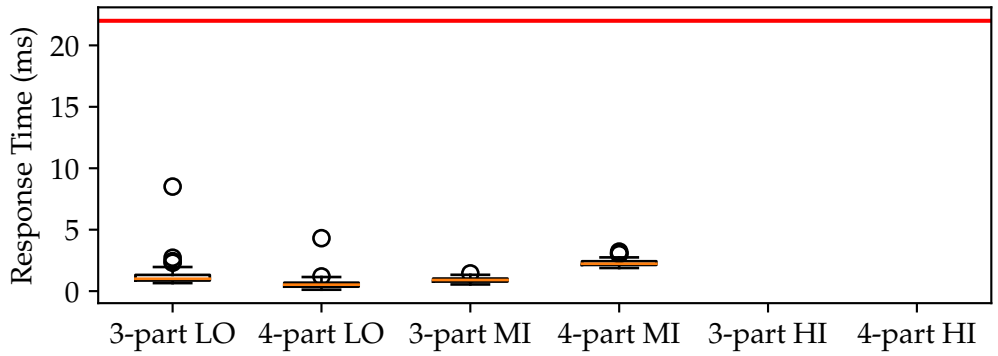
(a) Task C1: READ_BUS_IP, $L = HI$.



(b) Task C2: REAL_TIME_CLOCK, $L = HI$.

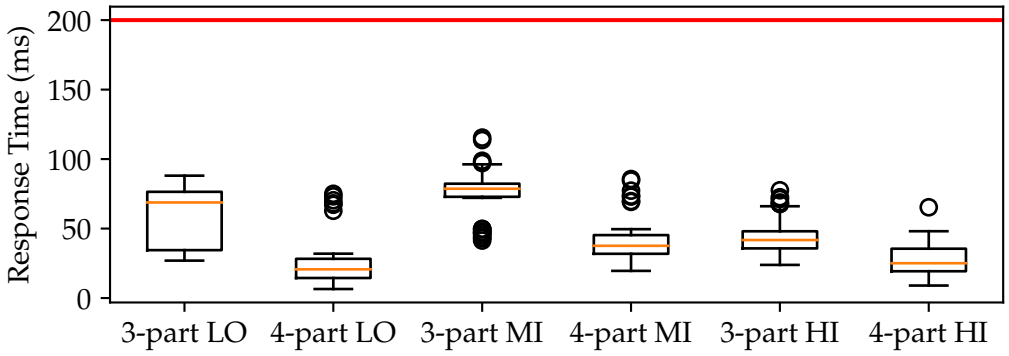


(c) Task C3: COMMAND_ACTUATORS, $L = MI$.

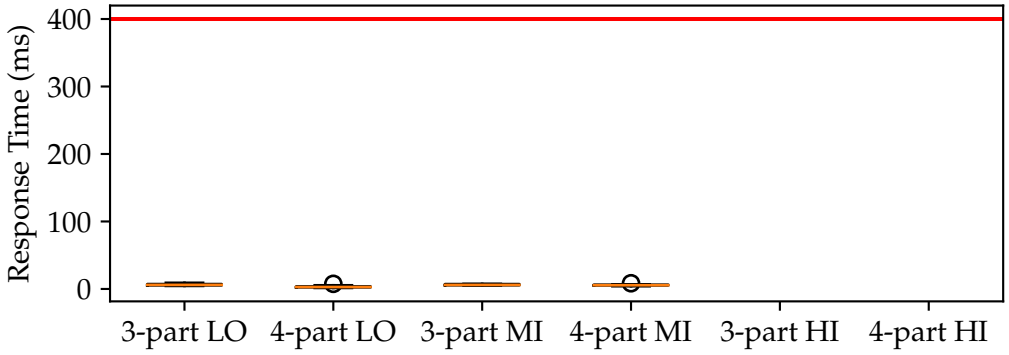


(d) Task C4: REQUEST_WHEEL_SPEEDS, $L = MI$.

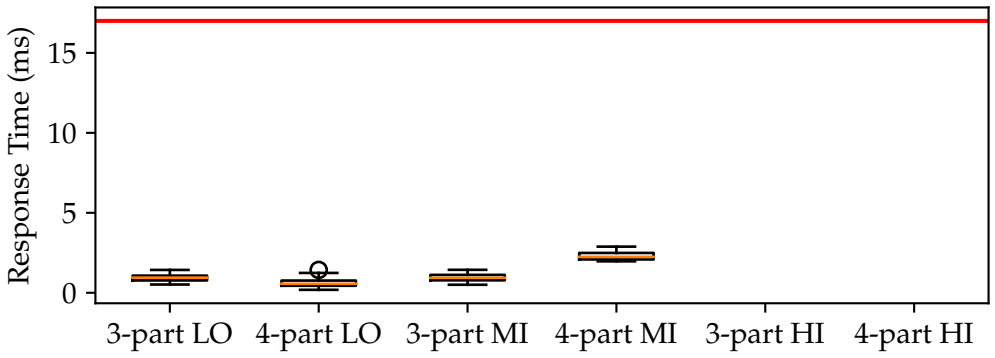
Figure B.1: Response times for Olympus AOCS periodic tasks: C1, C2, C3 and C4.



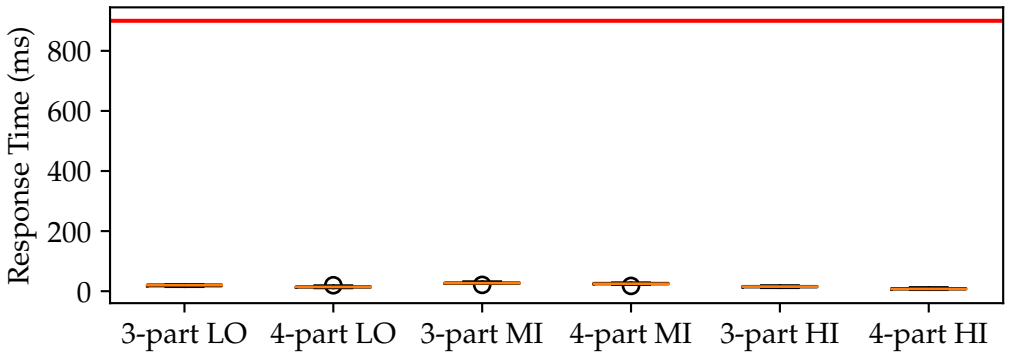
(a) Task C5: CONTROL_LAW, $L = HI$.



(b) Task C6: PROCESS_DSS_DATA, $L = MI$.

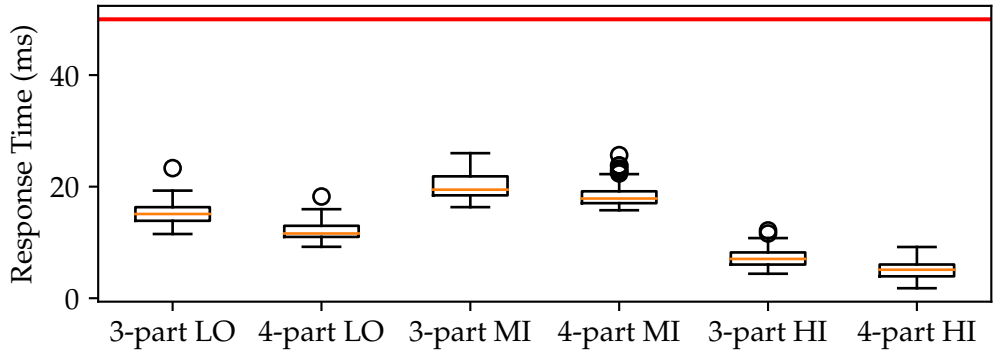


(c) Task C7: REQUEST_DSS_DATA, $L = MI$.

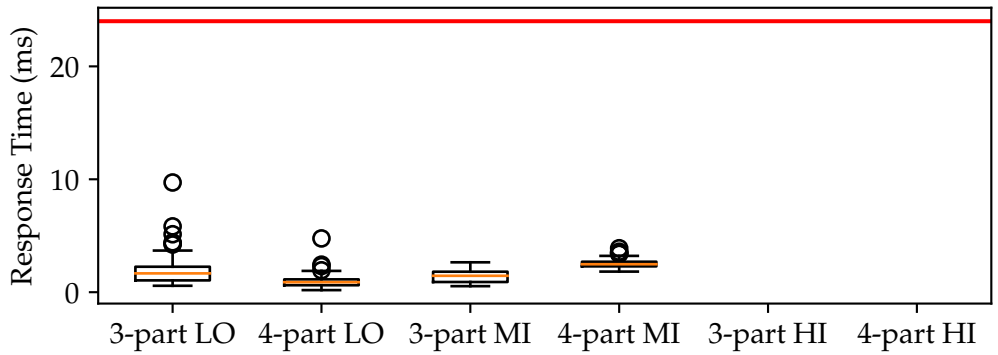


(d) Task C8: CALIBRATE_GYRO, $L = HI$.

Figure B.2: Response times for Olympus AOCS periodic tasks: C5, C6, C7 and C8.

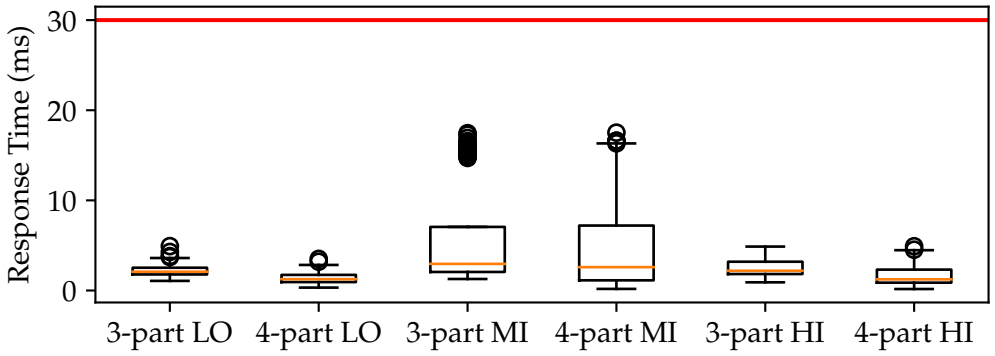


(a) Task C9: PROCESS_IRES_DATA, $L = HI$.

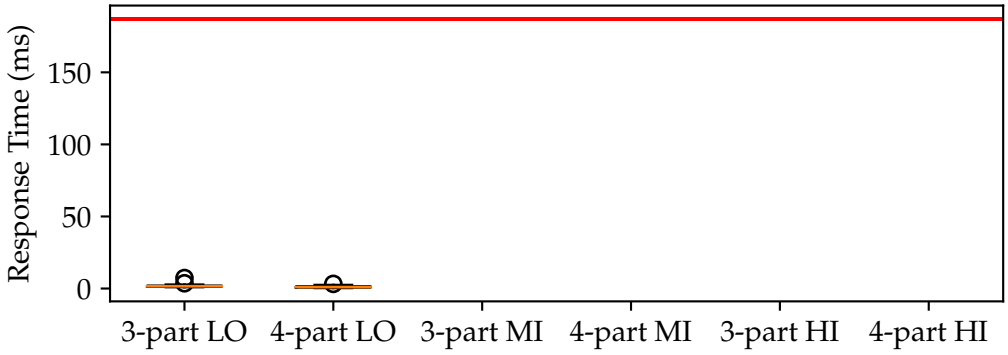


(b) Task C10: REQUEST_IRES_DATA, $L = MI$.

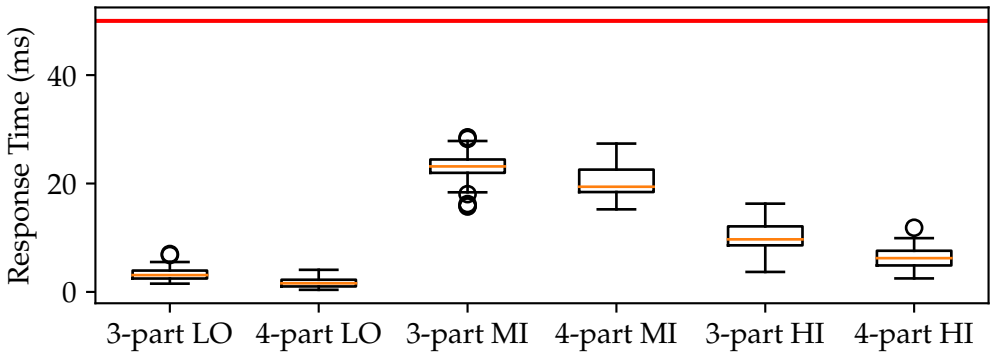
Figure B.3: Response times for Olympus AOCS periodic tasks: C9 and C10.



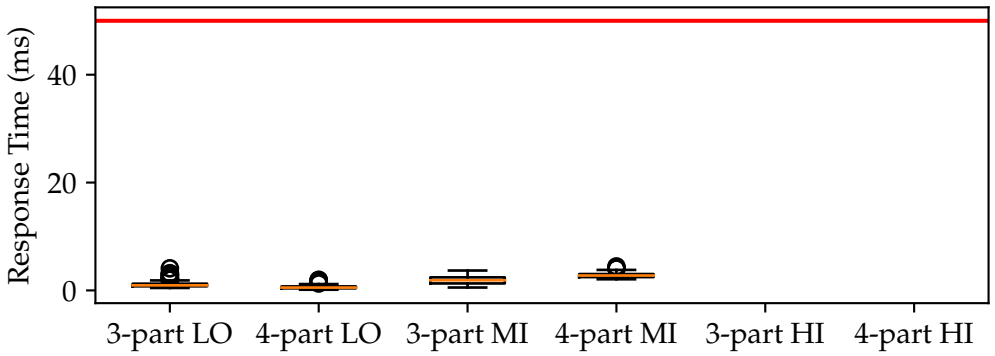
(a) Task S1: TELEMETRY_RESPONSE, $L = HI$.



(b) Task S2: TELECOMMANDS, $L = LO$.

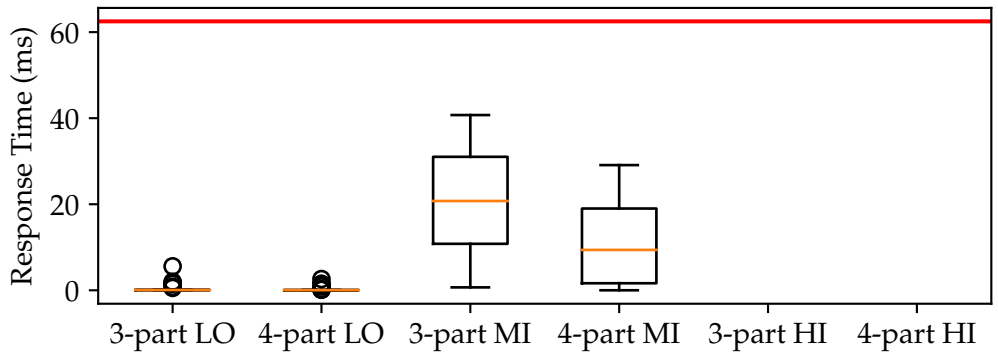


(c) Task S3: READ_YAW_GYRO, $L = HI$.

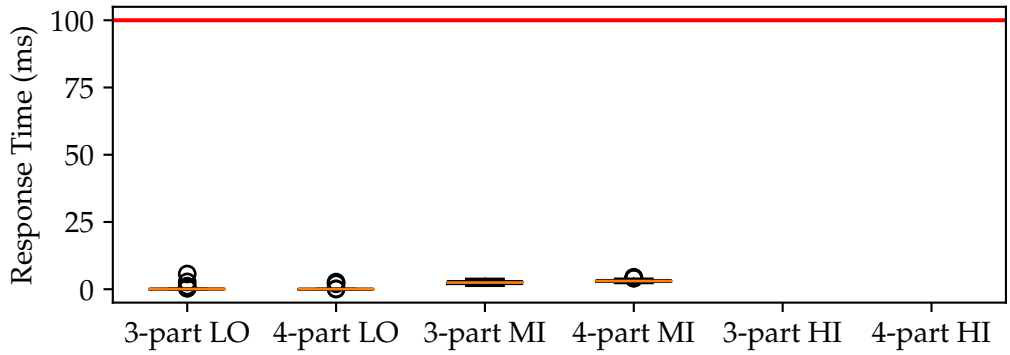


(d) Task S4: MESSAGES_HERE, $L = MI$.

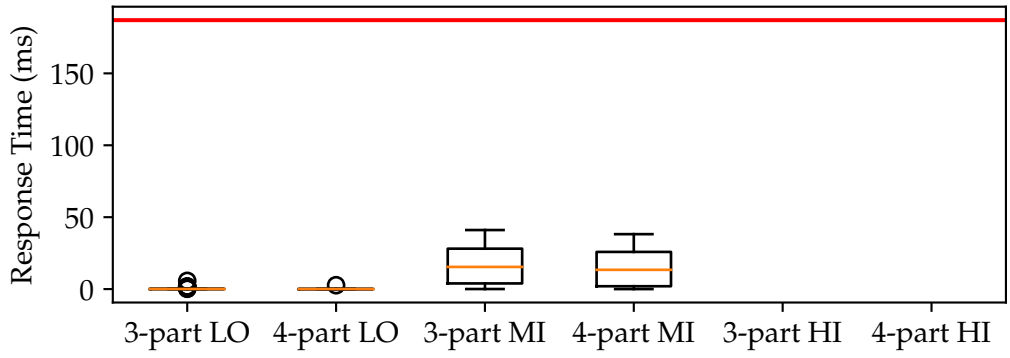
Figure B.4: Response times for Olympus AOCS sporadic tasks S1, S2, S3 and S4.



(a) Task S5: TM_HERE, $L = MI$.



(b) Task S6: ZI_HERE, $L = MI$.



(c) Task S7: TC_HERE, $L = MI$.

Figure B.5: Response times for Olympus AOCS sporadic tasks S5, S6 and S7.

Abbreviations

AOCS	Olympus Attitude and Orbital Control System.
ASIL	Automotive Safety Integrity Level.
AUTOSAR	Automotive Open System Architecture.
BCET	Best-case Execution Time.
BSW	Basic Software.
CBS	Constant Bandwidth Server.
CFA	Control-flow Analysis.
DPS	Dynamic-priority Scheduling.
ECU	Electronic Control Unit.
EDF	Earliest Deadline First.
FFI	Freedom From Interference.
FPS	Fixed-priority Scheduling.
G-EDF	Global Earliest Deadline First.
HV	Hypervisor.

IP	Intellectual Property.
IPET	Implicit Path Enumeration Technique.
IPL	Interrupt Priority Level.
MBTA	Measurement-based Timing Analysis.
MCAL	Microcontroller Abstraction Layer.
MCS	Mixed-Criticality System.
MMU	Memory Management Unit.
OEM	Original Equipment Manufacturer.
OS	Operating System.
P-EDF	Partitioned Earliest Deadline First.
RTA	Response-time Analysis.
RTE	AUTOSAR Runtime Environment.
SFQ	Start-time Fair Queueing.
SIL	Safety Integrity Level.
SPM	System Performance Monitor.
SWC	AUTOSAR Software Component.
VM	Virtual Machine.
VMM	Virtual Machine Manager.
WCET	Worst-case Execution Time.

References

[1] U. Abelein, H. Lochner, D. Hahn, and S. Straube, "Complexity, quality and robustness-the challenges of tomorrow's automotive electronics," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012, pp. 870–871.

[2] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, Dec 1998, pp. 4–13.

[3] L. Abeni, G. Lipari, and J. Lelli, "Constant bandwidth server revisited," *SIGBED Rev.*, vol. 11, no. 4, pp. 19–24, Jan. 2015.

[4] S. Altmeyer, B. Lisper, C. Maiza, J. Reineke, and C. Rochange, "WCET and Mixed-Criticality: What does Confidence in WCET Estimations Depend Upon?" in *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, ser. OpenAccess Series in Informatics (OASIcs), F. J. Cazorla, Ed., vol. 47. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 65–74.

[5] ARM Information Center. (2002) ARM PrimeCell™ VectoredInterrupt Controller (PL192) technical reference manual. ARM. [Accessed: 14 May. 2016]. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0273a/index.html

[6] ——. (2009) ARM1176JZF-S™technical reference manual. ARM. [Accessed: 14 May. 2016]. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0419c/index.html

[7] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, Sep. 1993.

[8] N. Audsley, "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times," University of York, York, United Kingdom, Tech. Rep., Nov. 1991.

[9] AUTOSAR. (2012, Aug.) AUTomotive Open System ARchitecture. [Accessed: 30 Sep. 2012]. [Online]. Available: http://www.autosar.org

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177.

[11] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications," in *Euromicro Conference on Real-Time Systems*, Jul. 2008, pp. 147–155.

[12] S. K. Baruah, A. Burns, and R. I. Davis, "Response-time analysis for mixed criticality systems," in *IEEE 32nd Real-Time Systems Symposium (RTSS)*, Nov. 2011, pp. 34–43.

[13] S. Baruah and A. Burns, "Implementing mixed criticality systems in Ada," in *Reliable Software Technologies - Ada-Europe*, ser. Lecture Notes in Computer Science, A. Romanovsky and T. Vardanega, Eds. Springer Berlin Heidelberg, 2011, vol. 6652, pp. 174–188.

[14] J. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," *Networking, IEEE/ACM Transactions on*, vol. 5, no. 5, pp. 675–689, Oct. 1997.

[15] E. Bini, G. Buttazzo, and G. Buttazzo, "Rate monotonic analysis: the hyperbolic bound," *Computers, IEEE Transactions on*, vol. 52, no. 7, pp. 933 – 942, Jul. 2003.

[16] H. Bo, D. Hui, W. Dafang, and Z. Guifan, "Basic concepts on AUTOSAR development," in *International Conference on Intelligent Computation Technology and Automation (ICICTA)*, vol. 1, May 2010, pp. 871–873.

[17] M. Broy, I. Kruger, A. Pretschner, and C. Salzmann, "Engineering automotive software," *Proceedings of the IEEE*, vol. 95, no. 2, pp. 356 –373, Feb. 2007.

[18] M. Broy, "Challenges in automotive software engineering," in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 33–42.

[19] A. Burns, "Scheduling hard real-time systems: a review," *Software Engineering Journal*, vol. 6, no. 3, pp. 116–128, May 1991.

[20] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, Jan. 2016.

[21] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*, 4th ed. Addison Wesley, May 2009.

[22] A. Burns, A. J. Wellings, C. Bailey, and E. Fyfe, "The olympus attitude and orbital control system a case study in hard real-time system design and implementation," in *Ada-Europe International Conference*. Springer, 1993, pp. 19–35.

[23] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Boston, MA: Springer US, 2011, ch. Fixed-Priority Servers, pp. 119–159.

[24] J. P. Buzen and U. O. Gagliardi, "The evolution of virtual machine architecture," in *Proceedings of the National Computer Conference and Exposition*, ser. AFIPS. New York, NY, USA: ACM, Jun. 1973, pp. 291–299.

[25] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper, "Applying static WCET analysis to automotive communication software," in *Real-Time Systems, 2005. (ECRTS 2005). Proceedings. 17th Euromicro Conference on*, Jul. 2005, pp. 249 – 258.

[26] A. Chandra and P. Shenoy, "Hierarchical scheduling for symmetric multiprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 19, no. 3, pp. 418–431, Mar. 2008.

[27] A. Chandra, M. Adler, P. Goyaly, and P. Shenoy, "Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors," 2000.

[28] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference*

Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Los Angeles, California: ACM Press, New York, NY, Jan. 1977, pp. 238–252.

[29] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The XtratuM approach," in *European Dependable Computing Conference (EDCC)*, Apr. 2010, pp. 67–72.

[30] R. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *26th IEEE International Real-Time Systems Symposium*, Dec. 2005, pp. 257–270.

[31] ——, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *27th IEEE International Real-Time Systems Symposium*, 2006, pp. 257–270.

[32] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.

[33] R. Devillers and J. Goossens, "Liu and layland's schedulability test revisited," *Information Processing Letters*, vol. 73, no. 5, pp. 157–161, 2000.

[34] F. Dorin, P. Richard, M. Richard, and J. Goossens, "Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities," *Real-Time Systems*, vol. 46, no. 3, pp. 305–331, 2010.

[35] Emlid. (2014) Raspberry pi real-time kernel. [Online]. Available: https://emlid.com/raspberry-pi-real-time-kernel/

[36] J. Engblom, A. Ermedahl, M. Nolin, J. Gustafsson, and H. Hansson, "Worst-case execution-time analysis for embedded real-time systems," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 4, pp. 437–455, Oct. 2003.

[37] A. Ermedahl, "A modular tool architecture for worst-case execution time analysis," Ph.D. dissertation, Uppsala Universitet, Uppsala, Sweden, Jun. 2003.

[38] A. Esper, G. Nelissen, V. Nélis, and E. Tovar, "How realistic is the mixed-criticality real-time system model?" in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*. ACM, 2015, pp. 139–148.

[39] European Space Agency. (1994) Olympus: End of mission. [Online]. Available: http://www.esa.int/For_Media/Press_Releases/OLYMPUS_End_of_mission

[40] C. Evripidou and A. Burns, "Scheduling for Mixed-criticality Hypervisor Systems in the Automotive Domain," in *WMC 2016 4th International Workshop on Mixed Criticality Systems*, Porto, Portugal, Nov. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01419143

[41] C. Evripidou and G. Morgan, "Method and apparatus for hosting a multitasking guest on a host system," Patent US Patent App. 15/215,113, 2017. [Online]. Available: https://www.google.com/patents/US20170024247

[42] C. Evripidou, G. Morgan, and A. Burns, "Method and apparatus for hosting a multitasking guest on a host system," Patent EP Patent App. EP20,150,177,684, 2017. [Online]. Available: https://www.google.com/patents/EP3121716A1

[43] ——, "Method and apparatus for hosting a multitasking guest on a host system," Patent CN Patent App. CN 201,610,826,878, 2017. [Online]. Available: https://www.google.com/patents/CN106453515A?cl=en

[44] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and precise WCET determination for a real-life processor," in *Embedded Software*, ser. Lecture Notes in Computer Science, T. Henzinger and C. Kirsch, Eds. Springer Berlin / Heidelberg, 2001, vol. 2211, pp. 469–485.

[45] freescale Semiconductor. (2008) FreescaleâĂŹs embedded hypervisor for qoriq p4 series communications platform (white paper). [Online]. Available: http://www.nxp.com/assets/documents/data/en/white-papers/EMBEDDED_HYPERVISOR.pdf

[46] P. Goyal, X. Guo, and H. M. Vin, "A hierarchical CPU scheduler for multimedia operating systems," in *USENIX 2nd Symposium on OS Design and Implementation (OSDI)*, Oct. 1996, pp. 107–122.

[47] P. Goyal, H. M. Vin, and H. Chen, "Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks," *SIGCOMM Comput. Commun. Rev.*, vol. 26, no. 4, pp. 157–168, Aug. 1996.

[48] J. Gustafsson, B. Lisper, C. Sandberg, and N. Bermudo, "A tool for automatic flow analysis of C-programs for WCET calculation," in *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, Jan. 2003, pp. 106 – 112.

[49] J. Hansen, S. A. Hissam, and G. A. Moreno, "Statistical-based WCET estimation and validation," in *Proceedings of the 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[50] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of wcet tools," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1038 – 1054, Jul. 2003.

[51] G. Heiser and B. Leslie, "The OKL4 microvisor: convergence point of microkernels and hypervisors," in *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, ser. APSys. New York, NY, USA: ACM, 2010, pp. 19–24.

[52] A. Hergenhan and G. Heiser, "Operating systems technology for converged ECUs," in *6th Embedded Security in Cars Conference (escar)*. Hamburg, Germany: ISITS, Nov. 2008.

[53] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson, "RTOS support for multicore mixed-criticality systems," in *IEEE 18th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2012, pp. 197–208.

[54] ISO, "Road vehicles – functional safety," International Organization for Standardization, International Standard ISO-26262, Nov. 2011.

[55] M. Joseph and P. Pandya, "Finding response times in a real-time system," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, 1986.

[56] R. Kaiser, "Alternatives for scheduling virtual machines in real-time embedded systems," in *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, ser. IIES. New York, NY, USA: ACM, 2008, pp. 5–10.

[57] ——, "Bringing together real-time and virtualization," in *Embedded World Conference*, Nuremberg, Germany, Feb. 2009.

[58] ——, "Combining partitioning and virtualization for safety-critical systems," *Embedded World Conference*, Jan. 2009.

[59] R. Kaiser and S. Wagner, "Evolution of the PikeOS microkernel," in *First International Workshop on Microkernels for Embedded Systems*, I. Kuz and S. M. Petters, Eds., National ICT Australia. Kensington, Australia: NICTA, Jan. 2007, pp. 50–57.

[60] F. Kirschke-Biller, S. Fürst, S. Lupp, S. Bunzel, S. Schmerler, R. Rimkus, A. Gilberg, K. Nishikawa, and A. Titze, "AUTOSAR - A worldwide standard: Current developments, roll-out and outlook," in *15th International VDI Congress Electronic Systems for Vehicles*, Baden-Baden, Germany, Oct. 2011.

[61] J. Knight, "Safety critical systems: challenges and directions," in *Proceedings of the 24rd International Conference on Software Engineering*, May 2002, pp. 547–550.

[62] O. K. Labs. (2012, Aug.) OKL4 microvisor. [Accessed: 01 Oct. 2012]. [Online]. Available: http://www.ok-labs.com/products/okl4-microvisor

[63] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig, "Flattening hierarchical scheduling," in *Proceedings of the tenth ACM international conference on embedded software*, ser. EMSOFT. New York, NY, USA: ACM, 2012, pp. 93–102.

[64] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *Proceedings of Real Time Systems Symposium*, Dec. 1989, pp. 166–171.

[65] J. Y.-T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, no. 4, pp. 237–250, 1982.

[66] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, ser. DAC. New York, NY, USA: ACM, 1995, pp. 456–461.

[67] Linux Foundation. (2013, Jul.) Xen Project advances open source virtualization with new release. San Francisco, CA. [Accessed: 16 Feb 2016]. [Online]. Available: http://www.xenproject.org/about/in-the-news/155-xen-project-advances-open-source-with-new-release.html

[68] ——. (2015, Apr.) Xen Project Schedulers. San Francisco, CA. [Accessed: 16 Feb 2016]. [Online]. Available: http://wiki.xenproject.org/wiki/Xen_Project_Schedulers

[69] ——. (2015, Apr.) Xen Project Software Overview. San Francisco, CA. [Accessed: 16 Feb 2016]. [Online]. Available: http://wiki.xen.org/wiki/Xen_Project_Software_Overview

[70] G. Lipari and E. Bini, "A methodology for designing hierarchical scheduling systems," *Journal of Embedded Computing*, vol. 1, no. 2, pp. 257–269, 2005.

[71] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[72] M. Masmano, I. Ripoll, and A. Crespo, "An overview of the XtratuM nanokernel," in *Proceedings of the Workshop on Operating System Platforms for Embedded Real-Time Applications*, 2005.

[73] M. Masmano, I. Ripoll, A. Crespo, and J. J. Metge, "XtratuM: a hypervisor for safety critical embedded systems," in *11th Real-Time Linux Workshop*, Jan. 2009.

[74] A. Masrur, T. Pfeuffer, M. Geier, S. Drössler, and S. Chakraborty, "Designing VM schedulers for embedded real-time applications," in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES+ISSS. New York, NY, USA: ACM, 2011, pp. 29–38.

[75] C. Maxim, A. Gogonel, D. Maxim, and L. Cucu, "Estimation of Probabilistic Minimum Inter-arrival Times Using Extreme Value Theory," in *6th Junior Researcher Workshop on Real-Time Computing (JRWRTC 2012) in conjunction with the 20th International Conference on Real-Time and Network Systems (RTNS 2012)*, Jan. 2013, pont-à-Mousson, France, November 8-9, 2012.

[76] E. Missimer, K. Missimer, and R. West, "Mixed-criticality scheduling with I/O," in *Proceedings of the 28th Euromicro Converence on Real-Time Systems (ECRTS)*, Toulouse, France, Jul. 2016, pp. 120–130.

[77] M. Mollison, J. Erickson, J. Anderson, S. Baruah, and J. Scoredos, "Mixed-criticality real-time scheduling for multicore systems," in *IEEE 10th International Conference on Computer and Information Technology (CIT)*, 2010, pp. 1864–1871.

[78] G. Morgan, "Deeply embedded real-time hypervisors for the automotive domain," in *20th IEEE Real-Time and Embedded Technology and Applications Symposium*, Berlin, Germany, Apr. 2014.

[79] ——, "Safety and security with hypervisor technology," in *Embedded World Conference*. Nuremberg, Germany: Design & Elektronik, Feb. 2016.

[80] T. Nolte, "Hierarchical scheduling of complex embedded real-time systems," in *Ecole d'Ete Temps-REel (ETR)*, 2009.

[81] *OKL4 Microkernel Reference Manual*, Open Kernel Labs, Alexandria, Australia, Sep. 2008, [Accessed: 30 Sep. 2012]. [Online]. Available: http://www.reds.ch/share/cours/SEEE/okl4-ref-manual-3.0.pdf

[82] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[83] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner, "Software engineering for automotive systems: A roadmap," in *2007 Future of Software Engineering*, ser. FOSE. Washington, DC, USA: IEEE Computer Society, 2007, pp. 55–71.

[84] P. Puschner and A. Burns, "Guest editorial: A review of worst-case execution-time analysis," *Real-Time Systems*, vol. 18, pp. 115–128, 2000.

[85] M. Åsberg, M. Behnam, F. Nemati, and T. Nolte, "Towards hierarchical scheduling in autosar," in *IEEE Conference on Emerging Technologies Factory Automation (ETFA)*, Sep. 2009, pp. 1 –8.

[86] M. Åsberg, T. Nolte, and P. Pettersson, "Prototyping hierarchically scheduled systems using task automata and times," in *5th International Conference on Embedded and Multimedia Computing (EMC)*, Aug. 2010, pp. 1–8.

[87] (2012, Aug.) XtratuM. Real-Time System Group, Instituto de Automatica e Informatica Industrial, Universitat Politecnica de Valencia. Spain. [Accessed: 16 Feb 2016]. [Online]. Available: http://www.xtratum.org

[88] D. Reinhardt, D. Kaule, and M. Kucera, "Achieving a scalable E/E-Architecture using AUTOSAR and virtualization," *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, vol. 6, no. 2, pp. 489–497, 2013.

[89] D. Reinhardt and M. Kucera, "Domain controlled architecture - a new approach for large scale software integrated automotive systems." in *PECCS*, 2013, pp. 221–226.

[90] D. Reinhardt and G. Morgan, "An embedded hypervisor for safety-relevant automotive E/E-systems," in *9th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2014, pp. 189–198.

[91] L. Rierson, *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2013.

[92] W. River. (2009) Wind river hypervisor (product overview). [Online]. Available: https://www.windriver.com/products/product-overviews/wr-hypervisor-product-overview.pdf

[93] L. Sha, J. P. Lehoczky, and R. Rajkumar, "Solutions for some practical problems in prioritized preemptive scheduling." in *RTSS*, vol. 86, 1986, pp. 181–191.

[94] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[95] G. Software. (2017) INTEGRITY multivisor - virtualization architecture for secure systems. [Online]. Available: http://www.ghs.com/products/rtos/integrity_virtualization.html

[96] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, 1989.

[97] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 73–91, 1995.

[98] V. Sundaram, A. Chandra, P. Goyal, P. Shenoy, J. Sahni, and H. Vin, "Application performance in the qlinux multimedia operating system," in *Proceedings of the eighth ACM international conference on Multimedia*, ser. MULTIMEDIA. New York, NY, USA: ACM, 2000, pp. 127–136.

[99] SYSGO AG. (2012, Aug.) PikeOS RTOS and virtualization concept. [Accessed: 16 Feb 2016]. [Online]. Available: http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/

[100] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand, "An abstract interpretation-based timing validation of hard real-time avionics software," in *Proceedings of International Conference on Dependable Systems and Networks*, Jun. 2003, pp. 625–632.

[101] K. Tindell, A. Burns, and A. Wellings, "Mode changes in priority preemptively scheduled systems," in *Real-Time Systems Symposium*, Dec. 1992, pp. 100–109.

[102] F. Verbeek, O. Havle, J. Schmaltz, S. Tverdyshev, H. Blasum, B. Langenstein, W. Stephan, B. Wolff, and Y. Nemouchi, "Formal API specification of the PikeOS separation kernel," in *NASA Formal Methods*. Springer, 2015, pp. 375–389.

[103] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *28th IEEE International Real-Time Systems Symposium (RTSS)*, 2007, pp. 239–243.

[104] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based timing analysis," in *Leveraging Applications of Formal Methods, Verification and Validation*, ser. Communications in Computer and Information Science, T. Margaria and B. Steffen, Eds. Springer Berlin Heidelberg, 2009, vol. 17, pp. 430–444.

[105] J. Whitham, N. C. Audsley, and R. I. Davis, "Explicit reservation of cache memory in a predictable, preemptive multitasking real-time system," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, pp. 120:1–120:25, Apr. 2014.

[106] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[107] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: towards real-time hypervisor scheduling in xen," in *Proceedings of the ninth ACM international conference on Embedded software*, ser. EMSOFT. New York, NY, USA: ACM, 2011, pp. 39–48.

[108] S. Xi, M. Xu, C. Lu, L. T. X. Phan, C. Gill, O. Sokolsky, and I. Lee, "Real-time multi-core virtual machine scheduling in Xen," in *Proceedings of the 14th International Conference on Embedded Software*, ser. EMSOFT. New York, NY, USA: ACM, 2014, pp. 27:1–27:10.