UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

STRICT REAL-TIME SYSTEMS DESIGN OVER MULTIPROCESSOR-BASED PLATFORMS

TESIS DOCTORAL

JORGE GARRIDO BALAGUER
MÁSTER EN INGENIERÍA DE REDES Y SERVICIOS TELEMÁTICOS

2018

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS TELEMÁTICOS

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

STRICT REAL-TIME SYSTEMS DESIGN OVER MULTIPROCESSOR-BASED PLATFORMS

Autor: Jorge Garrido Balaguer
Máster en Ingeniería de Redes y Servicios Telemáticos

Director: Juan Antonio de la Puente Alfaro
Doctor Ingeniero Industrial

2018

Tribunal nombrado por el Magfco. y Excmo. Sr. Rector de la Universidad Politécnica de Madrid, el día ___ de ______________ de 20__.

Presidente: ______________________________________

Vocal: ____

Vocal: ____

Vocal: ____

Secretario: ______________________________________

Suplente: _______________________________________

Suplente: _______________________________________

Realizado el acto de defensa y lectura de la Tesis el día ___ de ______________ de 20__ en la E.T.S.I. /Facultad ________________.

Calificación:______________________________________

EL PRESIDENTE LOS VOCALES

EL SECRETARIO

Contents

List of Figures

List of Tables

Acronims

ABS Anti-lock Braking System

ACS Attitude Control System

ACK Acknowledge

ADC Analog to Digital Converter

AMBA Advanced Microcontroller Bus Architecture

BF Best Fit

CPU Central Processing Unit

DAS Data Acquisition Subsystem

DM-PM Deadline Monotonic with Priority Migration

EBOX Electronic Box

EDF Earliest Deadline First

EDF-WM Earliest Deadline First with Window-constraint Migration

ESA European Space Agency

FPGA Field Programmable Gate Array

FF First Fit

FIFO	First In First Out
FMLP	Flexible Multiprocessor Locking Protocol
FPS	Fixed Priority Scheduling
G-EDF	Global Earliest Deadline First
GS	Ground Station
ILP	Integer Linear Programming
IPCP	Immediate Priority Ceiling Protocol
JLSP	Job Level Static Priority
MB	Mega Byte
MGT	Magnetorquer
MMU	Memory Management Unit
MPCP	Multiprocessor Priority Ceiling Protocol
MrsP	Multiprocessor resource sharing Protocol
MSRP	Multiprocessor Stack Resource Policy
NACK	Not Acknowledge
NP	Non-deterministic Polynomial
OBC	On-Board Computer
OBDH	On-Board Data Handling
OS	Operating System
P-EDF	Partitioned Earliest Deadline First
PCP	Priority Ceiling Protocol

PDU	Power Distribution Unit
PIP	Priority Inheritance Protocol
PSU	Power Supply Unit
PWM	Pulse Width Modulation
RMS	Rate Monotonic Scheduling
RW	Reaction Wheel
SMP	Symmetric MultiProcessing
SoC	System-on-Chip
SRP	Stack Resource Policy
TTC	Telemetry and Telecommand
VHDL	Very high speed integrated circuit Hardware Description Language
WCET	Worst-Case Execution Time
WDT	Watch Dog Timer
WF	Worst Fit

Acknowledgements Agradecimientos

This work has been partially funded by the Spanish National R&D&I plan (project M2C2, TIN2014-56158-C4-3-P and predoctoral grant BES-2015-074982). This funding included a specific assignation to support a research stay at a foreign research institution. That research stay was conducted at the Real-Time Systems group at the University of York, under the supervision of professor Alan Burns, whom I wish to show my deep appreciation, as well as to professor Andy Wellings and Shuai Zhao.

Este trabajo ha sido parcialmente financiado por el plan Plan Nacional Español de I+D+i (proyecto M2C2, TIN2014-56158-C4-3-P, incluyendo la beca predoctoral BES-2015-074982). Esta financiación ha incluido apoyo económico para la realización de una estancia de investigación en una institución estranjera. Dicha estancia se llevó a cabo en el grupo de Sistemas de Tiempo Real de la Universidad de York bajo la dirección del profesor Alan Burns, a quien quiero agradecer especialmente su generosidad al considerarme para dicha estancia. Del mismo modo quiero agradecerle a él y al profesor Andy Wellings y al investigador Shuai Zhao la fructífera colaboración que desde esa estancia hemos venido manteniendo.

Publications

This thesis encompasses the main research activities on real-time systems over multiprocessor platforms conducted by the author under the supervision of professor Juan Antonio de la Puente at the Real-Time Systems and Architecture of Telematic Services (STRAST), integrated into the Information Processing and Telecommunications Center (IPTC) of Universidad Politécnica de Madrid (UPM), Spain. It also covers the results of the fruitful research stay conducted at the Real-Time Systems Group of the University of York (UoY) under the supervision of professor Alan Burns.

The research presented here has been published in different research forums of relevance in the real-time systems area and have all undergone peer review processes, being evaluated by recognized experts in the area. All of them have also been orally presented in plenary or poster sessions. The publications containing the major contributions to this thesis are listed below:

- [71] Jorge Garrido, Shuai Zhao, Alan Burns, and Andy Wellings. Supporting nested resources in MrsP. In Johann Blieberger and Markus Bader, editors, *Ada-Europe International Conference on Reliable Software Technologies*, pages 73–86. Springer, 2017.

- [69] Jorge Garrido, Juan Zamorano, Alejandro Alonso, and Juan A. de la Puente. Evaluating MSRP and MrsP with the Multiprocessor Ravenscar Profile. In Johann Blieberger and Markus Bader, editors, *Ada-Europe International Conference on Reliable Software Technologies*, pages 3–17. Springer, 2017.

- [68] Jorge Garrido, Juan A. de la Puente, and Juan Zamorano. MrsP on semi-partitioned systems. *28th Euromicro Conference on Real-Time Systems (ECRTS16) - WiP Session proceedings*, pages 4–6, 2016.

- [138] Shuai Zhao, Jorge Garrido, Alan Burns, and Andy Wellings. New schedulability analysis for MrsP. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2017 IEEE 23rd International Conference on*, pages 1–10. IEEE, 2017.

Abstract

In the past decades the digitalization and automation have played a key role in the economic growth and development. An increasing number of activities supporting the levels of welfare achieved are based on the adoption of modern technologies in which computers have enabled higher service levels: communications, transportation, industrial production, health care or education are some examples. Nowadays energy production and distribution are controlled by computer systems, as well as our daily communications via mobile phones and emails, and we rely on many electronic-driven safety measures now standard in the automotive industry.

Most of these embedded computer systems are in charge of controlling complex critical systems concerning human lives and business viability. These systems present strict requirements to ensure proper behaviour. Some of these requirements address non-functional aspects, as the temporal correctness of the system. In other words, for some relevant computer-controlled systems it is not only important to produce the correct output but also to produce it on time.

This dissertation aims to contribute on the development of systems with strict temporal requirements based on multiprocessor platform. Modern hardware technology has enabled the massive production of processors incorporating more than one computation unit per chip, increasing the overall computer performance. These new platforms present, however, a set of challenges to achieve the required reliability

according to the critical environments in which they operate.

In this dissertation the temporal analysis of real-time systems based on multiprocessor platforms is addressed. In particular it contributes in a number of ways on the definition and study of a new scheduling protocol, the Multiprocessor resource sharing Protocol (MrsP). This protocol aims to transfer the most successful approaches from monoprocessor theory and practice in combination with the most promising techniques for strict multiprocessor real-time systems design. This is substantiated in the use of ceiling priority and equivalent stack resource policies as task dispatching strategies with a preemptable spin-locking mechanism for shared resource access arbitration. Global spin delay interference is mitigated by a novel helping mechanism, based on the cooperative completion of resource accesses locally preempted.

Specific contributions addressed in this work include the formalization of a more exact timing analysis based on the heterogeneity of access times present in practice. The task model is then completed with support for nested resource access patterns, including means for analysing this task model with two scheduling analysis based on the well known response time analysis technique. These two different analysis can prove the system feasibility with different degrees of system knowledge: while the sufficient analysis enables testing the schedulability of the system with reduced knowledge of resource access patterns, a tighter, less pessimistic analysis can achieved when all shared resource access patterns have been fully characterized.

These contributions are supported by the extensive evaluation work already published. The evaluation section of this dissertation complements that work with the analysis of the flight segment of a space mission which is then compared with that of a traditional monoprocessor approach as well as with a comparable multiprocessor protocol.

Resumen

En las últimas décadas, la digitalización y automatización han tenido un rol determinante en el desarrollo económico. Cada vez más procesos y actividades están siendo impulsadas por la adopción de nuevas tecnologías, ofreciendo mayores cotas de bienestar. Muchas de estas tecnologías, incluyen el uso de computadores embarcados. Su aplicación se da en, entre otros, sectores como las telecomunicaciones, el transporte, la producción industrial, o la sanidad y la educación, más recientemente. Desde los modernos teléfonos móviles inteligentes a los sistemas de seguridad electrónicos cada vez más extendidos en la industría automovilística, la influencia de los computadores embarcados es cada vez mayor en nuestro día a día.

Muchos de estos sistemas se encargan de manejar sistemas críticos, en los que un fallo puede poner en peligro vidas humanas o causar graves pérdidas económicas. Es por ello que estos sistemas presentan requisitos muy estrictos con el objetico de prevenir tan dramáticas consecuencias. Entre ellos se encuentran los requisitos temporales. Estos requisitos implican que la respuesta del sistema, para ser válida no solo ha de ser correcta, si no además producida dentro de unos parámetros temporales determinados.

En esta tesis se pretende contribuir al estado de la técnica en el desarrollo de estos sistemas con requisitos temporales estrictos sobre plataformas multiprocesador. Si bien estas modernas plataformas ofrecen una mayor capacidad de cómputo, también presentan mayores di-

ficultades a la hora de alcanzar los estrictos requisitos que imponen los entornos en los que los sistemas embarcados operan.

En concreto, esta tesis se centra en el análisis temporal de estos sistemas basados en plataformas multiprocesador. La principal contribución es el estudio y definición del protocolo *Multiprocessor resource sharing Protocol* (MrsP por sus siglas en inglés). Este protocolo, en esencia, intenta trasladar a los sistemas multiprocesador los conocimientos y técnicas ya empleados satisfactoriamente en los actuales sistemas monoprocesador. Esto se sustancia en el uso del protocolo de techo de prioridad como método de planificación del uso de procesador, arbitrando el acceso a recursos compartidos entre procesadores mediante cerrojos con espera activa. Al realizarse este acceso y espera al techo de prioridad local del recurso, son potencialmente desalojables. Para limitar las consecuencias de un posible desalojo de una tarea durante su acceso al recuros, el protocolo incorpora un mecanismo de ayuda por el cual las tareas haciendo espera activa pueden colaborar con la tarea desalojada localmente a completar su acceso.

Las contribuciones de esta tesis incluyen la formalización de un análisis más exacto para el protocolo MrsP, al tener en cuenta la habitual heterogeneidad en los tiempos de acceso a los recursos en función de la acción a llevar a cabo. Posteriormente se extiende el protocolo para dar soporte a la anidación de accesos a recursos compartidos. Para ello, primero se definen las reglas que rigen estos accesos anidados, ofreciendose posteriormente dos métodos para demostrar el cumplimiento de los requisitos temporales. El primero permite realizar dicho análisis con un conocimiento limitado de los patrones de acceso a recursos compartidos. El segundo permite, a partir de un conocimiento completo de dichos patrones, obtener un análisis más ajustado, permitiendo demostrar la viabilidad de un mayor número de sistemas.

Estas contribuciones han sido contrastadas mediante un completo trabajo de evaluación publicado durante la investigación que aquí se recoge. En esta tesis, ese trabajo se ve complementado con el análisis del segmento de vuelo de una misión espacial. En la sección de evaluación se detalla el análisis del sistema mediante el protocolo MrsP, el cual se compara con el análisis de una implementación de dicho sistema en un monoprocesador, así como en un multiprocesador utilizando un protocolo comparable.

Chapter 1

Introduction

Real-time systems are those systems in which the behaviour is not only required to be logically correct but also provided within a set of temporal requirements. Since its appearance in the early 1970's, the real-time community built the required theoretical and practical frame based on the assumption that only one thread or computation flow could be executing in the system at any single time. Monocore processors support the implementation of multi-thread systems by time-division multiplexing the processor time.

In the monocore processor case, the scheduling of the system, i.e., the generation of a succession of thread execution periods satisfying the temporal requirements is focused on the processor allocation. In monocore processors, the access to other system resources, such as memory or communication buses is inherent to the access to the processor. As only one thread is executing at a time, the resources are assumed to be available.

Several approaches have been followed to achieve temporally predictable systems, based on the temporal characterization of the threads. The release schema, the execution deadline and the expected execution time are the main characteristics of a thread. Based on that information, different scheduling algorithms, both statical and dynamical have been proposed. These algorithms, given the correct characterization of the threads, can generate schedulings in which the temporal requirements of the system are met. Along with those algorithms, different analysis techniques can prove that the generated schedulings are correct.

The release pattern of a task can be, in general, periodic or non-periodic. Periodic tasks are activated every specified amount of time known as period. On the contrary, non-periodic tasks are released as a result of an external event. If a minimum inter-arrival time can be defined for a non-periodic task, it is said to be sporadic. Otherwise, it is considered aperiodic.

Every real-time task has a deadline, which is the maximum allowed amount of time between the release of the task and the end of its execution.

Finally, the execution time is the time required for the task to be completed after its release. Thus, the execution time must be equal or less than the deadline. As the system must meet the tasks deadlines on every possible system state, the most relevant execution time of a tasks is the worst-case execution time. The worst-case execution time or WCET is the maximum time that a task may need to complete its execution in the most unfavourable situation. A system is said to be schedulable if the scheduling algorithm can prove that all tasks can meet their deadlines even assuming that all tasks consume their worst-case execution time.

The increasing hardware complexity and the interaction with other tasks make the measurement of the worst-case executing time a non-trivial issue. However, there are mature approaches for monocore systems to achieve the acquisition of WCET values. These approaches can be divided into static or dynamic. Static WCET measurements are calculated by analysing the machine code to be executed with a model of the processor, finding the required cycles to execute it.

On the other hand, dynamic WCET measurements are obtained by executing several times the code and gathering time stamps in relevant sections.

Some of the engineering systems controlled by real-time systems are regulated by different legal artefacts. This is the case of avionics, in which the computing part of the system is addressed by the DO-178C RTCA Standard for the certification of commercial airplanes. Other industries have equivalent regulations as ISO 26262 for road vehicles, IEC 61508 / EN 50129 and IEC 62279 for trains or IEC 61513 for nuclear power plants. These kind of standards require that the design can prove that the system will meet its temporal requirements (among others) strictly at any time for those tasks requiring to do so.

The real-time systems state of the art, regarding monoprocessor systems, can be said to be mature. A rich environment of theory and design tools and approaches, widely used and proven supports the development and operation of high-criticality real-time systems.

In the last years, two main issues have arisen posing new challenges to the real-time systems state of the art: on the one hand, the evolution of the controlled systems has been increasingly demanding more computing capacity. On the other hand, the computing hardware industry has been moving to multicore processors in which more than one thread can be executing in parallel.

As stated before, real-time systems are commonly embedded in other engineering systems, such as airplanes, cars, mobile phones, appliances or industrial robots, among others. The computing requirements of those systems have shown a huge increment in the last decades: industrial processes controlled by robots have become faster and more accurate, mobile phones now support thousands of new applications, and cars and airplanes are now mainly electronically controlled and providing other services such as navigation and entertainment.

Those new applications do not only require more computing capacity, but also introduce processes of different degree of relevance in the same system. For example, the ABS system of a car is more important than the navigation system. They are applications of different criticality. Systems with applications of different criticality are known as mixed-criticality systems.

Regarding the evolution of processor technologies, by the end the 1980 decade the first processors with two cores appeared manufactured by Rockwell International. By that time, the industry approach to design faster processors was focused on increasing the integration and the clock speed. It was not until the beginning of the 1990 decade when this approach began to encounter problems due to heat dissipation. Since then, the market focused on increasing the computing capacity of the processors by embedding more than one computing unit in a single processor chip. In the early 2000s the main manufacturers such as Intel or AMD began commercializing their first multicore processors which have now become the vast majority in the market. This general adoption of the multicore approach has become a challenge for the real-time systems environment, as it is being forced to move to this kind of processors, due to the scarcity of monocore processors and the overcosts associated to that lack of available units.

The mentioned changes have opened a wide range of topics in which the classic approach to the real-time systems design, analysis and implementation have to be updated:

- Hardware architecture of multicore real-time systems: there are several different approaches to the hardware implementation of multicores. One subdivision can be made between homogeneous and heterogeneous multicores. Homogeneous multicore architectures are multicore processors in which all the cores are identical, while heterogeneous processors show different core implementations on the same chip. Furthermore, cores may have different degree of coupling: cores are highly coupled when they share some core units or components supporting multithreading on a single core i.e. two threads can be executing in parallel in the same core as long as both do not require a shared component at the same time. This technique is known as simultaneous multithreading (SMT). Less coupled cores share the memory address space and optionally cache memory, implementing shared memory communications. Finally loosely coupled multiprocessors are communicated via communication buses using message passing policies.

Either way, each approach has different implications in real-time systems both in logical and temporal behaviour.

- Software architecture of multicore real-time systems: the parallel execution on more than one core at a time has several implications on the software design and implementation:
  - Isolation between applications and criticality levels: one of the main issues when executing more than one thread at a time in a system is to ensure that they do not interfere in the execution of other threads, either spatially or temporally.
    - Spatial isolation means that each application does not interfere in the memory address space of other applications.
    - Temporal isolation means that each application does not compromise the temporal requirements of other applications.

    Both kinds of isolation have to be kept at any time, even in case of an error or malfunction in an application.
  - Scheduling and WCET problem. New scheduling algorithms are required to support and maximize the benefit of the parallel execution of threads. The parallel execution itself can be seen as an improvement to the schedulability and performance of the system, as the computing capacity is multiplied

  by the number of cores. However, there are other resources to be shared in addition to processors: memory, communication buses and devices, etc. Unlike in monocore processors, in multicore processors, execution times are highly influenced by resource sharing: if two or more threads are trying to use the same resource at a time, only one will be able to do so, and the rest will have to wait until the resource is freed. These waiting times pose an overhead in comparison to monocore processing times. An efficient resource sharing policy is required to bound this overhead and provide tighter worst-case execution times.

  – Core allocation and migration of tasks: related to the scheduling problem, there is the allocation problem. As there are more than one core in the system, an efficient algorithm to dynamically or statically allocate applications to cores is required. If the algorithm is dynamic, i.e. the core in which application is executed is decided at run time, the implications of migrating a task from one core to another are to be studied.

- Moving applications to multicore: last but not least, there is a cost to consider when moving applications to multicore. As stated before, real-time and embedded applications often require to be qualified or certified. The process to qualify or certify an application is both time and resource consuming and is required to be reconducted if the application is changed. Thus, systems must be designed to avoid retaking again qualification or certification efforts.

1.1 Goals

The main goal of the present work is to contribute to the evolution of real-time systems by exploring a novel approach to multicore systems. As stated before, there is a wide set of challenges to be addressed in order to successfully support the design, development and operation of strict real-time systems based on multicore platforms. Furthermore, the concept of multicore platform is a wide concept, with different architecture proposals. Similarly, as will be reasoned along this document, approaches to real-time systems are highly diverse. As a result, this work is focused on a specific area of those mentioned above, which is the design of a software architecture capable of supporting the new generation of real-time systems.

To achieve such a main goal, a set of different steps have been identified:

- To study the approaches followed from the hardware and software points of view for the development of real-time systems based on monocore architectures.
- To analyse the current trends and different approaches followed on real-time systems based on multicore architectures.
- To identify the main characteristics of successful monocore-based approaches that should be moved to multicore-based systems.
- To identify the requirements of next-generation of real-time systems.
- To contribute on the definition of a scheduling approach aligned with the identified requirements suitable for hard real-time systems, focusing especially on data sharing protocols.

- To evaluate the approach by comparing the outcomes of applying the proposed design with those of using current state of the art approaches.

- To validate the approach by applying the proposed design in a representative application, demonstrating its suitability to be adopted into industrial practice.

1.2 Organization of the document

The rest of the document is organized as follows: chapter 2 reviews the basic real-time theory and the main approaches followed by successful approaches to strict real-time systems on monocore architectures. Chapter 3 presents the current trends on multicore architectures and real-time approaches to them, focusing on those relevant to the contributions of this thesis. In chapter 4, a resource sharing protocol supporting the design of hard real-time multicore systems is presented. The protocol is extended and improved with the contributions thoroughly described in chapter 5. The protocol is then validated and evaluated in chapter 6 where the proposed approach is applied to the UPMSat-2 satellite design, along with other state of the art techniques. Finally, conclusions and future work lines are presented in chapter 7.

Chapter 2

Background

2.1 Real-Time Systems

Several definitions for real-time systems have been proposed over the years, all of them emphasizing on the essential nature of real-time systems: the specific requirements regarding response times of these systems. One of the most accepted definitions was proposed by Young [131]: "Any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period". The Oxford Dictionary of Computing also highlights temporal requirements of the system [49]: "Any system in which the time at which output is produced is significant". Finally, Randell et al. offer a slightly different point of view [110]: "A real-time system is a system that is required to react to stimuli from the environment (including the passage of physical time) within time intervals dictated by the environment". From these definitions, it can be noted that real-time systems do not only require a correct response from the logic point of view but also a response time meeting the temporal requirements of the system.

Commonly, real-time systems are in charge of controlling and monitoring processes from other engineering fields than computing. Devices requiring any information processing include nowadays one or several *embedded* processors, constituting what are known as real-time embedded systems. Real-time embedded systems include cars, airplanes, industrial robots, mobile phones, appliances or surveillance systems, among others. This kind of systems represent the vast majority of deployed processors around the world and have different characteristics than those on personal computers. In general, embedded systems have lower computing and storage capacity, as well as reduced electric supply. Also, I/O devices are very varied and specific for this kind of systems. For these reasons the design and implementation methods for real-time systems are focused on efficiency, robustness and time predictability.

It can be said that, in general, in all systems the response time is relevant. Users of any system like having as lower response times as possible. What makes real-time systems different are the consequences of not meeting the temporal requirements. In that sense, real-time systems are classified into three main groups depending on how strict are those requirements [35]:

- Hard real-time systems are those in which the temporal requirements have to be always met. This kind of systems often are in charge of controlling engineering systems managing valuable goods or those in which an error may compromise human lives. For example, ABS systems, present in airplanes and cars monitor and actuate, if necessary, between 50 and 100 times each second while breaking. In case of a late response of the system, wheels would lock up and skid thus loosing the required tractive contact to respond to steering wheel actions. That situation might result in an accident with victims.
- Soft real-time systems are those in which a deadline may be missed occasionally. Although reducing the quality of the system or service offered, the consequences are not considered catas-

trophic. Soft real-time systems can be subdivided in two categories:

- Firm real-time systems are those in which a late response of the system is not useful and is just discarded. In the case of firm real-time systems, the consequences of missing a deadline are less severe than in hard real-time systems, provided that the frequency is bounded to a certain limit. For example, if a TV decoder provides a frame later than when it is to be displayed, the frame is invalid (it is not displayed). In that case the user may perceive that the image frozen for a moment, reducing its user experience. If this happens too frequently the user may inquiry the manufacturer, thus affecting its reputation.
- Flexible real-time systems are those in which a late response still has a value that decreases with time. For example, an automatic tap which is in charge of stopping a flow of some liquid after a period of time (for example after filling a bottle). If the tap does not close in time, some liquid may be shed, increasingly loosing value.

2.2 Real-Time Systems Scheduling

2.2.1 Overview

As it has been stated before, the most relevant characteristics of real-time systems are the temporal requirements. In contrast to general purpose systems, in which the order of execution of different jobs is irrelevant, provided that concurrency issues are properly handled, in real-time system the order of execution matters. Given a set of tasks to be executed, probably only a subset of the full range of different execution orders may allow all of them to met their temporal requirements. The activity of generating an execution order meeting all the temporal requirements of the system is known as scheduling. Scheduling schemes usually provide two related features:

- An algorithm to order the execution of the different tasks.
- Means to demonstrate that the generated scheduling meets the temporal requirements even in the worst case.

For both features, each task in the system needs to be characterized. For each task several parameters are required. The most relevant one, that subdivides tasks into two main groups, is the activation policy. Depending on the activation policy, tasks can be:

- If the task is activated every given amount of time the task is said to be periodic. The amount of time between subsequent activations is known as period.
- If the task is activated as a result of an external stimulus, the task is said to be non-periodic. If the task can be activated at any given time, the task is said to be aperiodic. If the task cannot be activated before a minimum time interval has elapsed following its last activation is said to be sporadic. This amount of time between activations is known as inter-arrival time.

The second parameter that characterizes a task is the deadline. The deadline is the time by which the execution associated to a task must be completed.

The third parameter is the execution time. The execution time is the time that the task actually takes to complete the operation occupying the CPU, including the time spent executing run-time or system services on its behalf [9]. As, in general, in real-time systems

Symbol	Description
τ_i	Analysed task
r_i	Release time
c_i	Completion time
d_i	Absolute deadline
J_i	Activation jitter
C_i	Execution time
R_i	Response time
D_i	Relative deadline
T_i	Period

Table 2.1: Basic notation.

temporal requirements are to be met under all conditions, the worst-case execution time or WCET is generally considered for scheduling purposes.

Given these three parameters of a task, other concepts are to be introduced:

- Activation jitter: time deviation from a pure periodic activation pattern.
- Interference: as will be explained later, some scheduling schemas allow tasks to preempt others and expel them from the CPU. Thus, the preempted task does not execute from start to the end. The amount of time that the preemted task is not executing after its activation is known as interference.
- Response time: the response time is the amount of time between the task activation and the moment in which it terminates its execution, including jitters and interferences. Thus, the aim of scheduling algorithms is to ensure that a task response time never exceeds its deadline.

The constraints enforced to tasks with regard to the different characteristics previously outlined define a task model. The task model

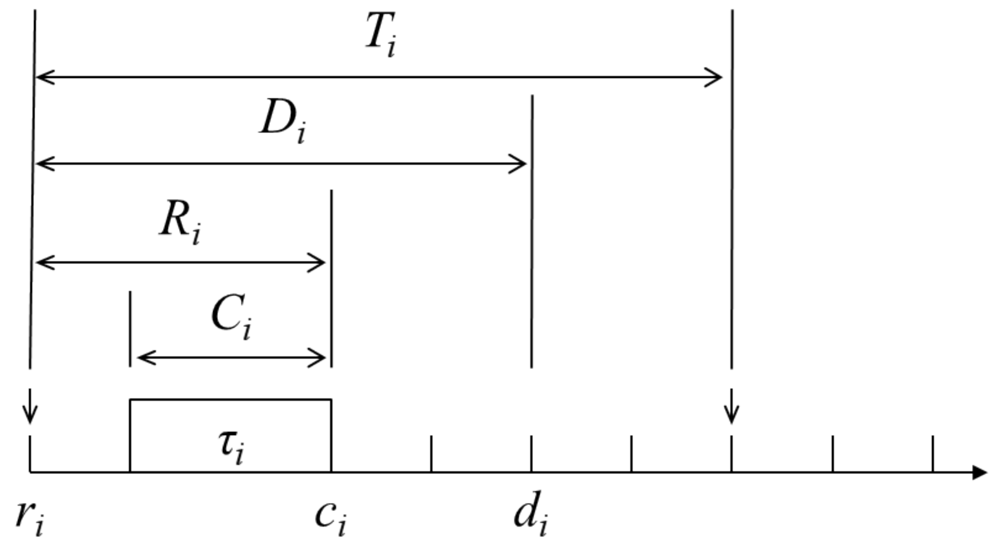


Figure 2.1: Basic task parameters.

that will be assumed during this thesis unless otherwise stated is the classic sporadic task model [97]. Under this task model, a real-time system is comprised by a set of tasks that are repeatedly invoked. These invocations are due to external events such as a timer expiration or a device interrupt. Tasks are constrained to have a periodic or sporadic activation pattern, i.e there is a minimum separation between activations of a task that is known at design time. Tasks upon invocation release a job that processes the triggering event. Jobs are pending from the time they are released until the time in which they are completed. While pending a job can be ready, if it is available for execution, or suspended, when it cannot be scheduled. Only one job of a task can be scheduled at a time, i.e. a job is not considered ready until the previous job released by the same task has completed.

2.2.2 Static Scheduling

Probably the most intuitive way of constructing a real-time system is by using a static scheduler. In this kind of schedulers, the execution sequence of the required procedures is generated at design time. The schedule is made up of the repeated execution of one or more cycles. This scheme is known as a cyclic executive [15]. The cyclic executive is then a table of procedure calls, known as major cycle, which is organized in minor cycles of fixed duration depending on the periods of the procedures to be executed. Edges of minor cycles are signalised by clock interrupts. When all the minor cycles have been executed, a major cycle has been completed and the execution starts again from the first minor cycle.

The construction of this scheduling algorithm ensures that, if a cyclic executive is feasible given the temporal requirements of the different jobs, no further schedulability analysis is required, as no concurrency or interference is possible following this approach. Thus, this scheme is said to be “proven by construction”. Due to this determinism in the execution, static scheduling is widely used in real-time systems of high integrity [132, 133].

However, the cyclic executive presents several drawbacks:

- All periods must be multiple of the minor cycle duration.
- Tasks with long periods are hard to be incorporated. A common approach is to define the hyperperiod (length of a major cycle) as the least common multiple of tasks periods.
- Tasks with long execution times have to be split in several procedure calls.
- Aperiodic tasks require complex scheduling artefacts to be included.

- Constructing the cyclic executive is an NP-hard problem [65]. Thus, this approach is not feasible for systems with a considerable number of tasks.

Due to these drawbacks and with the aim of improving the utilization of the real-time system, several dynamic scheduling approaches have been proposed.

2.2.3 Dynamic Scheduling

In contrast with static scheduling, in dynamic scheduling approaches the execution order is decided at runtime. To do this, several criteria have to be defined to decide the execution order and to prove that temporal requirements will be met.

The first criterion involved in dynamic scheduling is the concept of task states and lifecycle. Depending on the scheduling pattern, tasks can be in different states, but, in general, they can be reduced to the following:

- Running: the task is actually executing on a CPU.
- Runnable: the task is ready to be executed.
- Suspended: the task is not ready to be executed. It can be waiting for a timing event, e.g. a periodic task waiting for its next activation time, or waiting for a non-timing event, such an external stimulus, typical in sporadic tasks.
- Terminated: the task has ended its execution and will not resume executing. In general, in real-time systems, tasks are not allowed to terminate due to safety reasons.

A second criterion is required to decide at each time which of the runnable tasks is to be executed. Without further information, it would be an arbitrary decision. In order to provide the required predictability and to ensure the proper timing behaviour, some priority has to be given to one task above others. Thus, priority is the second criterion when deciding which task is to be executed. Priorities among tasks can be decided at design time or at runtime, as will be detailed later. Then, the scheduler is in charge of ensuring that the CPU is always allocated to the runnable task with higher priority in the system.

Finally, a third criterion mixes both previous concepts of state and priority. In a priority based scheduling scheme two possibilities arise: that at any time the processor is occupied by the runnable task with higher priority, or that the runnable task with highest priority only starts its execution after the CPU is released by the executing task. In the first case, the scheduling scheme is said to be preemptive. In the second case the system is said to be non-preemptive. In the first case, the scheduler has to decide which task is to be executed each time a task becomes runnable and when the running tasks concludes its execution, while in the second case, the scheduler only decides which tasks may begin or resume their execution after the running task has ended its execution. In preemptive systems, it is thus not guaranteed that a task, once it has started its execution, will continue executing until it is completed, as it can be preempted. This has an influence on the execution time of the preempted task, which is known as interference, as mentioned before.

2.2.4 Fixed Priority Scheduling

In a fixed priority scheme, tasks are assigned a base priority, which remains constant, at design time. This priority can be assigned applying different criteria.

Fixed priority scheduling (FPS) was first analysed by Liu and Layland in [95]. In that work they also propose a priority assignment protocol, known as Rate Monotonic Scheduling (RMS). In this protocol, priorities are assigned inversely proportional to the periods of tasks i.e. the task with higher frequency is assigned the highest priority in the system. The RMS protocol is considered to be optimal, i.e. if a set of tasks is schedulable in a priority scheme, it will be also schedulable in RMS.

The way to proof that the system is schedulable, or that the scheduling is feasible, differs depending on the scheduling pattern used. In the case of Rate Monotonic Scheduling, the way to prove the system schedulability proposed in [95] is to demonstrate that the task set keeps the processor utilization below a certain threshold. Equation 2.1 expresses a sufficient condition:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq N \cdot (2^{1/N} - 1) \tag{2.1}$$

where N is the number of tasks in the system, C is the worst-case execution time of a given task and T is the period of the given task. As can be seen in equation 2.1, the right part of the condition decreases as the value of N (number of tasks in the system) increases. It can be analytically proved that the equation asymptotically approaches 0.69. Thus, any set of tasks keeping the processor utilization below 0.69 would be schedulable using RMS, regardless of the number of tasks in the system.

$$\lim_{N \to \infty} [N \cdot (2^{1/N} - 1)] = \ln 2 \simeq 0.69 \tag{2.2}$$

Another criterion to assign priorities may be the importance or criticality of tasks. It is ensured that, in a fixed priority protocol, tasks with lower priorities are more affected by overloading. This means that, if any task misses its deadline it will be first the task with a low priority. Thus, if priorities are assigned with regard of their criticality, tasks less important will be missing their deadlines first. However, this criterion to assign priorities requires a more complex scheduling analysis, that will be introduced later, to illustrate other concepts as well.

2.2.5 Dynamic Priority Scheduling

In contrast to fixed priority scheduling, in dynamic priority scheduling protocols, task priorities are not assigned at design time but at runtime. To do so different criteria may be used. From them, the most wide spread example is Earliest Deadline First (EDF) scheduling, also proposed in [95]. As its name states, in EDF, the task with the earliest deadline has the greater priority in the system.

EDF has been also proven to be optimal. Furthermore, EDF leads to more processor utilization as, demonstrated by [95], its scheduling analysis is reduced to satisfy equation 2.3.

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{2.3}$$

As can be derived from equation 2.3, the only requirement for the task set to be schedulable is that the total processor utilization is below 1. This poses a significant difference with the sufficient condition of 0.69 for FPS. While for RMS only a subset of task sets can reach a utilization of 1, under EDF this is a sufficient condition for every task set. However, EDF has some drawbacks to consider, being the most important one the complexity and overhead of the scheduler. In EDF the scheduler has to calculate which task has the earliest deadline at runtime, requiring an extra processor utilization for this computation. Also EDF is considered less safe, as in case of system overload situations, the behaviour is unpredictable i.e. it cannot be stated which task may miss its deadline. Furthermore, EDF can lead to domino effects, in which many tasks miss their deadline consecutively.

2.2.6 Shared Resources

In previous sections, task models and solutions reviewed do not consider the intrinsic effect of task synchronization in the scheduling process. While isolated task models provide good foundations as first step approaches to task scheduling, realistic systems as those mentioned in the introduction have a complexity level which is difficult to address without task synchronization mechanisms. Tasks in real systems do share logical and physical resources, to which access needs to be arbitrated. By using common synchronization mechanisms as semaphores, monitors or rendezvous coordination, the commonly required mutual exclusion access to shared resources can be achieved. However, this is at the cost of not only complicating the schedulability analysis, but also adding new undesirable system states.

Shared resources are logical (commonly data structures) or physical (as peripheral) elements that are used by more than one task in the system, but have requirements on the concurrent access to them. Although there are exceptions, shared resources require to be accessed in mutual exclusion. This means that only one task should be performing an access to a resource at a time. No other task should access the resource until the accessing task has finished or completed its access to the resource. The section of code associated with mutual exclusion requirements is known as critical section. To implement the mutual exclusion, the most exercised approaches use the notion of a lock: a lock is a logical element that gives the holder the right to access the shared resource. As it can be only held by one task, the mutual exclusion is assured. While some protocols allow some tasks to take the lock from others (in the same way some tasks can preempt others, getting the right to use the processor), the common approach is to force other tasks to wait until the lock is released by the holding task to get it.

As a result of the above, it can happen that, regardless of the paradigm used to grant priorities to tasks, a higher-priority task has to wait for a lower-priority task to complete its access to a shared resource. This situation is known as priority inversion [90, 89] in fixed priority scheduling and deadline inversion in EDF schemes. Higher-priority tasks are then said to be blocked by lower-priority tasks. If no extra mechanism than those presented above are used, this can lead to uncontrolled priority inversion situations, with indefinite periods of blocking.

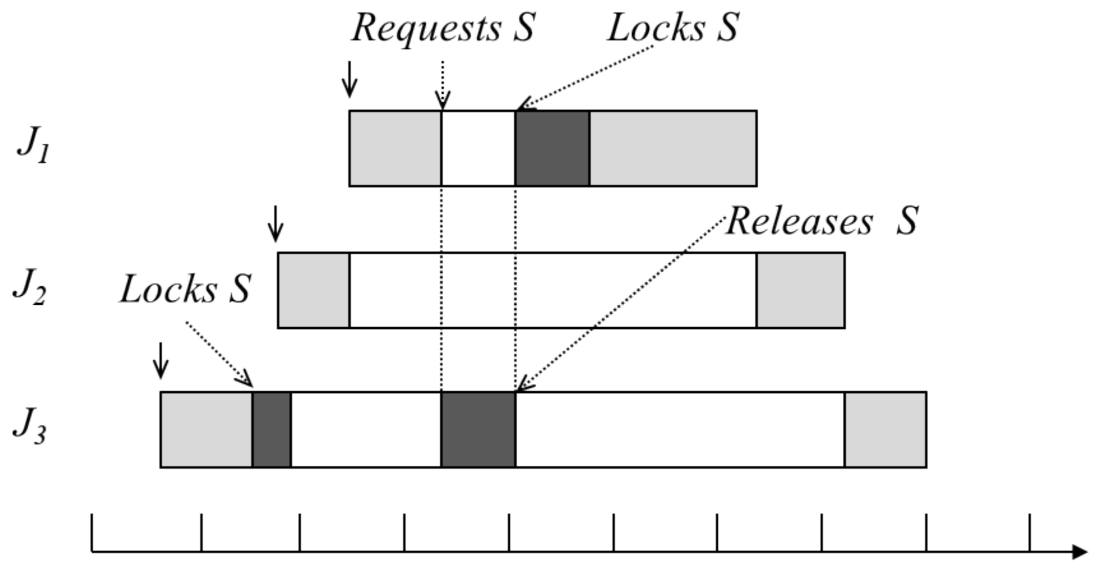
It has been demonstrated that proving the schedulability of periodic task sets synchronized with semaphores to enforce mutual exclusion is an NP-hard problem [97]. In fact, only heuristic approaches developed in [97], with non-preemptive accesses to resources have been identified as providing feasible schedules. Heuristic approaches were further investigated in [140, 139, 109].

Priority Inheritance Protocol

In order to bound the blocking times different protocols have been proposed. One approach widely studied and used is the Priority Inheritance Protocol [115] (PIP). This protocol started a successful trend of protocols in which the conjugation of dynamic priorities and basic synchronization primitives (binary semaphores) provide a set of determinant scheduling properties.

The intuition behind priority inheritance protocols is that, if a lower-priority task blocks a higher-priority task, as a result of having locked a shared resource, it ignores its own priority and executes at a higher-priority in order to achieve a faster and, most important, decidable releasing time of the resource. This new priority is known

Figure 2.2: Priority inheritance example.



as active priority, and is equal to the highest priority among the tasks being blocked.

In the example shown in figure 2.2, when J_1 is blocked by J_3, the latter one increases its priority to J_1 priority, i.e. it inherits J_1 priority. In this way, J_3 can get to execute, release S lock and decrease its priority to its base priority, unblocking J_1. When J_1 is completed, J_2 and J_3 can be subsequently be redispatched to completion.

This behaviour provides systems implementing the basic priority inheritance protocol with a set of fundamental properties with regard to scheduling:

- Lower-priority tasks can only block higher-priority tasks as a result of having locked a shared resource. This blocking, under PIP can be categorized as:
  - Direct, if the higher-priority task is trying to access the locked resource.
  - Push-through, if the higher-priority task is blocked as a result

of a lower-priority task having raised its priority.

- Each lower-priority task can block a higher-priority task as long as the duration of one critical section.

- Each task can only be blocked as many times as shared resources it accesses.

However, the basic priority inheritance protocol also has two main problems. First, the requirement to support nested resources was identified, i.e. allow resources to require other resources to complete. While the protocol requires that the nesting of two resources r_1 (outer) and r_2 (inner) is done so that $r_2 \subset r_1$, the priority assignation cannot prevent by itself the formation of deadlocks. This is said in [115] to be solved by imposing a total ordering on the semaphore accesses. The second problem is that, while the blocking can be bounded as explained, it can be substantially long, as a result of blocking chains. To limit this blocking term, the Priority Ceiling Protocol was proposed.

Priority Ceiling Protocol

Although the priority inheritance protocol reduces the number of times a task can be blocked, this reduction may not be enough in terms of efficiency and processor utilization. In response, another set of protocols arose, the Priority Ceiling Protocols [115] (PCP). With a priority ceiling protocol, a task can only be blocked once during its execution, also preventing transitive blockings (task a blocks task b, which also blocks task c and so on) and deadlocks. Furthermore, mutual exclusion is also ensured by the protocol itself, as will be explained.

In priority ceiling protocols, priorities are not only a property of tasks but also of resources. Tasks have an initial default priority, and resources acquire the maximum priority of the tasks that use it. This priority is known as ceiling priority. The way tasks update dynamically their priorities and are allowed to lock resources differentiate the existing approaches.

In the original enunciation of the priority ceiling protocol [115], tasks execute at their assigned base priority. When trying to access a resource, the lock is granted if, and only if, the lock is free and the task priority is higher than the priority ceiling of all the currently locked resources in the system. If as a result of having locked a resource a lower-priority task blocks a higher-priority task, it inherits the ceiling priority of the resource (in contrast to PIP, where only the blocked task priority was inherited).

As a result, the priority ceiling protocol achieves an increased performance with comparison to PIP due to the following properties:

- The maximum amount of blocking suffered by a task is equal to the longest critical section accessed by a lower and higher-priority task.
- Transitive blocking is prevented.
- Deadlocks are prevented.

However, a new form of blocking is introduced: ceiling blocking. A task, with the highest priority of those runnable at a time (and thus executing) can be blocked due to not being able to lock a free shared resource.

With the mentioned properties and blocking terms, a sufficient analysis can be provided to PCP as long as the n periodic tasks in the system are assigned priorities according to the rate-monotonic algorithm. A PCP system is deemed to be schedulable if the following condition holds:

$$\forall i, 1 \leq i \leq n, \quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \leq i \cdot (2^{1/i} - 1) \tag{2.4}$$

where B_i is the maximum blocking τ_i can suffer, in form of direct blocking, push-through blocking or ceiling blocking. The priority influence on the analysis is reflected on the order in which tasks are analysed. Higher-priority tasks are analysed first, so they have to cope with less system utilization already allocated. This equation is an extension of 2.1 including the notions of priorities and shared resources in the form of blocking.

Immediate Priority Ceiling Protocol

Another approach to ceiling priority protocols is the immediate priority ceiling protocol (IPCP). Under this protocol tasks immediately update their active priority to the ceiling priority at locking time. In this way, it is ensured that, once it has locked the resource, the task will continue executing until it frees the resource or a task with higher priority, which does not share the resource, is released. Thus, blocking only occurs prior the task actually starts its execution. Furthermore ceiling blocking is eliminated, as tasks under IPCP have all required shared resources available once they start their execution. This highly reduces the number of context switches and hence the execution time. All the other relevant PCP properties as deadlock free execution and transitive blocking prevention are maintained.

To analyse these systems, a more complete technique is available. The Response Time Analysis method, developed in late 80's [75, 82, 11] provides a necessary and sufficient condition for a scheduling to be feasible, in which the result is not only binary (schedulable or not), but also the worst-case response time of each task can be calculated. If the response time of all tasks is less or equal than their respective deadline, the system is schedulable. Equation 2.5 shows how this technique is applied to IPCP.

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

The response time of each task (R_i) is the result of adding the worst-case execution time (C_i) to the maximum blocking (B_i) and interference (I_i) suffered in a worst-case activation. The interference is calculated in a similar way as in equation 2.4. For each higher-priority task $hp(i)$ it can preempt the analysed task τ_i as many times as the response time of τ_i divided by the period of the higher-priority task. On each preemption, the interference can be has high as the worst-case execution time of the preempting task (C_j) as shown in equation 2.6.

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

Stack Resource Policy

Both priority inheritance, priority ceiling and immediate priority ceiling protocols presented before were focused on fixed priority scheduling schemes. All those protocols have also a variant for dynamic priorities schemes, but those are not as efficient as for fixed priorities schemes [18].

For EDF, the most widely accepted is the Stack Resource Policy (SRP) proposed in [13, 14]. This policy is closely related to the immediate ceiling priority protocol defined for fixed priority protocols. Under SRP, each task is assigned a preemption level related to the relative deadline of the task. The shorter the deadline, the higher the preemption level assigned. This is the static priority for the task. At runtime, resources are assigned a ceiling value equal to the maximum preemption level of the tasks that use the resource.

Then, when a task is released it can only preempt the executing task if its absolute deadline is shorter (higher overall priority) and its preemption level is higher than the highest ceiling of the locked resources (blocking priority).

The performance of SRP is equal to immediate ceiling priority for fixed priority schemes. Thus, tasks can only suffer a single block at release time (without actually executing) and are free of deadlocks. When using EDF with tasks deadline equal or lower their periods, the schedulability analysis is done based on utilization factor with the following equation:

$$\forall k, 1 \leq i \leq n, \quad (\sum_{i=1}^{k} \frac{C_i}{D_i}) + \frac{B_k}{D_k} \leq 1 \tag{2.7}$$

again similar to the form of PCP analysis in equation 2.4, with interference and blocking being accounted.

2.2.7 Ravenscar Profile

The previously presented protocols are generic in the sense that define the main characteristics of a task model, not specifically aligned with any runtime or implementation. On the contrary, the Ravenscar profile [31] is a set of rules for the implementation of strict real-time systems using the Ada [9] programming language.

Strictly, the Ravenscar profile defines a subset of the Ada programming language that enables the construction of analysable systems [34]. This subset can be enforced by a single pragma since the Ada 2005 language revision [79, D.13.1], being equivalent to the set of pragmas listed in annex A. The system conformity with respect to the profile can be mostly checked at compilation time reducing validation costs and increasing the reliability of the approach.

The Ravenscar tasking model comprises both periodic and sporadic tasks that are dispatched following a FIFO within priorities policy. For sporadic tasks a minimum inter-arrival time is required for the timing analysis, while for periodic tasks a specific period between activations is required. Tasks cannot be created dynamically or terminated and can only present one suspension point.

Task communication and synchronization is done by means of the Ada approach to shared resources as considered in this document: protected objects (PO). Access to protected objects under Ravenscar systems is ruled by a ceiling locking policy, which, as defined in the Ada standard, is equivalent to that of the Immediate Priority Ceiling Protocol described above.

Ada protected objects can be accessed via three different kind of operations: functions, procedures and entries. Functions are read-only accesses, being protected and entry operations read-write operations and thus required to be executed under mutual exclusion. Entry operations are guarded by a boolean barrier. The barrier condition has to evaluate to true to allow access to the resource, being the calling tasks otherwise suspended on a queue associated to the entry. If any task is enqueued on a protected object entry, barriers are reevaluated at the end of read-write accesses to that PO. To ensure timing predictability on PO, the number of entries per PO is limited to one, and only one task can be queued at a time on that entry. As a result if the barrier condition is evaluated to true only one task can be re-activated. Furthermore, potentially blocking operations are not allowed inside protected operations and barriers are restricted to simple conditions (cannot include a further operation call). Other sources of time non-determinism, as those associated to memory usage, such as dynamic memory allocation are also forbidden in Ravenscar systems.

The adherence to these restrictions enables the implementation of Ada applications with a task model equivalent to that considered in the IPCP, with the major outcome of being analysable using equation 2.6.

Chapter 3

Multiprocessor Real Time Systems

As thoroughly reviewed in the introduction, there are a set of factors that have been inevitably pushing the real-time community towards the adoption of multicore systems. Among them, the increased complexity of controlled systems and the manufacturing limits on chip integration due to power consumption and heat dissipation [88] are considered key factors.

3.1 Challenges

The movement towards the adoption of multicore architectures brings forward a set of challenges. While some of them are new, some others have been already solved for uniprocessor systems, but require new solutions. In that sense, the complexity of the problem was highlighted as early as in 1969 by Liu [96] when he stated that only a small subset of the results on single processors could be applied to multiprocessor systems, as even executing a task in a system with free processors pose a much more difficult scheduling problem.

In that sense, as reviewed in chapter 1, the main problem in uniprocessor systems was to wisely select the task to be allowed to execute, in order to comply with the system timing requirements. This is known as the *priority* problem. With the advent of multicore systems, a new problem appears: the *allocation* problem.

- *Allocation*: the allocation problem is the need to determine in which processor a task is to be executed. This decision can be done offline, by assigning at design time one or more processors for a task to execute in, or online, by the scheduling algorithm deciding at run time where to allocate each task.

  When tasks can only execute on one processor (typically designated at design time), the system is said to be fully partitioned. If, on a partitioned system some of the tasks are allowed to execute on more than one processor, it said to be a semi-partitioned system. Finally, systems where all task are allowed to execute on more than one processor (normally decided at run time) are said to be global.

  The action of changing the allocation of a task from one processor to another is known as migration. Different task models impose different restrictions. The most restrictive one is the no migration policy, which, in fact, generates strictly fully partitioned systems, as stated above. A less restrictive approach is to allow task-level migrations, in which jobs of a task can be released on different processors, but when a job its released it is not allowed to migrate for the rest of the activation. Finally, job level migration systems allow jobs to migrate as long as a job is not executed on more than one processor at a time. It is important to note that, when migrations are allowed, there is always an associated overhead due to the required context switches, run-queue updates and cache invalidation among others [30] that should be taken into account in the timing analysis.

- *Priority*: in terms of priorities, however, there are no completely new paradigms in those proposals for multiprocessor systems that have shown to be adoptable in practice. Priorities can be fixed at design time at task or job level, or can be dynamically calculated at runtime. This does not mean that multicore systems do not pose new challenges in terms of priority assignment and its influence. On the contrary, the scheduling of tasks can be, as shown in the following sections, influenced not only by the tasks (or jobs) allocated on the same processors and their priorities, but also by the tasks/jobs allocated on other processors and their relative priorities [8].

These two main challenges are aimed at finding a suitable solution to the following, multiprocessor-intrinsic difficulties:

- Real parallelism: while the concurrent execution of subprograms was accepted decades ago for safety-critical systems, the parallel execution (more than one subprogram making progress at the same time) implies a higher degree of complexity. Even if those subprograms are completely independent, a set of system level data structures such as run-queues are to be maintained.
- Task models: as identified in [50] and will be highlighted during this chapter, the most relevant works on multiprocessor systems for hard real-time systems address supporting the conventional periodic or sporadic task models developed for uniprocessor systems. It is clear that, with the incorporation of more than one processing units, other computation paradigms can be explored. In particular, the parallel execution of elements belonging to the same task (e.g. by executing more than one job at a time or splitting one job computation into more than one processor) is an active research topic, based on initial works in late 00's [45, 46, 55]. The debate on the evolution of high-integrity aimed programming languages is not on the need to support parallelism but on how to do it [102, 104, 103, 111, 124].

- Hardware resources sharing: as identified in the previous chapter, one of the main challenges of multicore and multiprocessor systems is the physical resources sharing involved. From the high levels of hardware coupling present in SMT systems to loosely coupled multiprocessors, all systems share a set of resources which cannot be provided solely to one executing unit. This has been proved to have a high influence on task execution times [105, 40, 129].

- Logic resources sharing: as highlighted in section 2.2.6, task synchronization by logic resource sharing is fundamental for building complex real-time systems. In that section, different examples of scheduling algorithms were presented in which the coordination to access logic shared resources was addressed by granting the right to execute to the envisaged task to access the resource. On a system where more than one task can be executing in parallel this approach is no longer valid by itself to ensure mutual exclusive access to shared resources.

3.2 Multicore Real Time Systems Scheduling

Although there is a set of new problems to overcome associated to the adoption of multiprocessor systems for real-time systems, and besides Liu statement, there is a preference for applying classic scheduling policies to multiprocessor systems. In that sense, as mentioned before, one classic approach to multiprocessors is to generate a set of isolated partitions, trying to applying a divide and conquer strategy, reducing the scheduling problem to a set of single processor scheduling problems.

While fully partitioned approaches may seem desirable, as they allow reusing classic scheduling solutions, they also offer several drawbacks. Among them, the main criticism to fully partitioned systems comes from the fact that it generates systems that are not *work conserving*. Work conserving systems are those in which no computing unit is allowed to remain idle as long as there is a ready job not executing in the system.

Opposite to this approach is global scheduling. In globally scheduled systems, the scheduling problem considers all tasks and processors at the same time, in an integrated and complete run-queue. In general, globally scheduled systems are work conserving, although there are exceptions such as quantum scheduled systems [50], i.e. those in which scheduling decisions are only taken at predefined intervals of time known as quanta. As a result they are supposed to perform better under higher workloads.

Finally, there are a set of different approaches, considered hybrid approaches [50], where the boundaries between partitioned and globally scheduled systems are not so clear. In particular, semi-partitioned systems, virtualized and clustered systems will be addressed in this chapter.

3.2.1 Fully Partitioned Systems

As mentioned before, fully partitioned systems seem to be the way to go. They enable transforming most of the analysis and design problems of multiprocessor systems into uniprocessor scheduling problems. Then, uniprocessor systems methods can be applied for each processor in the system. In fact, this is one of the main advantages of fully partitioned systems:

- Fully partitioned systems enable the use of mature uniprocessor theory and practice, thus easing and reducing the cost of developing real-time systems.

- Fully partitioned systems simplify the management of the system: as each processor has its own run queue and associated information, the approach is scalable, in the sense that, scheduling overheads do not depend on the size of the system task set but only on the size of the processor task set. This has been demonstrated to be a key factor in practice [30].
- Migration overheads are also prevented, since they are not allowed.
- If real isolation among processors is provided, a timing failure in one processor should not affect the other processors, thus mitigating the consequences of the failure.

However, these benefits encounter also a set of non-negligible drawbacks:

- Early works as [54] lacked in some sense of practical perspective: proposals were developed on the basis of certain assumptions, mainly from the hardware perspective, not necessarily true for nowadays multiprocessors. One of the most important one is the assumed isolation among processors: as already mentioned, execution times of fully partitioned systems are affected by the execution of other processors as they share a set of unique hardware parts.
- Fully partitioned systems are, if no migration is allowed, not work conserving. This means that there can be idle processors at the same time as there are runnable tasks in other processors.
- Last, but not the least, there is a strong problem with regard to the allocation: finding an optimal allocation is known to be a NP-Hard problem [54, 65, 94]. The allocation problem for fully partitioned systems has been commonly compared to the more

general "bin-packing" problem, where a set of n items $x_1 \cdots s_n$ of different sizes $s_1 \cdots s_n$ are to be stored in a set of bins of a fixed capacity. The problem consists in determining whether there exists a certain ordering or allocation in which the set of items can be allocated in a number B of bins. It is then straightforward to consider tasks to be the items to be allocated, the tasks utilizations equal to the sizes and the available processors are the bins in which the items are to be allocated. An extensive bibliography of different approaches can be found for the bin-packing problem, many of them trying to allocate big items first, similarly to rate monotonic and deadline monotonic algorithms. The most relevant approaches, also exercised in multiprocessor real-time systems are:

- First-fit: every element is assigned to the first bin where it can be allocated. If the item cannot be allocated, it is assigned in a new bin. If all bins available are already used then the set is not allocatable following first-fit (FF) algorithm. This algorithm has been proven to asymptotically require $\frac{11}{9} \approx 1.22$ times the number of bins than an optimal allocation [80, 81] for a large number of bins, while for numbers of bins less than 4 it requires 1.5 times as much as an optimal allocation [118].
- Best-fit: every element is allocated in the bin where it will leave the less remaining capacity. If it does not fit on any bin, it is allocated on a new one unless all available bins have been used. Best-fit (BF) algorithm has the same relations with regard to an optimal allocation than first-fit [80, 81, 118].
- Worst-fit: elements, as opposed to best fit, are allocated in the bin where they will leave the maximum remaining capacity. This approach is known to behave slightly worse than first-fit and best-fit with an asymptotic higher bound

of $\frac{5}{4} = 1.25$ times the number of bins of an optimal allocation [80, 81].

From those above, although yielding slightly worst results on the utilization bound, generally worst-fit is a common approach in real-time systems. This is so because in these systems it makes more sense to evenly spread the load across the available processors, unless other considerations (as power saving policies) are taken in to account. Systems with even distributions of load are more able to cope with timing failures where tasks are unable to complete operation in the expected worst-case execution time.

MPCP

One of the first approaches to multiprocessor scheduling considering shared resources was the Multiprocessor Priority Ceiling Protocol (MPCP) [106] which, as the name suggests, was aimed to transfer the well known Priority Ceiling Protocol and its relevant properties to multiprocessors.

MPCP considers fully partitioned systems scheduled with preemptive fixed priorities, assigned following the rate monotonic algorithm. Critical sections are guarded by binary semaphores, which can be local, if they are only accessed by tasks hosted on the same processor, or global, if they are accessed from more than one processor. The task model does not support any kind of nesting between global and local resources [106, 63].

Shared resources are awarded priorities as in PCP, i.e. they have the same priority as the higher-priority task accessing them. The particularity of MPCP is that, *de facto* critical sections can only be executed on specific processors, called synchronization processors. These processors can also host tasks, executed at lower priority than any shared resource, in order to not alter the access to critical sections. Following this approach, tasks may need to "migrate" to synchronization processors to access shared resources, where the priorities given to each task and resource rules the access requests. As a result, requests are dealt in prioritized queues. With this definition, MPCP systems hold the following properties:

- Deadlock free: as tasks cannot deadlock themselves, and resources are only accessed in synchronization processors, where all related resources should be placed, and these synchronization processors are ruled by PCP, there can be no deadlocks.
- The maximum local blocking suffered by a job is $n_i^G + 1$ critical sections, where n_i^G is the number of times a job j_i accesses global resources. As jobs suspend themselves at their application processor when migrating to the synchronization processor, lower priority jobs can be executed and lock local resources, that may cause blocking to j_i when migrating back.
- A job can wait for at most one critical section of lower-priority tasks on the synchronization processor per access to a global resource. As the synchronization processor is ruled by PCP, this blocking can be considered as "arrival blocking" to the sync processor, and thus inherit this property from PCP.
- A job can wait as much as $CS_i \times \lceil \frac{T}{T_i} \rceil$ for each higher-priority job j_i during period T to execute on a synchronization processor (where CS_i is the execution time of the critical section).

The schedulability analysis is performed by applying equation 2.4 on each application processor, where the term B_i is the sum of blockings suffered due to the three previous sources explained before.

The proposal is also extended to a model in which there can be more than one synchronization processor, but then nested resources are not allowed [106].

DPCP

The Distributed Priority Ceiling Protocol [107, 108] is another approach aimed to transfer the Ceiling Protocol to multiprocessors. In this case, the task model includes a preemptive fully partitioned system with priorities assigned according to the rate-monotonic algorithm. In contrast to the model considered for MPCP, under DPCP all processors are assigned tasks, which can share local resources in PCP way, and global resources, which are guarded by semaphores locked by read-modify-write instructions. Nested resources are not supported. Lock requests not immediately satisfied are serviced by means of prioritized queues, and waiting tasks suspended. Critical sections are executed in the task host processors at the ceiling priority of the resource. The following properties can be demonstrated for DPCP:

- Deadlock free: inherited from PCP and the fact that nested resources are not allowed.
- The maximum local blocking suffered by a job is $n_i^G + 1$ critical sections, where n_i^G is the number of times a job j_i accesses global resources. Same as in MPCP, this is because each access to a globally shared resource can lead to a suspension in the local processor.
- A job can wait for at most one critical section of lower-priority tasks on the synchronization processor per access to a global resource. This follows from the prioritized queues ruling outstanding requests.

- A job can wait as much as $CS_i \times \lceil \frac{T}{T_i} \rceil$ for each higher-priority job j_i (local or remote) during period T.
- Under DPCP, local lower-priority tasks can also preempt higher-priority tasks not executing global resource critical sections, as a result of being queued for a high-priority global resource and then be signalled to execute, raising their active priority. As lower-priority tasks can queue themselves on global resources each time the analysed task is suspended, for each lower-priority task τ_k, the number of priority inversion preemptions is bounded to be $min(n_i^G, 2n_k^G)$

The resulting equation for the schedulability analysis is equation 2.4, with the B_i factor being the sum of the blocking factors mentioned above.

MSRP

The Multiprocessor Stack Resource Policy (MSRP), as defined in its original enunciation [64] is an extension of EDF + SRP protocols to multiprocessors, where tasks are statically allocated to processors. Shared resources are considered global or local depending on the allocation of tasks accessing them and only the latter are allowed to be nested. The main differences with the already studied multiprocessor protocols is that, if a request is not immediately satisfied tasks do not suspend but spin-wait until access is granted. Furthermore, under MSRP, the spin-wait and access to the resource is done in a non-preemptable fashion. Another relevant difference between MSRP and MPCP and DPCP is that outstanding requests are serviced in FIFO order. As a result, MSRP inherits the following properties from SRP:

- Once a task starts executing it cannot be locally blocked by lower-priority tasks. It can only be preempted by higher-priority tasks.

- The time a task can be spin-waiting for access to a global resource r^j is bounded by the sum of worst-case accesses w^j_{ih} from remote processors p where there is at least one task τ_i accessing the resource:
$$spin(r^j, P_k) = \sum_{p \in \{P - P_k\}} \max_{\tau_i \in T_p, \forall h} w^j_{ih} \tag{3.1}$$
- The maximum local blocking is bounded to one critical section (happening before the task starts executing). This blocking is the maximum between that caused by a local lower-priority task accessing a local or a global resource, i.e. $B_i = max(B_i^{local}, B_i^{global})$ where B_i^{local} is the worst-case access of a local lower-priority task to a local resource and B_i^{local} is the worst-case access of a local lower-priority task to a global resource plus the maximum spin delay as calculated in equation 3.1 suffered by that task accessing that resource.
- The worst-case execution time of a task is the sum of its execution time (including accesses to shared resources) and the spin delay suffered:
$$C_i' = C_i + \sum_{r^j_{ih}} spin(r^j, P_k) \tag{3.2}$$

Then the scheduling analysis for MSRP is passed if for all processors in the system, with tasks T_{P_k} ordered by decreasing preemption level, the following inequality is satisfied:

$$\forall i = 1, \cdots, n_k \quad \sum_{l=1}^{i} \frac{C_l'}{T_l} + \frac{B_i}{T_i} \leq 1 \tag{3.3}$$

This MSRP classic analysis was improved in [27] with a holistic analysis approach. Following this approach, the blocking suffered by a job or task is not analysed independently for each access request but all the blocking is considered as a whole. Typically requests were considered independent, and the worst case applied to all possible access requests. The holistic approach removes this source of pessimism by limiting the number of possible blocking sources during a job activation, ensuring that no extra blocking is accounted for.

This approach was further improved in [128] where two novel approaches are used: first, mixed-integer linear programming (ILP) techniques are first used to analyse blocking suffered by tasks as a result of shared resources. Secondly, the approach to account for the blocking is separated from the execution time of the task, thus providing a less pessimistic (more exact) analysis.

The blocking analysis considered in [128] does not rely on inflating the execution time of the task with the blocking suffered. On the contrary, it is analysed separately as shown in the following response time equation:

$$r_i = e_i + b_i + \sum_{\tau_j \in \mathbf{hpl}(i)} \left\lceil \frac{r_i}{T_j} \right\rceil e_j \tag{3.4}$$

where e_i is the worst-case execution time not considering blocking, and T represents the period of a given task. $\mathbf{hpl}(i)$ returns the set of tasks of higher priority than τ_i. Note the relevant difference with the analysis of equation 3.2 where the spin delay is added to the execution cost itself.

Then, the optimization objective of the ILP program is to maximize b_i, what is the maximum blocking possible caused by all tasks to τ_i:

$$b_i \triangleq \sum_{\tau_x \neq \tau_i} + \sum_{r^j \in R} + \sum_{v=1}^{N^i_{x,j}} (X^S_{x,j,v} + X^A_{x,j,v}) \cdot L_{x,j} \tag{3.5}$$

where $r^j \in R$ is each resource in the system, the values of $v = 1$ to $N^i_{x,j}$ represent the $v-th$ access of task τ_x to resource r^j, $X^S_{x,j,v} + X^A_{x,j,v}$ represent the possible spin delay and arrival delay caused by τ_x in its $v-th$ to resource r^j, being both bounded to be $[0, 1]$. Finally, $L_{x,q}$ is the longest access time by τ_x to resource r^j.

With b_i defined, a set of constraints to limit the possible values of $X^S_{x,j,v}$ and $X^A_{x,j,v}$ are given in [128] demonstrating, among others that:

- A single request from another task cannot cause τ_i at the same time spin delay and arrival delay, i.e. there is a bound so that $X^S_{x,j,v} + X^A_{x,j,v} \leq 1$.
- A job can suffer arrival delay from one resource, as inherited from SRP.
- For a resource to cause arrival blocking to τ_i it has to be accessed for at least one local lower-priority task and one local higher-priority task.
- Local higher-priority task cannot cause arrival blocking to τ_i.
- Neither local lower or higher-priority tasks can cause spin delay to τ_i.
- There cannot be more accesses causing spin delay to τ_i in its $v-th$ access to r^j than those that can be generated from remote tasks while τ_i is pending minus those already accounted for in the previous accesses.

This analysis is shown to be asymptotically better than those from [64] and [27] with increased benefits as the system gets more complex in terms of the number of tasks.

OMLP

The $O(m)$ locking protocol (OMLP) [29] is a locking protocol with versions for globally and partition scheduled systems. In general, the protocol is defined by the need of acquiring two locks to access a resource, instead of the common single lock requirement. These locks are to be sequentially acquired:

- Priority Queue lock: the first lock to be acquired is the PQ lock. This lock is also referred as to m-exclusion lock, as it can be held by up to m tasks. It is essentially a processor lock, since, in globally scheduled systems, the lock can be held by as many tasks as processors are present in the system, while for partitioned systems, there is one PQ lock on each processor.
- FIFO lock: after acquiring the PQ lock, tasks are queued for a second lock, resource specific, granted in FIFO order, named FQ. With the exception of the job at FQ head, queued jobs are suspended. Under global schedulers the head inherits the highest priority of the jobs blocked in FQ or PQ. Under partitioned systems, PQ lock holders execute priority-boosted (non-preemptively) unless suspended.

Under partitioned systems where each processor implements an EDF scheduler (P-EDF), the blocking suffered by a job per activation can be defined as the sum of the following terms:

- Local blocking: denoted as B_i^{prio}, occurs when τ_i is preempted by a lower-priority task priority boosted as released with FQ lock. It is easily bounded as the worst-case access time of a lower-priority task to a resource. As to be priority boosted a task has to acquire the local single PQ lock, τ_i can only suffer local blocking if a lower-priority task already held the PQ lock before it was released. Formally:

$$B_i^{prio} \triangleq max\{L_{x,k} | \tau_x \in part(P_i) \wedge 1 \leq k \leq q\} \tag{3.6}$$

where $L_{x,k}$ is the longest access of τ_x to a resource r_k and $part(P_i)$ denotes that a certain task is allocated to processor P_i.

- Remote direct blocking: denoted as B_i^{fifo}, occurs when τ_i is waiting in a FIFO queue, and it can be as long as the number of times it accesses a global resource, times the number of remote processors from where it can be accessed times the worst remote accesses time for that resource:

$$B_i^{fifo} \triangleq \sum_{k=1}^{q} N_{i,k} \cdot (m-1) \cdot max_{1 \leq x \leq n}\{L_{x,k}\} \tag{3.7}$$

- Remote transitive blocking: denoted as B_i^{trans}, occurs when τ_i is blocked due to a lower-priority task holding the PQ lock. Then it is blocked as much as the worst-case lower-priority access cost to a global share resource (contended by up $m-1$ other tasks as much):

$$B_i^{trans} \triangleq (m-1) \cdot max_{1 \leq k \leq q} max_{1 \leq x \leq n}\{L_{x,k}\} \tag{3.8}$$

The total blocking is the sum of the three terms $b_i \triangleq B_i^{prio} + B_i^{fifo} + B_i^{trans}$ and can be used to determine the schedulability of each processor P_o if:

$$\sum_{\tau_i \in part(P_o)} \frac{c_i + b_i}{T_i} \leq 1 \tag{3.9}$$

where c_i is the worst-case execution time of τ_i and T_i its period.

The most innovative proposal from OMLP is its double queue ruled by different criteria to bound the priority inversion blocking to $O(m)$, it is, the number of processors in the system.

OMIP

The $O(m)$ independence preserving protocol (OMIP) [28] addresses instead the problem of latency-sensitive tasks that suffer priority inversion blocking due to shared resources that they do not access.

The intuition behind OMIP is that, when priority boosting is used, as in PCP, SRP and protocols inspired by them, such as OMLP, tasks not requiring any locked resource might be prevented from executing by lower priority tasks as having increased their priority (priority inversion).

Opposite to the priority boosting strategy, OMIP presents a mechanism in which having locked a resource does not increase the local priority of the task (i.e. its rights to execute on its hosts processor), but increases its affinities (i.e. its rights to execute on others than its host processor). In particular, a task holding a resource lock under OMIP is allowed to migrate upon local preemption to any processor in which there is a job that is suspended waiting for the locked resource and would be otherwise executing if it had the resource lock. This is known in OMIP as migratory priority inheritance.

In the evaluation section of [28], it is shown how high-frequency latency-sensitive tasks not accessing shared resources can behave under OMIP as there were no locks in the system, i.e. without suffering from any priority inversion. However the same evaluation shows how there is a trade-off between this benefit and an increased response time for lower priority tasks accessing shared resources. In particular in scenarios of high contention OMIP behaves worse than OMLP for lower priority tasks accessing shared resources, as they are more likely to be preempted by higher priority independent tasks. On the contrary, when latency sensitivity is the scheduling bottleneck OMIP is the protocol of choice among them.

RNLP

The Real-time Nested Locking Protocol (RNLP) [125] was the first approach to fine-grained analysis of nested resources under multiprocessor systems. In contrast to other previous supports using group locks, RNLP only requires an irreflexive partial order on the nesting to avoid deadlocks and provide a fine-grained analysis.

The protocol is characterized by its modular approach in which there is an initial k-exclusion lock that has to be acquired before an access request can be served by a request satisfaction mechanism (RSM). RNLP is in that sense similar to the previously mentioned OMLP protocol, with a two-steps process to access a resource. The novel approach of RNLP is the possibility of combinating the k-exclusion locks and satisfaction mechanisms depending on the system configuration to improve efficiency.

The k-exclusion lock can be contended by spinning or suspending waiting tasks, being k values optimized in n (number of tasks in the system) or m (number of processors in the system) depending on the system configuration and RSM used.

The request satisfaction mechanisms decide which of the tasks holding a k-exclusion is granted access to its required resource. Up to four different satisfaction mechanisms are proposed in [125].

- S-RSM: access to resources is performed by spin-waiting in a resource associated queue, non preemptively. This approach can be applied to all kind of systems: global, partitioned or clustered systems.
- B-RSM: in contrast to S-RSM, tasks do suspend if they are not granted direct access to the resource. Tasks holding a k-exclusion token are priority boosted, above any non-k-exclusion holding task, to ensure progress. As S-RSM, it can be used on all three

system configurations.

- I-RSM: as B-RSM, tasks are suspend while waiting for a resource, but task are not priority boosted but inherit priorities from waiting tasks. In contrast to the previous RSMs, it can only be applied to global systems, as priorities among processors are not comparable.

- D-RSM: similar to I-RSM but priorities are donated instead of inherited and can only happen within clusters. As global and partitioned systems can be seen as particular cluster configurations, D-RSM is also applicable to all systems.

RNLP was later refined in [126], where different issues of the original proposal are addressed. First, they include the possibility of locking a group of nested resources on a single system call, to reduce the associated overhead. These groups are called dynamic group locks (DGLs), as they can be dynamically defined during execution. They also address the issue of short-on-long blocks, where due to transitive blocking a short resource request is blocked by a long resource request. To solve this issue, a biased generalization of RNLP is proposed to favor short requests. Finally, they incorporate replicated resources to improve the system behaviour when an access request can be satisfied by more than one instance of a resource.

The protocol was again extended in [127] where a mechanism is proposed to support reader and writer differentiated accesses. Several instances of critical sections need only to be accessed in mutual exclusion by writing operations, while reading ones can be done concurrently. Furthermore, the authors consider also a situation in which a read access can be upgraded to a write operation, depending on the value read.

SPEPP

The Spinning Processor Executes for Preempted Processor (SPEPP) presented in [121] introduces a practical way of reducing the cost of remote preemptions when accessing global resources in partitioned systems. Commonly, this issue has been addressed by different protocols by accessing the resources non-preemptively, or with a boosted priority. Furthermore, in many protocols, waiting is also performed non-preemptively to avoid the situation in which a task is granted a lock, but is not scheduled in its own processor, due to different factors. SPEPP avoids these situations with a novel helping approach.

When a task requires a globally shared resource, it adds itself to a FIFO queue ruling the access to the resource. This process not only adds the identity of the task willing to access the resource, but also the action to be performed inside the resource. Then it spin-waits (preemptively) until access is granted (the critical section is executed non-preemptively). If during the waiting it is preempted, and not rescheduled before it becomes the head of the queue, any other task in the system also spin waiting for the resource can complete on its behalf the requested operation. As such, the execution time spent to access the resource (time occupying the processor) is bounded to n times the resource access time. Then, when the helped task is resumed, it finds its access request satisfied and result, if any, ready to be used.

This protocol shows how a mechanism can be implemented to ensure that whenever a task is actively waiting for access to a resource, progress is done in such resource. However, the overheads posed by the need of fully specifying the action to be performed and the need of checking the result of the operation has kept SPEPP to be just an academic exercise.

M-BWI

The Multiprocessor Bandwidth Inheritance (M-BWI) protocol [56] also addresses the issue of locks held by tasks that, for some reason, are not able to keep progressing. In particular, M-BWI is aimed for systems scheduled within execution time *servers*, limiting the processor time a task can consume. In case a resource holding task runs out of budget or it is preempted, it can be helped by a waiting task by consuming part of the helper budget. This is implemented in practice by migrating the task to the helper server in order to complete its access and release the lock.

3.2.2 Global Systems

Along with fully partitioned systems, another main multiprocessor system design trend appeared in the advent of multiprocessor platforms: globally scheduled systems. In contrast to fully partitioned systems, the approach of globally scheduled systems is to consider the scheduling problem of the system as a whole. In this way, there is a single ready-queue and tasks are dispatched to any of the available processors. This presents a set of benefits when compared to fully partitioned systems:

- The computing capacity (processors) is seen and managed as a whole. In this way, the inherent spare capacity left when allocating tasks in fully partitioned systems is avoided. Furthermore, the system as a whole is benefited by any task completing before its WCET, and not only those tasks allocated to the same processor.
- In case of time failure, globally scheduled systems tend to be safer, as all the system resources can be used to mitigate the effects of the failure.

- In general, globally scheduled systems are work conserving, since tasks are allowed to execute on any idle processor, i.e. there can be no ready task not dispatched and idle processors at the same time.
- Related to the previous point, globally scheduled systems suffer typically less context switches and preemptions, as tasks are dispatched to idle processors first [7].
- Globally scheduled systems do not require load balancers, since the global ready queue and dispatching policy does inherently balance the load among the system processors.

Scheduling systems globally also encounters drawbacks, mostly related with task migrations. While being able to dispatch a task on any available processor gives higher flexibility and composability to the system, it also produces migration overheads. Tasks do always have related data structures, that are to be transferred from one processor to another when migrating the task. Furthermore, in systems with cache memories, affinities are lost upon migration affecting execution times. To reduce this effect, some approaches do not permit jobs to migrate: jobs of a task can be dispatched on any processor in the system, but once a job has started its execution, it is allocated to the processor where it has been released for the rest of the activation. This is known as task-level migration, in contrast to job-level migration systems, where jobs are allowed to migrate at any time. The latter approach is the most common and thus assumed in this section unless explicitly stated.

Initial approaches to globally scheduled systems gave unsuccessful results. In particular, the results in [54] showed how global EDF can lead to very poor schedulability, down to $1 + \varepsilon$ for arbitrary small ε. The so called Dhall effect [50] shows how a globally EDF scheduled system is infeasible when m or more tasks with short periods (and deadlines) and infinitesimal utilizations are to be scheduled along with a task with a longer period and utilization close to 1. As a result, interest in globally scheduled systems decreased for up to two decades, until research in [101] and [61] demonstrated that a very high utilization task is required for the Dhall effect to occur, along with other task set characteristics.

Not surprisingly, approaches then tended to provide solutions to support those high utilization tasks. For periodic task sets with implicit deadlines and fixed priorities, Andersson et al. [6] bounded the theoretical maximum (optimal) utilization to:

$$U_{OPT} = (m+1)/2 \tag{3.10}$$

Then, an approach named EDF-US[ς] was proposed in [119], where tasks with a higher utilization than a threshold ς are given the highest priority, setting the threshold to $m/(2m-1)$ the utilization bound is:

$$U_{EDF-US[m/(2m-1)]} = m^2/(2m-1) \tag{3.11}$$

Later, two different works [73, 19] found out a tighter bound than equation 3.10, not requiring any artificial priority assignation as EDF-US in which a maximum utilization target (other than 1) u_{max} can be defined and proven:

$$U_{EDF} = m - (m-1)u_{max} \tag{3.12}$$

New results on EDF-US [16] proved that letting ς to be 1/2 results in the utilization bound identified by Anderson [6]:

$$U_{EDF-US[1/2]} = (m+1)/2 \tag{3.13}$$

Finally, EDF(k) proposed by Goossens et al. [73] slightly modifies the EDF-US approach. Instead on giving the highest priority to tasks above a threshold, EDF(k) gives the highest priority to the k tasks with the highest utilizations. Letting u_{sum} to be the task set overall utilization and u_k the utilization of the kth highest utilization task, EDF(k) schedulability is proven if:

$$m \geq (k-1) + \left\lceil \frac{u_{sum} - u_k}{1 - u_k} \right\rceil \tag{3.14}$$

In [16] a variant of EDF(k), called EDF(k_{min}), is shown to achieve the optimal utilization if k is chosen to be the lowest value satisfying equation 3.14.

$$U_{EDF[k_{min}]} = (m+1)/2 \tag{3.15}$$

For more general task models, as those with constrained or arbitrary deadlines, similar results can be achieved. In these cases, however, the evaluated parameter is density (δ) rather than utilization (u) . First, equivalent results as of equation 3.12 were presented for constrained deadlines [24] and for arbitrary deadlines [17]:

$$\delta_{sum} \leq m - (m-1)\delta_{max} \tag{3.16}$$

Furthermore, Bertogna [23] proved that the approach of EDF-US could also be applied to sporadic task sets with constrained or arbitrary deadlines. This new approach, called EDF-DS[ς] uses a density threshold to grant the highest priority in the system. As in EDF-US, setting the threshold to be 1/2, optimality is achieved:

$$\delta_{sum} \leq (m+1)/2 \tag{3.17}$$

Regarding globally scheduled systems with dynamic priorities, they are known to dominate all other global approaches [50]. However, they are in practice less exercised, as they tend to produce frequent preemptions and migrations, causing relevant overheads.

Optimality has been proven for different algorithms scheduling periodic task sets, such as PFair, LLREF or EDZL families. However, optimality cannot be achieved for sporadic task sets without clairvoyance [57].

The Proportionate Fair (PFair) [20] algorithm is based on a concept known as quanta scheduling: scheduling decisions are only taken at fixed intervals of time, being the slots in between known as quantas. Execution rights are given to tasks according to the processor time already allocated and the task utilization. Without considering overheads, PFair is optimal ($U_{PFAIR} = m$). However, overheads due to rescheduling at each quanta makes the realistic utilization far below that value. PFair was implemented by Holman and Anderson [76] and found out that in practice the main overheads come from the contention for the communication bus at the rescheduling time between quantas. To address this issue they proposed a staggered version of the algorithm, by which each processor would have its quanta slightly delayed from the previous one, thus reducing the bus contention at the cost of also reducing the schedulability. Other approaches to PFair include PD [21], PD^2 [5], ERFair [4] and BF [141]. More notable is the Earliest Pseudodeadline First algorithm (EPDF) [4] that applies PFair to sporadic task sets giving different support to light and heavy tasks. EPDF was shown to be optimal for sporadic tasks sets with implicit deadlines on systems comprising two processors, but not on systems with more processors.

Another algorithm considering time fractions is the Largest Local Remaining Execution Time First (LLREF) [43]. In the case of LLREF these fractions are not equal in length but ruled by normal scheduling events, e.g. task releases, deadlines, and so on. At the beginning of each section, the m tasks chosen to execute are those with largest execution times remaining according to their utilization and the length of the section. This work was later extended in [60] considering task completing before their deadline, providing a work-conserving algorithm with reduced overheads compared to the original LLREF in systems with utilizations below 100%. Another derivative of LLREF is the LRE-TL [62] algorithm, in which it is found out that there is no need to choose the largest local task with remaining execution time, but any task with remaining execution would give the same results in terms of schedulability. Furthermore, this approach is proven to generate less migrations, being optimal for sporadic task sets with implicit deadlines and also applicable to sporadic task sets.

Finally, the Earliest Deadline until Zero Laxity (EDZL) [91] dominates EDF global scheduling. EDZL follows EDF rules until a task would require to execute for the remaining amount of time until its deadline. Then, this task executes at the highest priority until completion. A utilization bound is presented in [42], for task sets with implicit deadlines, being e Euler's number:

$$U_{EDZL} \leq m(1 - 1/e) \approx 0.63m \tag{3.18}$$

An interesting EDZL derivative, Earliest Deadline until Critical Laxity (EDCL) [83] applies a critical laxity criteria to increase job priorities only upon release or completion of a job. While this slightly reduces the schedulability compared to EDZL, it also reduces the maximum number of context switches to two per job, being then in practice superior to EDZL.

A number of protocols support task models including shared resources for globally scheduled systems. In fact, most of such protocols are versions or derivatives of partitioned protocols modified for global scheduling. Of those described in section 3.2.1, OMLP, RNLP, M-BWI and MSRP have such variants for globally scheduled systems. Due to its special relevance and interest, the MSRP version for globally scheduled systems is specifically addressed.

Global MSRP

In [53], a pair of analysis for global EDF systems is presented. These analysis consider task synchronization under two different mechanisms, queue locks and lock-free synchronization. Here we will focus on queue locks.

The system model comprises a sporadic task model, with task accesses to shared resources being synchronized using queue locks. These queue locks are implemented as FIFO spinning queues, being the spinning and access to the resource non-preemptive. Tasks are scheduled globally following EDF algorithm.

Under this configuration, tasks can be blocked under two scenarios: spin-waiting for accessing an already locked resource, or when it cannot preempt later deadline tasks due to those being accessing shared resources. The spin-waiting blocking is easily accounted for by bounding the worst-case access cost to a resource to be m times the access time.

The blocking due to non-preemptive execution of later deadline tasks is bounded to happen just once and before the task executes, as in SRP. Then, worst case for this blocking (B_i) is equal to the maximum access cost to a shared resource by a longer relative deadline task. Then for any system where $\forall \tau_i,\ p_i - B_i \geq e_i$ i.e. where all tasks the period/deadline (p) minus the arriving blocking (B) is greater than the minimum inter-arrival time, the system is schedulable if it is deemed schedulable according to equation 3.12.

FMLP

The Flexible Multiprocessor Locking Protocol (FMLP) [26] is a locking protocol applicable to partitioned and global EDF, and thus its "flexible" denomination. While the protocol itself is applicable to both paradigms, its mains contributions are only aimed for global EDF, and thus reviewed in this work as a global approach.

For both versions, FMLP distinguishes between short and long resources (at developer designation). Resources are grouped under FMLP and groups can only contain short or long resources. Nested resources are grouped together under the same group lock, and non-nested resources are grouped individually. While short resources can be accessed by long resources, long resources cannot be accessed from short resources.

Requests for short resources are serviced in FIFO order, in a non-preemptable fashion. Tasks in the FIFO queue do busy-wait until they are granted the lock. Once the lock is held, tasks keep executing non-preemptably and all requests to resources in the same group are serviced immediately. The lock is released when completing the outermost request within the group.

Requests for long resources are also serviced in FIFO order. However, long resource groups are guarded by semaphores. Tasks not granted the group lock are suspended. The task holding the resource inherits the priority of the highest-priority task blocked in the group lock. As mentioned before, long resources can request short resources, performing those request following short resources rules (non-preemptably).

As tasks can suspend or be non-preemptable, the authors name as GSN-EDF (G-EDF algorithm for suspendable and non-preemptive jobs) the application of FMLP to global EDF. This dual resource access paradigm is aimed to improve the efficiency of the common non-nested short resources accesses while maintaining low overheads for higher-priority tasks. The blocking calculation under GSN-EDF is defined as:

$$B(\tau_i) = BW(\tau_i) + NPB(\tau_i) + DB(\tau_i) \tag{3.19}$$

where $B(\tau_i)$ is the total blocking suffered by a task τ_i per activation calculated as the sum of the following three sources of blocking:

- Busy-waiting blocking $BW(\tau_i)$: is the time spent busy-waiting for access to a short resource. As short resources are accessed non-preemptably and requests are serviced in FIFO order, assuming a system with m processors, only $m-1$ other requests can be satisfy before a given request is given the lock. Thus, each request of τ_i to resource R can suffer at most $spin(\tau_i, R) = Msum(m-1, S)$ which is the sum of the $m-1$ longer access request to resource R. Then, the total busy-wait blocking suffered by τ_i per activation can be denoted as:

$$BW(\tau_i) \leq \sum_{R \in Q} spin(\tau_i, R) \tag{3.20}$$

  where Q is the set of resources accessed by τ_i.

- Non-preemptive blocking $NPB(\tau_i)$: is the time spent blocked when all m processors are occupied and τ_i has a higher priority than at least one executing task, but cannot preempt it as a result of the lower priority task being accessing a short resource (non-preemptively). This, under GSN-EDF can only happen when the task is first released and after each suspension due to being block on a long resource lock. As shown before, $spin(\tau_a, R)$ is the time a task can spin wait for accessing a resource R and thus its access cost (time non-preemptive) is bounded by $np(\tau_a) = max\{spin(\tau_a, R) + |R|\}$ being R the longest short resource accessed by τ_a. Then the total non-preemptive blocking can be computed as:

$$\begin{aligned} NPB(\tau_i) \leq max\{np(\tau_a) : \tau_a \in B(\tau_i)\} + \\ L(\tau_i) \cdot max\{np(\tau_a) : \tau_a \in A(\tau_i)\} \end{aligned} \tag{3.21}$$

where $B(\tau_i)$ is the set of jobs of tasks other than τ_i with periods longer than τ_i (for the arrival blocking) and $A(\tau_i)$ is the set of all other jobs, and $L(\tau_i)$ is the number of access request per activation of τ_i to long resources (number of times it can suspend).

- Direct blocking $DB(\tau_i)$: to define the direct blocking, first the holding time of a job has to be defined. A job holding time waiting for a lock belonging to a long resource group ($ht(\tau_i, R_x)$) is the time consumed accessing resources in the group R_x plus all the non-preemptive requests to short resources: $ht(\tau_i, R_x) = |R_x| + \sum_{R' \in I}(spin(\tau_i, R'))$ where I is the set of short resources accesses from R_x. Then, a job of τ_i is considered to be directly blocked when it is one of the m higher-priority tasks in the system and tries to access a group of long resources R_x but its lock is held by another task. This other task can be, in turn, non-preemptively blocked by a third task (due to be released after being suspended) or making progress inside the resource group (consuming its holding time). As any task can be queued before τ_i for R_x lock, the direct blocking per outermost access to a R_x group lock is:

$$\begin{aligned} db(\tau_i, R_x) = \sum_{T_a \in Z} (max\{np(\tau_x) : \tau_x \in A(\tau_a)\} + \\ max\{ht(\tau_a, R_a) : R_a \in G(\tau_a)\}) \end{aligned} \tag{3.22}$$

  where Z is the set of other tasks that can access the group R_x, and $G(\tau_a)$ is the set of outermost requests from τ_a to R_x resource group. So, the total direct blocking suffered per a job of task τ_i per activation is calculated as:

$$DB(\tau_i) = \sum_{R_x \ inL} db(\tau_i, R_x) \tag{3.23}$$

where L is the set of all outermost request to long resources groups performed by a τ_i job.

The relative easy way of calculating the blocking, along with the nested resources grouping and different locking proposals for short and long resources are the main traits of the FMLP proposal.

However, the different access policies for long and short resources, which is the most innovative proposal does not hold for partitioned EDF (P-EDF). This is due to two main factors: first, while in a globally scheduled system all resources are global, in a partitioned one, some resources are only accessed by jobs allocated to a single processor, and are considered local resources. Second, there is no point in comparing priorities from different processors, and so the priority inheritance in long resources accesses is pointless. As such, the FMLP application of P-EDF, named PSN-EDF (again SN for suspendable non-preemptive) rules all global resource accesses in a non-preemptive way, regardless of whether they are long or short, undermining the most innovative characteristic of FMLP.

3.2.3 Semi-Partitioned Systems

Both fully partitioned and globally scheduled systems present a number of pros and cons to be considered. Semi-partitioned approaches aim to exploit benefits from both approaches by, from a base of a partitioned system, letting a subset of the system tasks migrate or be split among the processors. These tasks are also known as shared tasks. In this way, system utilization can be increased as compared to fully partitioned systems, while scheduling overheads can be reduced, as compared to those from globally scheduled systems.

DM-PM

The Deadline Monotonic with Priority Migration (DM-PM) protocol [84] is aimed to provide a deterministic way of scheduling shared tasks within a fixed-priority scheduled system. In this protocol, tasks are allocated to processors filling their utilization until no other task can fit into that processor. Then, when no processor can fully allocate any of the remaining tasks, these are split among processors filling the remaining CPU time.

In DM-PM, shared tasks are split into consecutive processors, i.e., the first shared task uses the remaining capacity on P_1, then the remaining capacity on P_2, etc. until it is completely allocated. As it would probably not exactly fill the second processor, the next shared task would also share the processor with that first shared task. As a result, on each processor, there can be: no shared task if higher index processors are not required to allocate shared tasks; one shared task if it allocates the last task to share; or two shared task if the end of one task overlaps with the beginning of the next one.

DM-PM assigns priorities following a Deadline Monotonic [94] policy, with the exception of the shared tasks, which are given the highest priorities on each processor. In case of processors with more than one shared task, ties are broken in favor of the latest task assigned to that processor, i.e. the task that has those processors as the lowest index among the ones to which it has been allocated to. As a result, the system is optimized when the assignation of tasks to processors is done in decreasing order of relative deadline. As a result, shared tasks are those that naturally should have been given the highest priorities regardless of being shared tasks.

During execution, each shared task starts its execution on the lowest index processor on which it has been allocated to. Then, when its execution capacity is consumed it is migrated to the next processor until its execution is finished. This means that on each release, the task only executes once on each processor, always migrating in the same order. The implementation of this behaviour is identified in [84] as one of the simplest among all the previously proposed semi-partitioned approaches, as only one timer for shared tasks on each processor is required in order to determine when it is to be migrated.

The schedulability analysis of DM-PM is performed by Response Time Analysis. Since tasks have a fixed priority on each processor, and fractions of shared tasks are modelled as independent tasks, this analysis does not essentially differ from that of fully partitioned systems. It has to be noted, however, that the approach presented in [84] does not include the notion of nested resources, and so only the interference of higher-priority tasks and the execution time of the analysed task are considered in such RTA analysis.

A version of DM-PM protocol for Earliest Deadline First schedulers is presented in [85] using a window concept to determine whether tasks are allowed to migrate to the next processor. This protocol, named EDF with Window-constraint Migration (EDF-WM) presents a more complex schedulability test and its implementation is identified by the authors as notably more complex than the fixed priority approach.

C=D

Another semi-partitioned scheme designed for Earliest Deadline First schedulers is described in [33]. On it, a set of optimization strategies are defined to solve the allocation and task splitting policy, but the authors consider also a 'first-fit' bin packing algorithm as acceptable and 'next-fit' for the tasks to be split or shared. This, in practice means that tasks are assigned to one processor until no further task can be allocated. Then the next non-allocated task is split to fill the remaining capacity. The remaining fraction of the task is sent to the following processor and the process is repeated until all tasks have been allocated (schedulable system) or all processor are filled (unschedulable system).

As a result of the splitting scheme presented, if the system is schedulable, there can be as many as $m - 1$ shared tasks in the system, and each task is only split into two fractions executing on consecutive processors.

The split tasks are scheduled on each processor so that the first fraction is given a deadline equal to the execution time that can be allocated. As such, the split task has the highest priority in its first processor. The second fraction of the task is then given a deadline and computation time equal to its original values minus those allocated on the first processor. As a result, its priority will be lower than the next split task. Its priority against all other allocated tasks in its second processors depends on each specific task set following this simple approach.

At runtime, each split task has the affinity of its first processor. When it is executed and its deadline is elapsed, the task affinity is changed to the second processor. As such, it does not require any special feature that should not be already present on any EDF scheduler allowing migrations.

The schedulability of the system is determined following regular EDF methods. However, a relevant difference of the C=D partitioning scheme is how to determine the computation time that can be allocated to a task on its first processor (and thus its deadline). This is done using a sensitivity analysis in which an initial computation time that makes the overall utilisation of the processor equal to 1 is assigned (and its deadline equal to that value). Then the maximum test interval is computed and evaluated using QPA (Quick convergence Processor-demand Analysis) [136, 137]. If there is a failure, the execution time value is reduced until the test is passed or the execution time is 0, meaning that no fraction of the task can be allocated to that processor. The system is schedulable if all tasks have been fully allocated to a processor at the end of this splitting algorithm.

PDMS_HPTS

Splitting the highest-priority tasks is also one of the main strategies followed by the Partitioned Deadline Monotonic Scheduling Highest-Priority Task Splitting (PDMS_HPTS) [87] family of algorithms. As its name suggests, PDMS_HPTS considers a set of m processors scheduled with fixed priorities assigned following the deadline monotonic algorithm.

The discussion in [87] about task splitting is conducted by two main concepts: the utilization bound (UB) as it has been used in this document, meaning the maximum utilization an algorithm can ensure schedulability of any valid task set; and the size bound (SB) which is the highest size ($\frac{C}{D}$) allowed for a task in the task set.

This splitting algorithm is as follows: tasks are allocated to processors following an arbitrary order. When a processor P_j first fails to be schedulable after allocating a new task to it, the highest-priority task τ_h according to Deadline Monotonic is chosen to be split. A maximizing algorithm is then used to find the highest-computation time C' that can be allocated to P_j maintaining its schedulability. From the original τ_h two split tasks are generated:

$$\tau_h \ : \ (C, T, D) \Longrightarrow \begin{cases} \tau' \ : \ (C', T, D) \\ \tau'' \ : \ (C - C', T, D - C') \end{cases} \tag{3.24}$$

where τ' remains allocated in P_j and τ'' is added to the set of tasks to be allocated. P_j is then marked as not available and thus no new task will be tried to be allocated in such processor.

As the highest-priority task is chosen to be split, τ' worst-case response time is equal to its worst-case execution time, i.e. $R' = C'$, as PDMS_HPTS does not consider shared resources and thus blocking. To prevent τ' and τ'' to execute in parallel, τ'' release is set C' units of time after τ' release.

When a task is split following the previous algorithm, there is a penalty, denoted as δ which is function of the original task size $S(\tau)$:

$$\delta = S(\tau') + S(\tau'') - S(\tau) \leq 2(1 - \sqrt{1 - S(\tau)}) - S(\tau) \tag{3.25}$$

For systems with a size bound ($SB \leq 0.25$) this penalty is lower than 2% [87].

Regarding the utilization bound, for task sets with implicit deadlines, the bound is identified to be above 60%, which improves the 50% utilization bound for fully partitioned fixed priority task sets.

$$UB_m(PDMS\_HPTS) = \frac{0.6003 * (m-1) + 0.6931}{m} = 0.6003 + \frac{0.0928}{m} \tag{3.26}$$

For task sets with a size bound ($SB \leq 0.414$) the utilization bound grows to a 69%.

Furthermore, if tasks are allocated in decreasing order, the resulting algorithm, called PDMS_HPTS_DS yields a utilization bound equal to its size bound:

$$UB(PDMS\_HPTS\_DS) = SB(PDMS\_HPTS\_DS) = 0.6547 \tag{3.27}$$

Semi-partitioned Mixed Criticality Systems

As stated in the introduction, multiprocessor systems are motivated by the will of increasing the overall computing power of the system. This can be used to execute more processes, heavier processes, or a combination of both. When more than one process is executed on a system, it is possible to have processes of different importance. In real-time scheduling theory the importance of a process or task is denoted by its criticality, i.e. the resulting effect of a task time failure.

Different levels of criticality require different levels of assurance. For higher-criticality levels a complete assurance is required, while for lower levels a certain degree of uncertainty is allowed. One common way to increase the assurance level is to inflate the WCET of a task. As it has been already highlighted, this goes against the schedulability of the system, as pessimism on WCET generates idle processor times that cannot be allocated to other tasks.

One common way to address mixed-criticality systems is to give tasks different WCET values, one for each level of criticality [123]. For higher levels, the WCET value is the safest considered, and it is decremented as the criticality level also decrements. Then, the schedulability of each criticality level is evaluated using the task assigned WCET value for that level.

A relevant approach is proposed by Mollison et al. in [98]. In this work, a system with 5 levels of criticality, as in the RCTA standard DO-178C [112] is considered. This approach proposes different scheduling algorithms for each level, managed via containers. In particular, levels A and B, the two highest levels in the system, are partition scheduled: level A is scheduled on each processor using a cyclic executive, while level B is scheduled using EDF. The remaining levels C, D and E are scheduled using global schedulers: levels C and D are scheduled using global EDF while E level is considered for non-real time tasks and can thus be scheduled as required.

The schedulability of level A is straightforward, as any feasible cyclic executive for each processor is valid. Level B is schedulable by construction by imposing two requirements: level-B utilization values for tasks belonging to A and B levels must not be over 1; and level-B periods must be integer values of level-A hyperperiod. For levels C and D, a modified version of G-EDF analysis proposed by Leontyev and Anderson in [93] is presented. The remaining utilization available for each level C and D can be then calculated considering the relevant WCET values for each level. Level E tasks are considered "best effort" so no schedulability analysis is required, but its expected utilization bound can also be calculated using the same approach as for levels C and D.

Another interesting approach to mixed-criticality semi-partitioned systems is presented in [32] where a dual criticality system is considered. Jobs and tasks are given a criticality which can be high (HI) or low (LO). A platform of m identical processors is considered where tasks are scheduled by a cyclic scheduler, which is common scheduling industrial practice. A particularity of the approach proposed in [32] is that, at any given point, only tasks of the same criticality are executed, as first suggested in [72]. This means that, on each minor cycle, there has to be a scheduling switching point S, where HI tasks do not longer execute and LO tasks do. S value is calculated as follows: a minimum value is considered as the minimum time required for HI criticality tasks under LO scheduling, i.e. the most optimist condition, which is calculated as:

$$S^{min} = \max\left(\frac{\sum_{X_i=HI} C_i(LO)}{m}, \max_{X_i=HI} C_i(LO)\right) \tag{3.28}$$

where X_i indicates the criticality of the task τ_i and $C_i(X_i)$ the WCET value of τ_i under X_i criticality level. Equation 3.28 considers the optimist WCET value for HI tasks. However, it has to be proven that HI tasks are schedulable also with their HI WCET values. To do so, a new term for HI tasks is introduced: $C_i(EX)$ represents the excess time for a HI task as the difference between its $C_i(X_i)$ values.

$$C_i(EX) = C_i(HI) - C_i(LO) \tag{3.29}$$

With this value, the maximum extra time that HI tasks would need after S if they all execute their HI WCET value is:

$$\Delta^{HI} = \max\left(\frac{\sum_{X_i=HI} C_i(EX)}{m}, \max_{X_i=HI} C_i(EX)\right) \tag{3.30}$$

Then, it can be stated that HI tasks are schedulable if $S^{min}+\Delta^{HI} \leq D$. Regarding LO tasks, its schedulability has to be proven against $C_i(LO)$ values for both HI and LO tasks. Again, the minimum time required by all LO tasks to complete is:

$$\Delta^{LO} = \max\left(\frac{\sum_{X_i=LO} C_i(LO)}{m}, \max_{X_i=LO} C_i(LO)\right) \tag{3.31}$$

And again schedulability is demonstrated if after S there is at least Δ^{LO} time for LO tasks to complete before the end of the frame: $S^{min}+\Delta^{LO} \leq D$. As a result, a frame is schedulable for HI and LO jobs if HI jobs are schedulable even extending longer than their $C(LO)$ values and HI and LO jobs are schedulable if all HI jobs complete at their $C(LO)$ value:

$$S^{min} + \max(\Delta^{LO}, \Delta^{HI}) \leq D \tag{3.32}$$

When the entities to be scheduled are tasks instead of individual jobs, a strategy is required to determine which tasks are to execute on each of the minor frames to meet their time requirements. In [32] the number of minor cycles k is assumed to be an integer power of 2, being tasks periods harmonic to that k. Tasks with periods equal to k are allocated first following the previously presented mechanism to ensure schedulability. Then, tasks with periods multiple of k are allocated. It may happen that a task cannot be fully allocated to a single frame. Instead, it needs to be split among different frames. For example, consider a task with period $2k$ and C(HI) = 6 and C(LO) = 4. The proposal in [32] is to divide the computation time in pairs of $(C_i(LO), C_i(EX))$. Initially $\frac{C_i(HI)}{p}$ of $(C_i(LO)$ is assigned to each frame until there is the total $C_i(LO)$ is allocated. Then the excess is allocated until the sum of $C_i(LO) + C_i(EX)$ equals the total $C_i(HI)$ task value. On the example the pairs would be: (3,0),(1,2) so $C_i(LO) + C_i(EX) = C(HI) = 6$. If this makes the first frame not schedulable, $C_i(LO)$ work can be transferred to the second frame: (2,0),(2,2). This process is repeated until the task is fully allocated, otherwise the system is unschedulable.

As jobs in general are added to each processor until its frame is full, some tasks are required to migrate. As a value of S cannot be smaller than the biggest $C_HI(LO)$ value, fragments of a job cannot overlap.

3.2.4 Virtualized Multicore Systems

In the previous section, a way to build systems with applications of different criticality levels has been presented. While in some of the proposals there is a specific effort to provide some isolation among criticality levels, as suggested in [72], this isolation cannot be considered complete. In terms of isolation, it can be distinguished:

- *Temporal isolation*: a time fault on one isolated part does not affect other isolated parts. For example, a deadline miss in Isolated Part 1 does not cause Isolated Part 2 to begin its execution later.

- *Spatial isolation*: isolated parts cannot access (neither read or write) memory addresses of other isolated parts. This can be achieved using the memory management unit (MMU) or mixed software and hardware procedures [99].

- *Fault isolation*: any kind of fault state should not propagate among isolated parts.

A successful approach addressing both isolation requirements is presented in [113], where a *separation kernel* is proposed. This kernel is again a combination of hardware and software procedures to provide a set of security properties to resulting systems.

Systems complying with this kind of isolation requirements are known as partitioned systems. Similarly to partitioned scheduling, partitioned systems consist on a set of isolated parts known as partitions. However, while in partitioned scheduling partitions were mainly used to allocate tasks to one specific processor, the notion of partitions in partitioned systems is used for one or more task that can execute in one or more processors with isolation of other partitions.

Current industrial practice in partitioned systems is to use virtualization technology [47] to achieve the mentioned isolation. With this technology, the behaviour a of a machine can be simulated using software techniques, with optional hardware support. While from the point of view of a partition it executes on its own machine, several virtual machines (partitions) can be run into a single physical device. Thus, different partitions of different criticality levels can be run on a single platform. Furthermore, virtual machines can execute different operating systems, thus allowing to use a proper one for each partition. For example, an industrial appliance could use a partition with a real-time kernel for controlling its operation and a Windows or Linux partition to provide a human interface.

Apart from virtual machines, two other pieces of software are needed to support a virtualized system. The virtualization layer is a piece of software in charge of the computer resources virtualization, i.e. to provide a set of resources to each virtual machine. A hypervisor, or virtual machine monitor (VMM), is a software or a mixture of hardware and software layers that enable the execution of different partitions on the same physical machine. It is in charge of the creation, destruction and management of virtual machines and their resource usage.

Hypervisors can be classified into two main categories or types: type 1 hypervisors, or bare-metal hypervisors that run directly on the platform hardware, and type 2 hypervisors or hosted hypervisors that run above an underlying operating system e.g. VMware or VirtualBox are type 2 hypervisors. With regard to real-time systems, type 1 hypervisors are the most promising approach, as they can provide better performances and control over the system platform (it would not be safe to execute a high-criticality partition over a general purpose OS, as the latter cannot provide real-time properties by default).

The most experienced way to schedule partitioned systems is to use hierarchical scheduling techniques [47]. This approach uses a divide and conquer strategy: the schedulability problem is divided into hypervisor level (or higher level) scheduling and kernel, partition or lower level scheduling. Both levels can be interrelated depending on the approach followed:

- Flat scheduling: in this approach, the interrelation between tasks in the system is analysed first. Then an appropriate partitioning is generated aiming to maintain in the same partition interrelated tasks in order to reduce context switches and bus load. This approach is, however, limited when the aim is to isolate partitions of different criticality. A good example of this approach is proposed by Salazar [114] which uses coloured graphs to find an appropriate partitioning of tasks considering different non-functional requirements (including timing or security among others).

- Served-based scheduling: partitions are considered to be server-based schedulers, with each partition being allocated an executing capacity and replenishment period. While with this approach the know-how from non-virtual scheduling can be reused, it is not appropriate for higher levels of criticality. An approach providing a response time analysis to fixed-priority application servers executing above a cyclic executive hypervisor is presented in [1] and improved by using end to end flow analysis in [100].

- Compositional scheduling: each partition requests an amount of computation (normally as a pair of length and period) and the level 1 scheduler tries to accommodate all partition requirements. This in practice means to generate a cyclic executive of partitions. While this approach may be less efficient (as all partitions will require their pessimistic amount of computation to increase assurance), it can provide a higher amount of predictability, specially when cyclic executives are used for level 1 scheduling. A representative approach is presented in [130].

3.2.5 Manycore Systems - Clustered Systems

As it has been shown in previous sections, different approaches have been proposed trying to overcome the intrinsic drawbacks of the traditional fully partitioned and global approaches. While finding an efficient allocation (solution to the bin-packing problem) is the main issue with fully partitioned systems, the excessive overhead of migrations and long ready queues are the main challenges when considering globally scheduled systems. Semi-partitioned systems address this by allowing certain tasks to be split and migrated among processors, easing the allocation problem with a bounded effect on the length of ready queues and migrations. Virtualized systems apply a divide and conquer strategy providing isolation among subsets of the system tasks at the cost of requiring extra hardware and software solutions.

Another possible approach is to create subsets of the available resources (essentially processors) to generate more efficient systems. These subsets are known as clusters. As stated during the introduction, the hardware design trend is to reduce processor frequencies but provide more than one computing unit on the same chip. While for general purpose equipment the trend followed by the main manufacturers has been to slightly reduce the frequency, adding 2, 4 or 8 processing units per chip [78, 3], there are some other approaches with highly reduced frequencies, but notably more processing units. This kind of designs are known as *manycore* systems and are generally characterized by the close coupling of sets of processors. These sets or clusters share different resources as accesses to general I/O buses, cache memories, or faster communications within them. The size of a cluster in terms of processors is denoted by c. It is common practice to design systems where m and c are powers of 2, with a number of clusters $\lceil \frac{m}{c} \rceil$ also being a power of 2, e.g. a 64 processors system (m) with 4 clusters of 16 processors (c).

These architectures can be used to build real-time systems with specific properties. The common practice is to statically partition the system offline by allocating tasks to clusters and schedule them globally at runtime. In this way the allocation problem is eased (as the size of the bins is greater) while the drawbacks of globally scheduled systems are alleviated (ready queues length are divided by the number of different clusters in the system as well as migration costs are reduced thanks to shared cache memories) [37].

From the point of view of the system schedulability, clusters can be abstracted as independent systems. In that sense, schedulability tests of non-clustered systems are still valid: once a system has been partitioned, each cluster can be tested using the methods reviewed above. In particular global EDF schedulers are used in clustered systems [117, 92]. Clustered scheduling has been implemented over LITMUS^{RT} [38], with C-EDF (clustered EDF) and C-PD^2 schedulers [27].

3.3 Summary

In this chapter a revision of the most relevant scheduling methods for multicore real-time systems has been presented. All these proposals contributed up to some extent to advance the state of the art by introducing novel concepts and techniques or reusing the existent to obtain better schedulability results.

Unfortunately none of the presented approaches has successfully overcome all the challenges outlined in the introduction and described in this chapter. Most of the presented protocols impose some restrictions to the task or shared resources models. In particular, only a subset of them support intertask communication using nested resources. This limits the complexity of the systems that can be addressed with those protocols undermining their practical interest.

Another factor that diminishes the interest of some of the mentioned protocols is the uneven timing properties that can be demonstrated for the different kind of tasks present in the system. Some protocols have excessively long upper bounds on access costs to globally shared resources. Others, such as those in which the access to globally shared resources is done non-preemptively, present long periods of priority inversion, negatively affecting the behaviour of latency-sensitive tasks.

Finally some protocols, while being sound from the point of view of the task model supported and scheduling capabilities in theory, are not of practical use. This can be due to the excessive high overheads at runtime, such as in global systems, or due to their complex implementation and cost of undergoing validation and verification activities, thus reducing their interest to be industrially adopted.

In the next chapter another multiprocessor protocol enabling the sharing of resources among processors, MrsP, will be presented. This protocol, presents a balanced compromise between access costs to those resources and the priority inversion generated by those accesses. The protocol adopts well-known techniques of monocore practice inheriting some relevant properties that give MrsP an outstanding scheduling performance while allowing an implementation of moderate complexity.

Chapter 4

Introduction to the Multiprocessor Resource Sharing Protocol

4.1 Protocol Definition

Among the proposed protocols for resource sharing in multiprocessor environments, the Multiprocessor Resource Sharing Protocol (or MrsP for short) proposed in [36], has acquired prominent relevance. The similarities between MrsP and the Priority Ceiling Protocol (PCP), in both theoretical approach and scheduling test in the form of Response-Time Analysis [10], have granted MrsP a general acceptance among the Real-Time community.

The intuition behind MrsP is also similar to PCP: resources are given a local ceiling priority, which is equal to the highest priority of a task accessing a resource from a given processor. At run-time, tasks acquire the priority (as active priority) of the ceiling priority of the resource on the processor while accessing it. This, as in PCP, effectively bounds to one the number of tasks that can access the resource concurrently from each processor. In a single processor system, this directly implies that the resource is free and can be directly accessed. However, the same does not hold for multiprocessor systems.

In multiprocessor systems, tasks may be trying to access the resource from more than one processor at a time. This means that: a) the behaviour of waiting tasks has to be defined and b) the accessing order for concurrent requests has to be arbitrated.

Regarding the waiting task behaviour, MrsP is a spin-waiting protocol, i.e. tasks waiting for accessing a resource keep executing in their host processor. This spin-waiting is done at the local ceiling priority of the accessed resource, to maintain the previously mentioned property of only one task accessing a resource at a time from a given processor.

Regarding the access arbitration, under MrsP, tasks are granted access to a contended resource in FIFO order. As a result, the resource access cost can be bounded. If only one task per processor can be accessing a resource at a time and access requests are serviced in FIFO order, then the cost of accessing a resource is equal to the maximum execution time of the resource multiplied by the number of processors from which at least one task accesses that resource.

This access cost bound assumes that the access to a resource is always completed without interruption, i.e. no task is prevented from making progress in the resource when it holds the associated lock. However, this is not realistic: under MrsP, tasks are preemptable when accessing shared resources. As they execute at the local ceiling priority of the resource, any release of a higher-priority task would preempt the accessing task. As a result, not only its access cost but also those from queued tasks might be compromised if progress is stopped on the resource. To prevent this situation, Burns and Wellings [36] introduce a helping mechanism, by which tasks being locally preempted while holding a shared resource lock can be helped by tasks spin-waiting for access to the resource.

The authors define the protocol by the following rules [36]:

1. All resources are assigned a set of ceiling priorities, one per pro-

cessor in which a task using the resource is allocated. The ceiling priority is equal to the highest priority of the task using the resource in each given processor:

$$Pri(r^j, p_k) = \max_{\tau_i : \tau_i \in G(r^j) and\ map(\tau_i) = \{p_k\}} \{Pri(\tau_i)\} \tag{4.1}$$

2. Access request on any shared resource results in the priority of the task being immediately raised to the local ceiling priority of the resource.

3. Pending access request to a resource are serviced in FIFO order.

4. Tasks waiting for gaining access to a resource continue to be active and executing (possibly spinning) with priority equal to the local ceiling of the resource.

5. Any tasks waiting for gain access to a resource must be capable of undertaking the associated computation on behalf of any other task attempting to access the same resource.

6. This cooperating task must undertake the outstanding requests in the original FIFO order.

This definition of the MrsP protocol implies that the PCP protocol is implemented locally on each processor, thus holding its main properties:

- Only one task per processor can be accessing any specific resource.

  This is one of the main properties of PCP, and it is provided by the priority raising when accessing the resource. As the accessing task acquires the ceiling priority of the resource, no task with higher priority using the resource can preempt it.

- The maximum length of the FIFO queue for a resource is equal to the number of processors from which it can be accessed. That is $map(G(r^j))$.

  This property is derived from the "serialization" obtained from the previous property. Then, if only one task per processor can be accessing the resource, it follows directly that the queue length cannot be longer than the number of processors from which the resource is accessed.

- The accessing cost of a resource is equal to the executing time of the resource multiplied by the number of processors from which it can be accessed. That is $e^j = |map(G(r^j))|c^j$.

  As requests are serviced in FIFO order, and the length of the queue cannot be longer than $map(G(r^j))$, a task τ_i accessing the resource r^j may have to wait up to $|map(G(r^j))| - 1$ execution times to gain access to the resource and then spend its own resource execution time accessing it.

- Each job can be blocked at most once, before it starts its execution.

  As MrsP implements a local PCP, this PCP property is also inherited. Under the ceiling protocol, it can be proven by construction that, once a task τ_i starts its execution, no newly released job can preempt it and then access any resource that τ_i may need.

4.2 MrsP Helping Mechanism

As mentioned before, the presented access cost bound only holds if the task holding the resource lock never stops its execution until releasing the lock. This, in some protocols is achieved by just making the access to a shared resource non-preemptable (e.g. MSRP). However, this leads to a suboptimal solution, as it implies unnecessary blocking time for higher-priority tasks (in the form of priority inversion). Instead, Burns and Wellings propose in [36] a helping approach, as stated by rules 5 and 6. Following this approach, in case a task inside a shared resource is locally preemted, two possibilities arise:

1. No other task is waiting (actively) for access to the resource in other processor. The task cannot continue the execution locally, and no other task can help it to continue. This is a classic case of interference, and it should be accounted as such in the Response Time Analysis.
2. At least one task is active and waiting for access to the resource. The task cannot continue the execution locally, but can be helped in other processor to continue the execution. Then, the task execution is resumed in the processor waiting for access, only during the resource execution. For the locally preempted task analysis, it is interfered by the higher-priority task outside the resource. For tasks waiting for resource access, their timing behaviour is not affected by the τ_i preemption, as it frees the resource within the expected time thanks to the helping mechanism.

This helping mechanism is identified as implementable [36] following two different approaches:

- Symbolic execution: if the resource access result is independent from the task accessing (i.e. does not use any local data if the executing task), then the helping task can continue executing the remaining instructions of the preempted access.
- Task migration: if the resource access is not independent of the task accessing the resource, then the preempted task needs to be migrated to processor where it can execute to complete the access and release the resource. This is, in practice, the only feasible approach and will be assumed for the rest of the document.

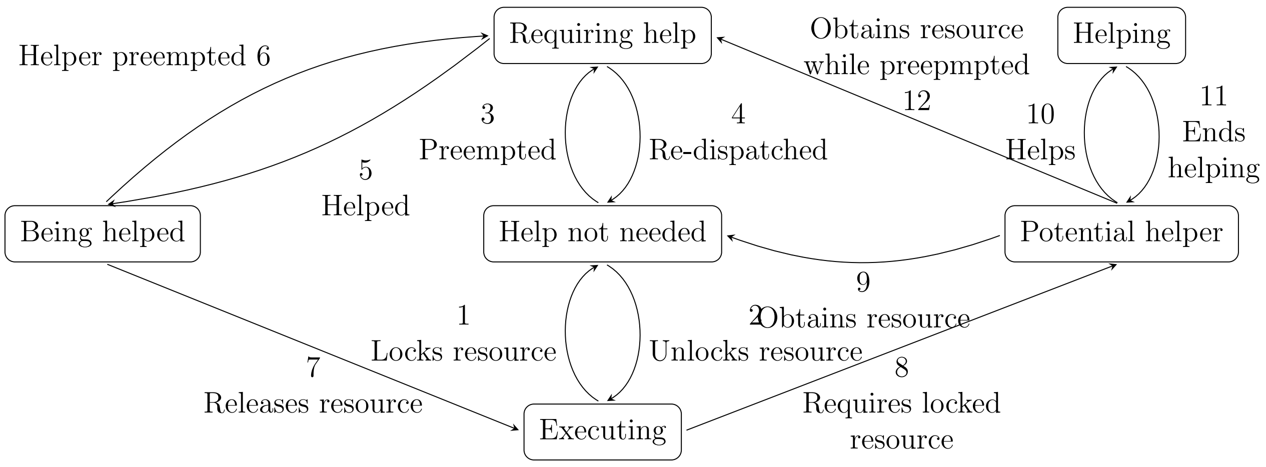


Figure 4.1: Task state diagram of helping mechanism without nested resources. Taken from [71]

Figure 4.1 represents the different logical states in which a task can be with regard to MrsP controlled resources:

- *Executing*: A task that does not require any resource to make progress.
- *Help not needed*: A task is making progress with a locked resource while being dispatched on its host processor by means of its active priority.
- *Requiring help*: A task holding a global resource that is unable to make progress (as it has been locally preempted) from its host processor.
- *Being helped*: A task that holds a global resource and has migrated to another processor in order to make progress.
- *Potential helper*: A task that requests an already allocated resource, and is spin-waiting for it.
- *Helping*: A task that was spin-waiting and pulled a requiring-help task to make progress on its host processor in order to help it to

release the requested resource.

Every task initially holds no resource, so its in the *executing* state. At a certain point, a task can request access to a global shared resource. As part of the process of this request, it increases its active priority to the Local Ceiling Priority of the resource. If the resource is free, it will lock the resource (transition, or tran, 1). Otherwise it will be spin-waiting blocked by this resource until access is granted to the resource (tran 8).

Transition 1, locking the resource, moves the task to the *help not needed* state. While in this state, the task can: finish the access to the resource and release its associated lock (tran 2), or be locally preempted while accessing the resource (tran 3).

If a task is locally preempted while holding a lock, it is considered to be *requiring help* to make progress on the resource. While it remains in the *requiring help* state, no progress is possible. If no other task requires the locked resource while being preempted, then this preemption time is just local interference, and the *requiring help* task will, at some point (when the preempting job terminates), be re-dispatched at its host processor due to its active priority (tran 4).

However, if at some point while being preempted, another task requests access (or was already spin-waiting) to the resource, this task will help the preempted one (tran 5 for the preempted task). This transition, in practice, implies a migration to the helper host processor, with the active priority updated to the Local Ceiling Priority of the held resource on that processor. Then, the task will make progress (*being helped*) until it releases the resource, migrating back to its host processor with its base priority (tran 7), or until it is preempted again on the helping processor, *requiring help* (tran 6) again until it is re-dispatched on its own processor or is helped again.

Tasks blocked by a locked resource are *potential helpers*. Their request is added to a FIFO queue and will be served when all requests in front have been satisfied. This can happen when the task is actually spin-waiting for the resource (tran 9), immediately making progress on the resource, or when the task is locally preempted. As it would hold a resource without making progress due to being locally preempted on its host processor, it would be considered to be *requiring help* (tran 12).

If, while being a *potential helper* due to being blocked by a locked resource, the holder of that resource is locally preempted and thus *requires help*, the helping mechanism is fired. This, in practice means that the *potential helper* task pulls the *requiring help* task to its host processor and lends it its active priority, to execute on its behalf (tran 10). The *helping* procedure ends when the helped task releases the held resource or it is preempted on the helping processor (tran 11).

Thus, for a task to be helped, there should be both a task *requiring help* and a *potential helper* for the same resource. The helping mechanism begins with transition 5 for the *requiring help* and transition 10 for the *potential helper*. Equivalently, the helping mechanism ends with a helped task transitioning by 6 or 7, and a helper doing transition 11.

4.3 MrsP Scheduling

As mentioned before, one of the main outcomes of MrsP is its simple Response Time Analysis based on the resource access cost bound presented before. Formally, for any resource r^j and task τ_i, let $G(r^j)$ be a function returning the set of tasks that access the resource r^j and $F(\tau_i)$ be a function returning the set of resources accessed by τ_i. Finally, let the function *map* be defined by taking a set of tasks and returning the set of processors where the tasks have been allocated.

The number of processors in which tasks using the resource r^j execute is defined as:

$$|map(G(r^j))| \tag{4.2}$$

Then, being c^j the maximum execution time for a resource, the cost of accessing a resource (e^j) is bounded by:

$$e^j = |map(G(r^j))|c^j \tag{4.3}$$

As MrsP inherits the fundamental properties of PCP protocol, including that once a task starts executing, it has all its resources (locally) available, the response time analysis equation for a task τ_i in multiprocessors systems implementing MrsP is:

$$R_i = C_i + \max\{\hat{e}, \hat{b}\} + \sum_{\tau_j \in \mathbf{hpl}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{4.4}$$

where $\hat{e}$ is the maximum execution time of a resource used by a local task with a priority lower than τ_i and a task of equal or higher priority, $\hat{b}$ is the maximum non-preemptive execution time induced by the RTOS, and C_i is the execution time of τ_i itself, that can be decomposed as:

$$C_i = WCET_i + \sum_{r^j \in \mathbf{F}(\tau_i)} n_i e^j \tag{4.5}$$

where $WCET_i$ is the worst-case execution time spent executing code outside shared resources. Then the cost of accessing shared resources is added, where n_i is the number of times the resource r^j is accessed by τ_i per activation.

Migrations Analysis

The previously described response time analysis is based on MrsP helping mechanism. This mechanism ensures that progress is done on a resource as long as some task is executing and attempting to access that resource or set of resources. This is achieved by the task actively waiting for access undertaking the execution of any preempted task blocking its access request. Although in [36] two different mechanisms are suggested, in practice, the helping mechanism is implemented by migrating the preempted task to the processor where the helping task is allocated.

In any architecture, such a migration implies a cost, as certain amount of task specific information has to be transferred from the host processor to the migration target processor. Furthermore, systems using cache memories have an extra overhead, as the task migration implies a lost of cache affinity for both the migrating and helping tasks. Any timing analysis of a system ruled by a protocol allowing migrations should consider such a cost.

In MrsP systems, migrations are triggered by a task holding a resource lock being preempted, i.e. a task with a higher priority than the resource ceiling priority is released. Then, if there is any task spin-waiting in another processor for the held resource, the preempted task is to be migrated to complete its access being helped.

As defined in [36], the active priority of the preempted task is the ceiling priority of the resource held on that processor. Knowing that, the number of preemptions on each processor can be bounded to the number of tasks releases with higher base priority than the local ceiling priority of the resource during the access time:

$$\sum_{\tau_k \in hpl(r^j)} \left\lceil \frac{e^j}{T_k} \right\rceil \tag{4.6}$$

A task, when migrated, can be also preempted on the migration target processor. As a result, the migration cost analysis has to consider the possible migrations due to being preempted on all processors where it can migrate to (processors where the resource is accessed by at least one task, which are known as migration targets). A safe upper bound for the total number of migrations while accessing a resource r^j can be then calculated as:

$$M^j = \sum_{m \in map(G(r^j))} \sum_{\tau_k \in hpt(m, r^j_m)} \left\lceil \frac{e^j}{T_k} \right\rceil + 1 \tag{4.7}$$

where $hpt(m, r^j_m)$ is a function returning all the tasks in processor m with a base priority higher than the ceiling priority of resource r^j on that processor. When a task is migrated, it has, at some point, to migrate back to its host processor, due to being redispatched or releasing the shared resource. In any case, this extra last migration has also to be considered.

As equations 4.6 and 4.7 suggests, theoretically, resources being accessed on processors with many higher-priority tasks may suffer a non-negligible number of preemptions. Moreover, in practice it may happen that a task being migrated to a processor is immediately preempted on that processor, not making any real progress. As a result, it is possible that the resource holder spends a significant amount of time migrating rather than making progress, greatly undermining the usability of the protocol [39].

To avoid this frequent migration problem and improve the efficiency of the helping mechanism, a non preemptive section is introduced in MrsP (MrsP-NP, NP standing for non-preemptive). During this period of time upon a migration, the holder executes non-preemptively before inheriting the ceiling priority of the accessed resource. This offers a trade off between the maximum number of migrations a holder can suffer and the arrival blocking suffered by higher-priority tasks. This new arrival blocking ($\hat{np}$) is included in the updated equation 4.8. Introducing this non-preemptive section, the number of migrations can be upper-bounded by $\lceil \frac{c^j}{C_{np}} \rceil$.

The length of the non-preemptive (C_{np}) section can be tuned as long as the high-priority tasks are able to meet their deadlines. However, different optimizations can be done to reduce the effect of this non-preemptive section for high-priority tasks. As shown in equation 4.8, the arrival blocking is now the maximum between the access of lower-priority tasks to shared resources ($\hat{e}$), and the lengths of the non-preemptive sections imposed by the operating system ($\hat{b}$) and the MrsP non-preemptive sections ($\hat{np}$). As a result, it can be deduced that if the non-preemptive section of MrsP resources is defined so that it is equal to the non-preemptive section of the operating system, it would have no negative effect on any task analysis. Furthermore, in systems where all non-lowest priority tasks suffer any resource-driven arrival blocking ($\hat{e}$), the lowest of this value (normally greater than $\hat{b}$) can be used as a C_{np} value without affecting the timing analysis. Finally, as the worst-case number of preemptions is calculated as a ceiling of the execution time of the resource divided by non-preemptable section, the C_{np} value is optimized when this division is exact.

$$R_i = C_i + \max\{\hat{e}, \hat{np}, \hat{b}\} + \sum_{\tau_j \in \mathbf{hpl}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{4.8}$$

Depending on the task set and length of C_{np} the number of migrations can be significantly lower than the bound given by equation 4.7. Since both approaches give a safe upper bound, the minimum of those values is to be used when calculating the maximum number of migrations per resource access:

$$M^j = \min\{ \sum_{m \in map(G(r^j))} \sum_{\tau_k \in hpt(m, r^j_m)} \left\lceil \frac{e^j}{T_k} \right\rceil , \left\lceil \frac{c^j}{C_{np}} \right\rceil \} + 1 \qquad (4.9)$$

It is worth nothing pointing out that the added cost of migrations for each resource access is then the number of migrations multiplied by the migration cost, *mig*. We consider *mig* to be a constant value for the worst case of a single migration cost. This value will highly depend on the system architecture, and should include all direct (information transfer and tasks re-queueing) and indirect (cache invalidation among others) costs for the given platform.

The total access cost to a resource is then its access cost plus the cost of the migrations, as shown in equation 4.10.

$$MC^j = M^j * mig \qquad (4.10)$$

This value should be finally added to the pure execution time of the resource accessed to compute the final cost of the resource access:

$$e^j = |map(G(r^j))| * (MC^j + c^j) \qquad (4.11)$$

4.4 Summary

The protocol, as presented in this chapter, presents a set of characteristics making it a promising approach for multiprocessor hard real-time systems: it is based on the well known PCP and SRP protocols, its schedulability analysis is simple and it has a well defined approach to upper bound the associated costs of executing in a multiprocessor platform. The cost of sharing a resource among processors is function of the number of processors sharing the resource and the overhead of migrating tasks across processors can be analysed as described.

The protocol presents, however, a number of areas where it can be improved. One of the main improvements to be addressed is the support for nested resources. While the protocol supports the flexible sporadic task model, the protocol definition in [36] vaguely addressed the support for nested resources, undermining the usability of the protocol.

In addition, the protocol also suffers from two common sources of pessimism in the scheduling analysis of multiprocessors systems. One is to consider uniform access times across processors to globally shared resources. Under FIFO spin-locking protocols it is common that access costs to shared resources are obtained by multiplying the worst case access time to the resource by the number of processors from where the resource can be accessed. In systems where the worst case access time is highly differentiated among processors this induces an unnecessary source of pessimism. The other notable source of pessimism is to inflate tasks WCET with the spin-delay result of accessing globally shared resources. In this way, the analysis is oblivious to the probably uneven frequency in shared resources access requests among processors, thus considering unnecessary long FIFO queues and longer spin delays.

The main contributions of this thesis are different extensions to the MrsP protocol that address the mentioned areas of improvement in order to fully unlock the protocol potential.

Chapter 5

Extensions to MrsP

5.1 Heterogeneous Resource Access Analysis

In the previous chapter, MrsP response time analysis equations were derived considering that the access to shared resources is uniform or homogeneous among the tasks using the resource. However, this assumption is far from realistic in many cases: accesses to shared resources are different depending on the internal state of the resource and the interface used to access it. A typical example of this difference is a common shared resource of multiprocessor systems: non-volatile memories. While accessing a memory address for a read operation can be of a similar cost as the main memory depending on the technology used, writing operations are normally of different order of magnitude. For such cases, this difference should not be disregarded.

For a task model considering heterogeneous or non-uniform access times to a shared resource, we can define the access cost of a resource r^j as:

$$e^j = \sum_{p_k} \hat{c}_k^j \tag{5.1}$$

where $\hat{c}_k^j$ is the maximum access time to a resource by a task executing on processor p_k. This approach keeps the interpretation of e^j as the worst-case access cost for a resource, where the task having the longer access time on each processor is queued for accessing it. However, any system having different higher-access times on each processor will benefit from this new analysis.

With this approach, access costs would still be considered equal for all tasks accessing the resource from a given processor. However, the e^j analysis can be tightened even more, up to task level. This is achieved by considering the specific access time to r^j of the task under analysis. Then, the worst-case access cost is sum of the worst-case costs on other processors, plus the access time of the task itself. Thus, the cost of accessing a resource e^j for a task τ_i is:

$$e_i^j = c_i^j + \sum_{p_k \backslash \mathbf{P}(\tau_i)} \hat{c}_k^j \tag{5.2}$$

where $\mathbf{P}(\tau_i)$ returns the processor p_k where task τ_i executes.

The value of e_i^j can be computed for each access of τ_i to r^j. Therefore, equation 5.3 can also be valid for task models considering tasks accessing shared resources from different interfaces. In this way, each access to the resource may have a different cost within the same task, represented as $c_{i,n}^j$, where n represents the $n-th$ access of τ_i to r^j.

$$e_{i,n}^j = c_{i,n}^j + \sum_{p_k \backslash \mathbf{P}(\tau_i)} \hat{c}_k^j \tag{5.3}$$

As the only change to the equations in [36] is the way e^j is calculated, the rest of the equations remain valid, including the calculation of R_i (eq. 4.4), which is still extremely similar to the original equation for PCP.

Given the notion of c^j in [36] as the maximum execution time for

Table 5.1: Non-volatile memory accesses per task activation.

	Task	*Writes*	*Reads*
Core_0	Task_1	1	0
Core_0	Task_2	0	0
Core_0	Task_3	1	0
Core_0	Task_4	2	0
Core_1	Task_5	0	1

a resource, when considering heterogeneous access times for a given resource, other accesses to the same resource have to be equal or lower than c^j. Formally, it means that $\hat{c}_k^j \leq c^j$. As this is the only factor altered in the response time analysis for MrsP, it directly implies that this analysis strictly dominates the one presented in [36].

Example 1

Consider as an example a set of tasks that share a resource that has to be accessed in mutual exclusion and that presents highly differentiated access times depending on the action requested, as can be a non-volatile memory. Table 5.1 presents the task allocation and resource accesses per activation to such resource for the present example. Consider an upper bound of 16 ms for a write operation access. Lets also consider the upper bound of a read operation of 1 ms.

As Task_5 executes alone on Core_1, only read accesses are performed from that processor. Consequently, all write accesses are invoked from Core_0. The system analysis has been conducted following the uniform response time analysis presented at section 4.3, with results presented in table 5.2. The system analysis has also been carried out with the heterogeneous, less pessimistic analysis presented in this section, with the results presented in table 5.3.

Regarding the original analysis, tasks Task_1 to Task_3 suffer 32 ms

Table 5.2: Example response times with original analysis. Tasks are ordered by decreasing priority. All times in ms.

	Task	*T*	*D*	*C*	*Write*	*Read*	*B*	*R*
Core_0	Task_1	100	100	10	16		32	72
	Task_2	200	200	20			32	94
	Task_3	400	400	20	16		32	188
	Task_4	1000	1000	30	2 x 16			354
Core_1	Task_5	1000	1000	100		16		132

Table 5.3: Example response times with heterogeneous analysis. Tasks are ordered by decreasing priority. All times in ms.

	Task	*T*	*D*	*C*	*Write*	*Read*	*B*	*R*
Core_0	Task_1	100	100	10	16		17	44
	Task_2	200	200	20			17	64
	Task_3	400	400	20	16		17	128
	Task_4	1000	1000	30	2 x 16			175
Core_1	Task_5	1000	1000	100		1		117

of arrival blocking, that is the worst-case access cost of one write access from Task_4 (its write access cost plus the pessimist equal write access cost of the Core_1 access). All tasks on both cores present this worst-case access cost for their own accesses.

For the heterogeneous analysis, the pessimism on Core_1 access is removed by considering its real access time of 1 ms. As a result, now the access cost of each access is, in the worst case, the sum of a write operation form Core_0 plus a read operation from Core_1, i.e. 17 ms.

The improvements obtained from the heterogeneous analysis are summarized in table 5.4. These results show an average response time reduction of a 37.14 %, where higher-priority tasks are the most benefited ones with respect to their response times, and lower-priority tasks are the most benefited in net time values, with the exception of

Table 5.4: Example analysis improvement summary.

Task	*Homogeneous*	*Heterogeneous*	*Reduction*
Task_1	72 ms	44 ms	38.89 %
Task_2	94 ms	64 ms	31.91 %
Task_3	188 ms	128 ms	31.91 %
Task_4	354 ms	175 ms	50.56 %
Task_5	132 ms	117 ms	11.36 %

Task_4 which has the highest resource usage and is thus more benefited in both percentage and net values.

5.2 Nested Resources Support

As was introduced in section 2.2.6, shared resources are a common mechanism used in real-time systems for task synchronization. This synchronization enables the implementation of complex systems that could not be otherwise developed without a proper cooperation among different executing entities. Sometimes, the required synchronization cannot be achieved using only one shared resource, or the solution achieved is suboptimal.

The nesting of shared resources, i.e. accessing a shared resource (critical section) from inside another shared resource supports the use of a wider range of solutions. Consider for example a subprogram, requesting an OS operation that has to be executed in mutual exclusion. It should be valid for the subprogram to do so by also ensuring its internal mutual exclusion. In that case, the OS critical section would be nested inside the critical section of the calling subprogram as depicted in listing 5.1.

Allowing nested resource accesses not only improves the task model expressiveness, but also increases its complexity. The specific side effects of nested resource accesses differ from one task model to another, Listing 5.1: Nested shared resource access.

```
protected body Shared_Resource is
        procedure Protected_Action is
        begin
                ...
                Access_OS_Critical_Section;
                ...
        end Protected_Action;
end Shared_Resource;
```

but generic issues can be identified:

- *Deadlocks:* a common issue with nested resources is the occurrence of deadlock situations, in which two or more tasks are circularly waiting in a hold-an-wait situation. Coffman [44] identifies four necessary conditions for deadlock situations to happen: 1) there is mutual exclusion in the use of the shared resources, 2) tasks holding a resource may have to wait for locking a further resource, 3) resource locks cannot be removed from the task holding them until the access to the resource is complete, and 4) there is a circular chain of requests, as depicted in figure 5.1.
- *Livelocks:* when deadlocks are prevented using a "not lock and wait" approach (addressing condition number 2 for deadlocks) but trying to acquire all required locks in a non atomic way, livelocks [12] may occur. In livelocks, affected tasks are not waiting for the required lock to be released but actively and concurrently trying to acquire the required locks without reaching a state in which a task can lock all required resource and make progress.
- *Starvation:* if requests are not satisfied following a fair policy, starvation situations, in which a task is never granted access to the required resource, might arise. For example, if deadlocks and livelocks are avoided by giving higher priority to specific requests based on the resource usage, some other requests may be never

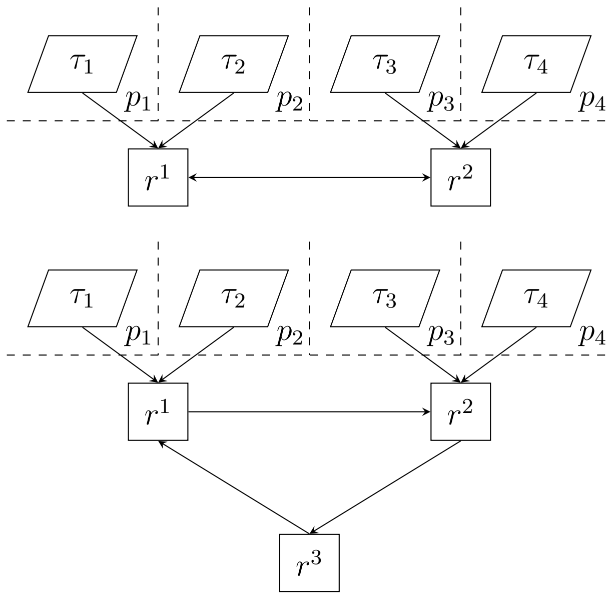


Figure 5.1: Nesting of resources that could lead to deadlocks.

satisfied. Consider a system with 3 tasks and 2 resources. All tasks require both resources in a nested fashion to complete their duties. Task 1 and 2 require the resources for a short period of time, but very frequently. On the contrary, task 3 uses the resources for a long time, but not so frequently. One approach to this problem would be to give priority to requests from tasks 1 and 2 over task 3. However, it may happen that the requests from task 1 and 2 are so frequent that they would never allow task 3 to access the resources.

In order to deal with the mentioned issues, and to generate an analysable task model, two main approaches have been attempted.

Grouping nested resources has been the most common way to address the issues caused by resource nesting. Under this approach, nested resources are locked and released as a whole, i.e. locking a resource locks all resources in the nest. In this way, the task acquiring the lock to enter the nesting of resources has all the inner resources

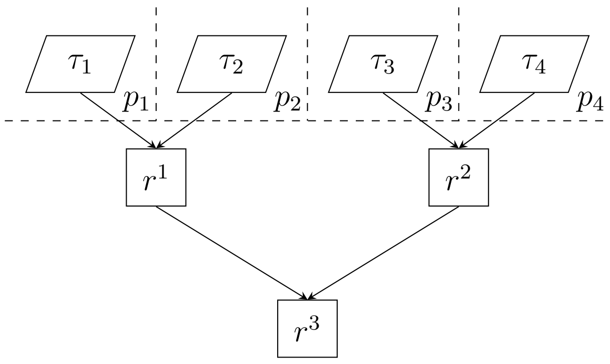


Figure 5.2: Example of resource nesting respecting partial order.

also locked so, if the priority assignment is correct, the task, at some point will finish its access, as no deadlock or livelock situation can arise. However, this approach highly undermines the concurrency of the system, as the resources are locked for a longer time than required. In fact, this solution makes in essence the lock groups equivalent to having one big resource encompassing all of them. Nevertheless, this approach has been the most experienced over the literature, with the Flexible Multiprocessor Locking Protocol (FMLP) [26] being its main representative.

On the other side, fine-grained approach to nested resources is recognized as the most efficient way to support complex concurrency in real-time systems. Proposals following this approach commonly avoid the mentioned nested resources risks by forcing a strict order on locks and releases. The seminal work [120] forced that no lock could be done after a release has been done on the nesting. An extension of this work is the Real-time Nested Locking Protocol (RNLP) [125, 127] which limits the concurrency on nested resource accesses by means of a token mechanism and provides a set of request satisfaction mechanisms aiming for optimality under different system configurations.

Recent work has provided a fully fine-grained blocking bound for nested non-preemptive FIFO spin locks under partitioned fixed-priority scheduling [25]. This is achieved using a novel graph abstraction of the blocking interaction among tasks and resources for which, given a set of invariants stating graph properties, an Integer Linear Programming (ILP) approach is used to find a subgraph yielding a safe worst-case blocking value.

A fined-grained nested resource task model provides better schedulability, since the involved resources are only locked during the effective access time of calling tasks. This improvement is achieved at the cost of more complex synchronization algorithms, which have to ensure deadlock and livelock free definitions, as well as bounded access costs. In the next subsection an extended shared resource model for MrsP supporting a fined-grain analysis for nested resources is presented.

5.2.1 Extended Shared Resource Model

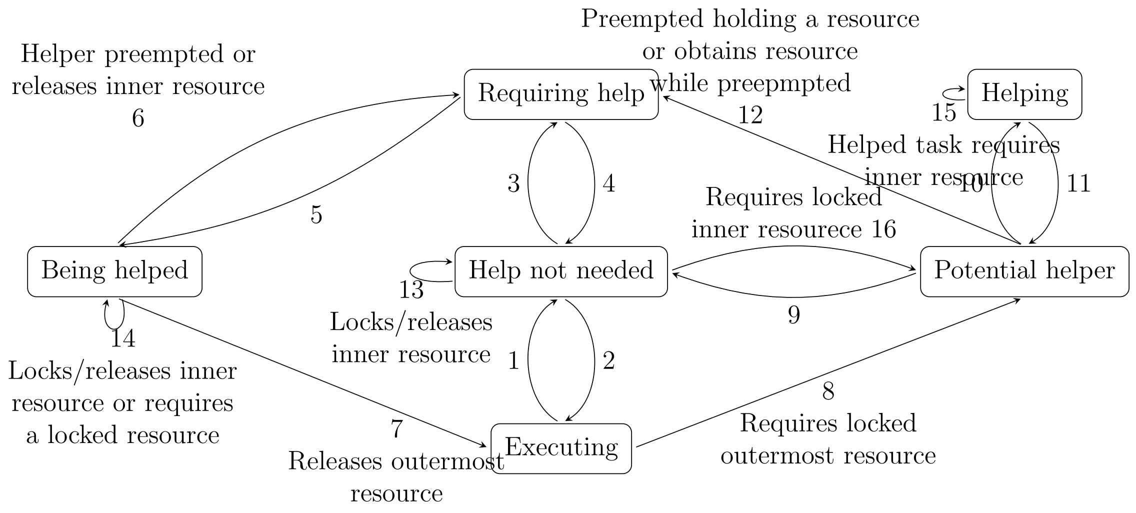

Preempted holding a resource or obtains resource while preepmpted
Helper preempted or releases inner resource
6
Requiring help
12
15
Helping
Helped task requires inner resource
10
11
3
4
5
Requires locked inner resourcee 16
Being helped
13
Help not needed
Potential helper
9
14
Locks/releases inner resource
1
2
Locks/releases inner resource or requires a locked resource
8
Requires locked outermost resource
7
Executing
Releases outermost resource

Figure 5.3: Task state diagram of helping mechanism with nested resources.

Figure 5.3 depicts the different logic states of tasks under MrsP, when considering nested resources. While the states remain the same, new transitions arise and some existing ones are now triggered by new events.

Tasks still begin *executing* without any shared resource, and transitions 1 and 8 are triggered when the task requires the outermost resource of a nested call, raising the active priority to the ceiling of that outermost resource. If the access request is satisfied immediately, the task executes without requiring any help. While executing in the *help not needed* state all locks and releases update the active priority of the task (tran 13) as in PCP. If a lock request finds a resource already locked, the task updates its priority to the local Ceiling Priority of the resource and becomes a *potential helper* for that resource (tran 16).

As with the non nested case, the task can, while *executing not being helped*, be locally preempted and thus *require help* to make progress (tran 3). If at some point while *requiring help*, another task is spin-waiting for one of the resources locked by this preempted task, it will be helped by the spinning task. However, in the nested case, the helper may be helping not due to requesting the inner-most locked resource, but due to requesting any of the resources held by the preempted task.

A task, when migrated to be helped (tran 5), is granted the priority of the helper task. While *being helped*, a task is allowed to lock and release further resources (tran 14), but these actions *do not change the priority of the helper*, and thus the priority at which the helped task is executing while *being helped.*

As with the non nested case, a task can, while *being helped*, release its outermost locked shared resource and migrate back to its host processor with its base priority (tran 7). Similarly, a task can leave the *being helped* state to *requiring help* (tran 6). In the nested case, this transition can be triggered by both the task being preempted on the helping processor, or releasing the required nested resource by the helping task. In this latter case, the task *being helped* still holds other resources, and still *requires help* to make progress.

Any task finding a required resource already remotely locked while *executing* or in *help not needed* state becomes a *potential helper* for that resource (trans 8 and 16). While being a *potential helper* a task can be preempted. In this case, if the task holds a resource, it *requires help* to make progress on that resource (tran 12).

Potential helpers are ready to help tasks *requiring help*, holding their required resource (tran 10). A task while *being helped* may require a locked inner resource. In this situation, the task is still considered to be helped (tran 14) and spin-waits for the locked resource. If the third task holding that inner resource is also *requiring help*, the helping task is ultimately blocked by this third task not making progress. As such, the helper task will also help the third task migrating it to its host processor, and giving it its active priority, executing instead of the task that was *being helped* before (tran 15). This transitive help is maintained until the third task releases the inner resource required by the original helped task.

The helping mechanism can end (tran 11) due to the same two reasons as in the non nested case: the resource required being released or the helper task being locally preempted, with the same implications as in the non nested case.

Formal definition

The detailed approach for MrsP systems supporting nested resources is now presented as a set of rules, lemmas, properties and theorems. Those from PCP and non nested MrsP are assumed and hold unless overridden by those presented here.

Rule 1. Resources under MrsP nest following a strict irreflexive partial order.

Rule 2. A task being helped executes on the helper processor with the helper active priority.

Rule 3. The helping mechanism can be initiated due to the helper task requesting access to any of the resources held by the helped task.

Rule 4. The helping mechanism is transitive, *i.e.* a helper task shall help the locally preempted task that is ultimately preventing it from making progress.

Property 1. A task holding one or more resources, that is not being blocked accessing another resource, will make progress if there is a task spin-waiting due to being blocked by any of the resources held.

This is the fundamental novel property from MrsP that we wanted to extend to nested resources, as it provides the safe upper bound expressed by equation 5.4.

Rule 5. Tasks only modify their active priority when they are dispatched in their own host processor, not being helped.

Tasks can lock and release resources whenever they are executing. If they do so while not being helped, the active priority of the task is modified according to PCP rules. If they do so while being helped, there is no modification of active priorities of the helping or the helped task. The helped task will update its active priority according to the resources held when it is dispatched again on its host processor when its leaving priority is the highest among the tasks eligible to execute.

Rule 6. The helping mechanism shall also be conducted between tasks allocated to the same host processor.

Rule 7. Tasks remain notionally eligible to be dispatched (at their leaving priority) on their host processor while being helped.

By considering tasks being helped and executing on another processor as eligible for dispatching on their host processor, lower-priority tasks are prevented from executing when a higher-priority task would be executing instead.

Lemma 1. A task is only allowed to begin its execution if all higher base priority tasks allocated on that processor are completed.

Proof. If no task is migrated, then all uncompleted tasks are ready to execute on that processor. Following PCP rules, the task dispatched is the one with higher active priority. For a task that has not locked any resource, all tasks with a higher base priority do also have a higher effective priority. As a task cannot have locked any resource before actually executing, it is proven. If a higher base priority has migrated, its leaving priority is at least equal to its base priority. Then, following Rule 7 the higher-priority task would be eligible to execute rather than any lower base priority task not holding any resource. □.

Lemma 2. A task can only make progress by being helped if there is a pending task on the same host processor with higher base priority than its active priority.

Proof. A task having a lower priority than the base priority of another task, will keep having a lower priority unless it locks a resource. Given PCP rules and Rule 7, the lower-priority task cannot be dispatched on its host processor with that lower priority until the higher base priority task is completed. Until then, it can only lock another resource while making progress because of helping. Due to Rule 5, this will not increase its active priority, this will keep it below the higher base priority of the pending task. As a result, a task cannot be executed if not being helped while there are higher base priorities pending tasks. □.

Corollary. Tasks with lower active priorities than pending tasks with higher base priorities cannot increase their active priority.

Proof. Proven during proof of Lemma 2. □.

Lemma 3. Each task can suffer at most a single local block per activation, and this blocking occurs before the task actually executes.

Proof. Lemma 3 in [36] proves this property for MrsP without nested resources, based on the properties of PCP. For nested resources, as tasks are not allowed to increase their priority while migrated, no task can preempt an already higher base priority task.

A higher-priority task may require more than one resource already locked by lower-priority tasks. However, due to PCP rules, only one task could have locked at least one of those resources on its host processor and increase its priority preventing the higher-priority task from executing (arrival blocking). The other tasks only could have locked resources required by the higher-priority task while migrated. As tasks are dispatched on their host processor by their leaving priority, no further arrival blocking owing to lower-priority tasks is possible. □.

Lemma 4. Nested MrsP does not suffer from deadlocks.

Proof. The source of deadlock in nested resources systems is when two or more locked resources are required to complete an operation, being at least two of them already locked by different tasks. By Rule 1 forcing irreflexive partial order, circular dependencies and thus deadlocks due to them are avoided. □.

Lemma 5. A safe upper bound to the number of concurrent access attempts to a resource r^j is given by $|V(r^j)| + |map(G(r^j))|$.

Proof. The number of direct accesses is safely upper bounded by $|map(G(r^j))|$ as all direct accesses from tasks are outermost accesses, and thus are all serviced while not being helped (and migrated), so this directly inherits all PCP properties. As only one request can be generated at a time from each processor, there is an upper bound on the number of processors from where the resource can be accessed. As shared resources have mutual exclusion, only one task can be requesting its inner resource at a time. The number of concurrent requests from outer resources is thus bounded to the number of such resources, i.e. $|V(r^j)|$. □.

Lemma 6. The cost of each individual access (e') to a resource r^j is bounded by $e'^j = c^j + \sum_{r_k \in \mathbf{U}(r_j)} n_j^k e^k$.

Proof. As a consequence of Rule 1, there is at least one terminal resource r^t in the system not accessing any inner resource, i.e. $\mathbf{U}(r^t) = \emptyset$. For such a resource, its individual access cost is:

$$e'^t = c^t$$

From Lemma 5, if we simplify the queue of a resource as $q^j = |V(r^j)| + |map(G(r^j))|$ then the total access cost to e^t is:

$$e^t = q^t c^t \Rightarrow e^t = q^t e'^t$$

For the set of resources accessing the terminal resource, $\mathbf{V}(r^t)$, the individual access cost can be expressed as the execution time of the resource plus the access cost to its inner resource r^t as:

$$e'^{t+1} = c^{t+1} + n_{t+1}^t (q^t * c^t)$$

then substituting e^t:

$$e'^{t+1} = c^{t+1} + n_{t+1}^t e^t$$

And its total access cost can be again expressed as the queue for accessing the resource times the cost of accessing it.

$$e^{t+1} = q^{t+1}(c^{t+1} + n_{t+1}^t e^t)$$

For the 2nd iteration of t outer resources:

$$e'^{t+2} = c^{t+2} + n_{t+2}^{t+1}(q^{t+1}(c^{t+1} + n_{t+1}^t (q^t * c^t)))$$

$$e'^{t+2} = c^{t+2} + n_{t+2}^{t+1}(q^{t+1}(c^{t+1} + n_{t+1}^{t} e^{t}))$$

$$e'^{t+2} = c^{t+2} + n_{t+2}^{t+1}(e^{t+1})$$

Then, for the level k of nesting:

$$e'^{k} = c^{k} + n_{k}^{k-1}(q^{k-1}(c^{k-1} + n_{k-1}^{k-2}(...q^{t+1}(c^{t+1} + n_{t+1}^{t}(q^{t} * c^{t})))))$$

$$e'^{k} = c^{k} + n_{k}^{k-1}(q^{k-1}(c^{k-1} + n_{k-1}^{k-2}(...q^{t+1}(c^{t+1} + n_{t+1}^{t}(e^{t})))))$$

$$e'^{k} = c^{k} + n_{k}^{k-1}(q^{k-1}(c^{k-1} + n_{k-1}^{k-2}(...e^{t+1})))$$

...

$$e'^{k} = c^{k} + n_{k}^{k-1}(q^{k-1}(c^{k-1} + n_{k-1}^{k-2} e^{k+2}))$$

$$e'^{k} = c^{k} + n_{k}^{k-1} e^{k-1}$$

This can be directly applied to resources sequentially requiring more than one different independent inner resources:

$$e'^{k} = c^{k} + \sum_{r^{k-1} \in \mathbf{U}(r^{k})} n_{k}^{k-1} e^{k-1}$$

Theorem 1. Equation 5.4 is a safe upper bound to the cost of accessing a MrsP shared resource and its required inner resources.

Proof. By construction, if Lemma 5 gives a safe upper bound on the number of possible concurrent accesses to a resource r^j and Lemma 6 reflects a safe upper bound on the cost of each individual access to r^j and its required inner resources, then equation 5.4 is a safe upper bound to the cost of accessing r^j. $\square$.

5.2.2 Scheduling Analysis

In this subsection an analysis is proposed in which a safe upper bound can be obtained for the access cost to a resource including any of the inner resources required to complete the access to that resource. To provide such analysis, a strict irreflexive partial order on the resource nesting is required. This not only prevents deadlocks, but also provides an end to the recursion in the analysis, as at least there has to be one resource in the system not requiring any other resource to complete its execution. Given this, the access cost for a nested resource is now defined as follows:

$$e^j = (|V(r^j)| + |map(G(r^j))|) * (c^j + \sum_{r^k \in \mathbf{U}(r^j)} n_j^k e^k) \qquad (5.4)$$

where $\mathbf{U}(r^j)$ is the set of inner resources directly accessed by r^j and n_j^k is the number of times an inner resource r^k is accessed on each access to r^j.

In equation 5.4, the length of the queue is as initially suggested in [36], where PCP limits the number of concurrent access attempts to a resource to one at a time per processor ($|map(G(r^j))|$) and the mutual exclusion nature of shared resources under MrsP ensures that only one access attempt can be performed at a time from any outer resource, giving the total number $|V(r^j)|$. Note that this queue length may be pessimistic, but the objective here is to provide sufficient analysis.

The resource access cost does not only include its own access cost but its cost plus the cost of accessing all the required inner resources. So e^j now represents the full cost for a task accessing nested resources via r^j as an outermost resource (those identified by function $F(\tau_i)$ in equation 4.5), or the cost for outer resources accessing e^j and all its inner resources.

This way of calculating the e value for nested resources now includes the possible transitive blocking on each access. As each access is not considered isolated, but includes the cost of inner resources queues (which are the source of transitive blocking), now equation 5.4 provides a safe upper bound.

Considering the extra blocking a task may suffer due to the helping mechanism, tasks are not allowed to update (increase or decrease) their active priority on their host processor while being helped. This way, lower-priority tasks cannot benefit from the helping mechanism to increase their priority while migrated, with the undesired side effect of causing extra blocking to local higher-priority tasks. In turn, tasks are dispatched on their host processor with the priority they had when they were locally preempted. This priority will be referred to as the *Leaving Priority* for the rest of the document. Migrated tasks *do* update their active priorities when they are re-dispatched on their host processor.

Example 2

To illustrate the approach, the example for nested resources analysis presented in [36] is revisited. Consider a system with four tasks, τ_1, ... , τ_4, executing on four different processors $p_1, ..., p_4$, and two resources, r^1 and r^2, with execution times c^1 and c^2 respectively. Tasks τ_1 and τ_2 access r^1 directly, and τ_3 and τ_4 access r^2 directly. In addition r^1 accesses r^2, so, for example, when τ_1 accesses r^1 it will, while holding

Table 5.5: Task allocation and resource usage of example 2.

Task	Processor	$F(\tau_i)$
τ_1	p_1	r^1, r^2
τ_2	p_2	r^1, r^2
τ_3	p_3	r^2
τ_4	p_4	r^2

Resource	$G(r^i)$	$V(r^i)$	$map(G(r^i))$
r^1	τ_1, τ_2	$\emptyset$	p_1, p_2
r^2	τ_3, τ_4	r^1	p_3, p_4

r^1 also access r^2, as depicted in figure 5.4.

As mentioned above, the nested resource analysis proposed is solved by iteration from inner to outer resources. In this example, there is one inner resource, r^2, and one outer resource, r^1. The accessing cost of the inner resource is (following equation 5.4):

$$e^2 = (1 + 2) * (c^2) = 3c^2$$

Then the cost of accessing the nesting of resources via r^1 can be now calculated, as the cost of accessing all its inner resources (r^2) is known:

$$e^1 = (0 + 2) * (c^1 + e^2) = 2(c^1 + e^2) = 2(c^1 + 3c^2)$$

Now e^1 is a safe upper bound, including transitive blocking, for the access to r^1 and all its required inner resources. Note that an incorrect answer is given for this example in [36].

Nested helping analysis specific cases

With the current definition of local and global resources and ceiling priorities, there are situations in which the analysis can benefit from other priority assignments. Specifically, resources accessed only by tasks allocated to the same processor via outer global resources receive a pessimistic analysis. This pessimism can be reduced and in some

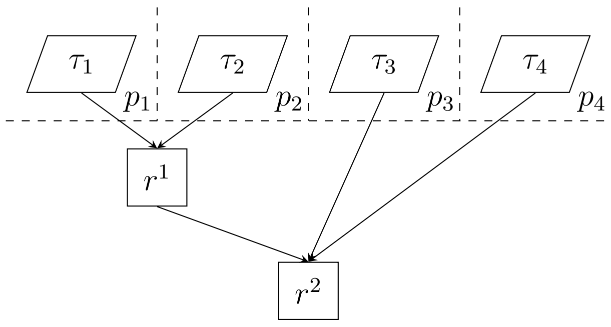


Figure 5.4: Graphical representation of resource usage in example 2.

cases eliminated by a combination of a particular priority assignment (giving global resources encapsulating a call to an inner local resource the ceiling priority of this inner local resource) and the definition of an equivalent task set reflecting the behaviour of the system with that particular assignment of priorities.

Example 3

Consider a system comprising a specific processor P_1 with a task set including, among others (irrelevant for the example) the following tasks: tasks τ_1, τ_2, τ_3 with lowest priorities on P_1, and τ_{10} with the highest priority on P_1. On this processor, there is a set of local resources r_l^1, r_l^2, r_l^3, which are only accessed by tasks τ_1, τ_2, τ_3 and τ_{10}. Task τ_{10} accesses the local resources directly, while τ_1, τ_2, and τ_3 do so via a global resource, different for each of them. These resources are accessed only from tasks from P_1 and another processor, but accesses from the other processor do not generate accesses to r_l^1, r_l^2 and r_l^3. The relevant information for the example is summarized in table 5.6.

Given the analysis presented in table 5.6, the access cost for the highest-priority task τ_{10} of each local resource would be (considering execution times of global resources c_g and local resources c_l): $r_l = 2c_l$, that is, the total access cost for the three resources $r_l^{1,2,3} = 3 \cdot 2c_l$. This analysis assumes that the higher-priority task may have to wait

Table 5.6: Task allocation and resource usage of example 3 without improvement.

Task	Processor	$F(\tau_i)$
τ_{10}	P_1	r_l^1, r_l^2, r_l^3
τ_3	P_1	$r^3 \rightarrow r_l^3$
τ_2	P_1	$r^2 \rightarrow r_l^2$
τ_1	P_1	$r^1 \rightarrow r_l^1$

Resource	$G(r^i)$	$V(r^i)$	$map(G(r^i))$
r^1	τ_1, τ_1'	$\emptyset$	P_1, P_2
r^2	τ_2, τ_2'	$\emptyset$	P_1, P_3
r^3	τ_3, τ_3'	$\emptyset$	P_1, P_4
r_l^1	τ_{10}	r^1	P_1
r_l^2	τ_{10}	r^2	P_1
r_l^3	τ_{10}	r^3	P_1

for the lower-priority tasks on each access to the local resources. This is due to the access of the lower-priority tasks via a global resource. If this was not the case, r_l^1, r_l^2 and r_l^3 would be pure local resources and be completely ruled by PCP. As the lower-priority tasks can be preempted while holding the global resources, and each of them can migrate to a different processor to make progress, the three of them can access their respective local resource concurrently with τ_{10} while being helped remotely. In this case, the helping mechanism produces a high-blocking time for a high-priority task accessing directly to local shared resources. This clearly contradicts the aim and intuition behind PCP and MrsP.

This problem can be addressed by reducing the concurrency of the lower-priority tasks. If r_l^1, r_l^2 and r_l^3 are given the same local Ceiling Priority then only one of the three tasks τ_1, τ_2, or τ_3 can gain access to their outer resource. As a result only one can be helped, and only one can gain access to the inner resource while migrated. The impact on τ_{10} is reduced to a single block.

5.2.3 Migrations Cost

Regarding migrations on the nested resource model, two main differences have to be considered. First, tasks accessing resources can be helped due to the helper being blocked by any of the resources held by the helped task. This includes outer resources of the one actually being accessed as stated by Rule 3. Following general PCP/SRP rules, outer resources should have equal or lower priorities than the resource under consideration. As a result, the safe upper bound needs to cover the case when the task is always helped with the lowest possible priority on each processor. This is, the lowest of the ceiling priorities of all the resources that have the resource r^j as part of its nesting. As such, we define function

$$lcp(r^j_m) = \min_{\tau_i : map(\tau_i)=\{p_k\} \ and \ \tau_i \in G(r^j) \vee \tau_i \in G(r^q) | r^q \in V^k(r^j)} \{Pri(\tau_i)\} \qquad (5.5)$$

as a function returning the lowest ceiling priority of those resources accessing, at any point of the nesting, r^j from processor m. For a resource with no outer resources on a processor, i.e. $\mathbf{V}_m(r^j) = \emptyset$, $lcp(r^j_m)$ returns the local ceiling priority of r^j on that processor. Then, the maximum number of migrations while being helped on processor m is defined in by equation 5.6:

$$Mp'^{r^j_m} = \sum_{\tau_k \in hpt(m, lcp(r^j_m))} \left\lceil \frac{e^j}{T_k} \right\rceil \qquad (5.6)$$

Note that this number is the number of times the task can be preempted during its access to resource r^k and all its inner resources. As the proposed equation 5.4 for calculating the overall access time for a resource e^r already includes the cost of accessing inner resources, this would lead to accounting more than once each possible preemption from each preemptor, on each processor. To prevent that, the number of preemptions when considering the access to r^k is the number of preemptions that may happen during the access to r^k minus those already considered in the access to its inner resources:

$$Mp^{r^j_m} = \begin{cases} 0, & \text{if } \sum_{r^h \in U(r^j)} Mp'^{r^h_m} \geq Mp'^{r^j_m} \\ Mp'^{r^j_m} - \sum_{r^h \in U(r^j)} Mp'^{r^h_m}, & \text{otherwise} \end{cases} \tag{5.7}$$

Note that, as in equation 5.6 we are considering the lowest possible priority for all resource accesses, i.e. the analysed resource and all inner resources, for the analysis matters, the active priority remains constant for the whole access. As such, preemptions are considered to happen at the release time of the higher-priority task and not after a resource lock release (as in the worst case the priority would not change).

The second difference with the non-nested case is the number of processors to where the task can migrate to. In the non-nested case, a task can migrate only to those processors from where a task directly accessed the resource. When considering nested resources, it can migrate to any processor from where a task accesses the resource directly or via any level of nesting. As such, the final equation to calculate a safe upper bound for the number of times a task can migrate during its access to a resource r^j is:

$$Mp^j = \sum_{m \in map(G(r^j)) \cup map(V(r^j))} Mp^{r^j_m} + 1 \tag{5.8}$$

This number of migrations can be reduced if, again, a non-preemptive section is considered after each migration. In this case, it has to be clarified that the non-preemptive section is only granted after a migration, and not by the fact of accessing an inner resource. If the non-preemptive section would be reset for each inner resource accessed, NP sections could overlap posing a greater arrival blocking to higher-priority tasks. Again, possible preemptions included on inner resources are to be subtracted from the analysis. The maximum number of migrations a task can suffer accessing a resource with nested inner resources is:

$$Mnp'^j = \left\lceil \frac{e^j}{C_{np}} \right\rceil \tag{5.9}$$

Subtracting those already included on inner resources analysis:

$$Mnp^j = Mnp'^j - \sum_{r^h \in U(r^j)} Mnp'^h + 1 \tag{5.10}$$

The value to be used in the analysis should be, again, the one from the approach giving a lower number of migrations:

$$M^j = \min\{Mp^j, Mnp^j\} \tag{5.11}$$

Same as for the non-nested case, the way of calculating the cost of these migrations is by multiplying the number of migrations by the cost of each one, as shown in 4.10. Including this analysis to equation 5.4, the final equation to calculate a safe upper bound to the cost of accessing a resource r^k in nested MrsP is as follows:

$$e^j = (|V(r^j)| + |map(G(r^j))|) * (c^j + MC^j + \sum_{r_k \in \mathbf{U}(r_j)} n_j^k e^k) \tag{5.12}$$

5.3 Improved Analysis

In the previous sections, the presented analysis was focused on providing a worst case access time to the resources, based on a limited knowledge of the system. On the analysis presented in section 4.3

task access times to shared resources were calculated considering only a generic WCET value for the access and the number of processors from where the resource can be accessed. As incrementally shown in section 5.1, less pessimistic access cost values can be obtained using extra knowledge of the system. In particular, in section 5.1 different access times were considered for each access of the task under analysis. This has been proven in [71] to provide better schedulability results. In that case those different access times were always assumed to contend for the resource against the worst access time to the resource from the remote processors involved.

However, this assumption is far from real, when the analysis can be performed with a full knowledge of the system. Consider a task accessing a shared resource once per activation, with a period of 1 second. Then consider another task, on another processor, accessing the same resource ten times per activation, with a period of 10 milliseconds. If there are no more tasks in the system, it is clear that the second task would only have to wait for the first one to release the resource once of each 1,000 access requests as much. As such, considering that both tasks would contend for the resource on each access induces a notable pessimism.

In this section an improvement on the MrsP scheduling analysis considering the access pattern of tasks to resources is presented, influenced by a technique known as holistic blocking analysis [27].

Another source of improvement exploited in this section is based on the differentiation between sources of blocking and how are those calculated and included in the analysis. The previously presented response time equations are based on inflating the task execution time with its resource access costs (see equation 4.5). As shown in [128] this is a remarkable source of pessimism, as, given the analysis equations, the interference factor for lower-priority tasks already includes those resource access costs and thus inflating the effects of blocking. In fact, it would increase the influence of impossible blocking situations as those mentioned in the previous paragraphs. Thus, a new analysis approach addressing these issues is required.

The study in [128] presents an analysis technique based on mixed-integer linear-programming (ILP). This analysis, based on a previous approach focused on suspension based locks [28], can be applied to spin-lock based systems, reducing the number of times each shared resource access is accounted to exactly one. The work in [128] extensively analyses spin-lock based systems, considering up to four scheduling algorithms, evaluated for their preemptive and non-preemptive versions. To support such a wide range of implementations, the authors first present a set of general considerations which are complemented by a different number of restrictions to support each specific case.

While the analysis presented in [128] highly improves the schedulability of the analysed systems, it cannot be applied for MrsP, as none of the mentioned cases studied in [128] considers any mechanism similar to MrsP helping procedures. In fact, in [128] only fully partitioned systems, without any possible migration are addressed. Also, the existing ILP based analysis calculates all the blocking effects without distinction, for example, between arrival blocking or spin delay. As such, there is no support for the MrsP helping mechanism, for which a set of conditions have to be defined and checked, in order to provide variable processor time depending on other tasks behaviour.

As a result, and based on the successful application of holistic analysis to spin-locks and influenced by the ILP-based approach [128] a new MrsP specific response time analysis has been developed.

5.3.1 New Scheduling Analysis

As previously mentioned, tasks under MrsP can suffer from three types of blocking, as identified in [128, 138]:

- *Direct spin delay:* a task is considered to be directly blocked when requesting access to a shared resources the associated lock is already acquired by a remote task. For MrsP tasks, as explained in 4.2, tasks incurring in direct spin delay can execute on behalf of the task holding the required lock in case it has been locally preempted on its host processor (helping mechanism).
- *Indirect spin delay:* a task is considered to be indirectly blocked when a higher-priority task on the same host processor is directly blocked. This indirect spin delay is accounted in the previously presented analysis as part of the C_j (executing time of a interfering task) in equation 4.4.
- *Arrival blocking:* a task suffers from arrival blocking when a lower-priority task requests access to a resource with a local ceiling priority higher than the base priority of the arriving (released) task.

The new scheduling analysis is still based on response time analysis techniques and is similar to the one presented in 4.3. The new general equation, however, better reflects the three different sources of blocking:

$$R_i = C_i + E_i + B_i + \sum_{\tau_h \in hpl(i)} (\left\lceil \frac{R_i}{T_h} \right\rceil \cdot C_h + I_{i,h}) \tag{5.13}$$

The response time R_i of a task τ_i is composed by: its pure C_i execution time without accesses to shared resources, the cost of accessing shared resources E_i, which includes the direct blocking suffered, the arrival blocking suffered B_i and the interference of higher-priority tasks, where the cost of each interference is now separated as the pure computation time C_h of the higher-priority task τ_h and the indirect blocking caused by τ_h to τ_i as $I_{i,h}$.

Direct and Indirect Spin Delay

These two factors, E_i and $I_{i,h}$ show a similar rationale and form: they do represent the cost of accessing shared resources by the analysed task τ_i (E_i); and by tasks of higher priority than τ_i executing on the same processor ($I_{i,h}$). This cost is analysed for a specific period of time. In particular, for the analysed task, the cost is computed for a period equal to its response time. As suggested before, the analysis is focused on the specific maximum contention that τ_i can encounter during its potential accesses to a shared resource, i.e. how many requests other tasks can issue to the required resources during its response time. Similarly, for interfering tasks, the analysis should only account for the contention suffered by higher-priority tasks during their real interference over τ_i.

$$E_i = \sum_{r^k \in F(\tau_i)} e_i^k(R_i, 0) \tag{5.14}$$

$$I_{i,h} = \sum_{r^k \in F(\tau_h)} e_h^k(R_i, R_h) \tag{5.15}$$

For both direct (equation 5.14) and indirect (equation 5.15) spin delays, the overall cost is nothing else but the sum of the costs of each accessed resource in the mentioned periods (with R_h jitter in the case of the indirect spin delay).

The cost of accessing each resource e^k by a task τ_x during a period of time l with jitter μ is calculated as the sum of the specific costs of each access. The rationale behind this is that each time the task accesses again the same resource, it will potentially have less contenders for the resource. In other words, the first time a task accesses a resource, in the worst case it will have to contend with all possible processors to access the resource. Then, the following accesses (during the same activation) might not have to contend for the resource with all the processors accessing it, because it might happen that all possible requests from those processors have been already accounted for. Note that this does not mean that on every activation the first access will be the one incurring the most spin delay, but that the worst case is always included in the analysis.

$$e_x^k(l,\mu) = \sum_{n=1}^{N_x^k(l,\mu)} e_x^k(l)(n) \tag{5.16}$$

In equation 5.16 the term $N_x^k(l,\mu)$ is introduced. It represents the number of access requests to resource r^k issued by task τ_x during the period l with jitter μ. Formally, it is the number of access requests τ_x can issue per activation multiplied by the number of potential activations during the analysed period, as shown in equation 5.17:

$$N_x^k(l,\mu) = \left\lceil \frac{l+\mu}{T_x} \right\rceil \cdot N_x^k \tag{5.17}$$

Term $e_x^k(l)(n)$ reflects the cost of the n-th access of τ_x to resource r^k. It is calculated by computing the number of possible contenders for the specific access to be performed. Based on the definition of N, let $Np_m^k(l)$ be the number of requests to resource r^k issued by tasks executing on processor P_m, during the period l. Then let $Nh_x^k(l)$ be the number of requests to resource r^k issued by local higher-priority tasks during the period l. Then, the maximum number of requests on a remote processor that may block directly τ_x is the number of requests

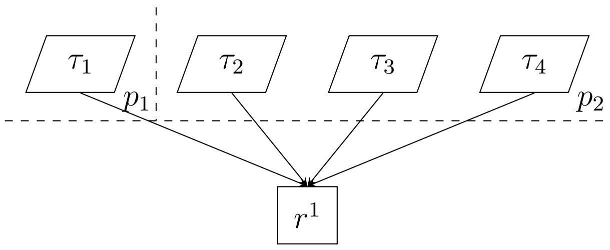


Figure 5.5: Less pessimistic analysis example. If all tasks request the resource with the same frequency, it would be pessimistic to assume that all p_2 allocated tasks would be blocked by a τ_1 request.

from that processor, minus those already suffered by higher-priority tasks during period l (with a minimum of 0):

$$Np_m^k(l) = \sum_{\tau_j \in \tau(P_m)} N_j^k(l, R_j) \tag{5.18}$$

$$Nh_x^k(l) = \sum_{\tau_h \in hpl(x)} N_h^k(l, R_h) \tag{5.19}$$

$$NS_{x,m}^k(l) = (Np_m^k(l) - Nh_x^k(l))_0 \tag{5.20}$$

Then, for each specific access, the cost is the sum of accesses from remote processors (with a cap of one per processor) multiplied by the resource execution time plus the execution time of τ_x itself:

$$e_x^k(l)(n) = \sum_{P_m \neq P(\tau_x)} (NS_{x,m}^k(l) - n + 1)_0^1 \cdot c^k + c^k \tag{5.21}$$

Arrival Blocking

In the holistic analysis, the B_i factor still represents the maximum arrival blocking a task τ_i can suffer, being the maximum between the worst access cost to a resource that can cause arrival blocking to τ_i and the maximum non-preemptive section enforced by the kernel:

$$B_i = \max\{\widehat{e_i}, \widehat{b}\} \tag{5.22}$$

However, the way the $\widehat{e_i}$ value is calculated differs from the sufficient analysis, following a similar mechanism as for the direct and indirect spin delay. First, the set of resources that can cause arrival blocking to a task τ_i is identified:

$$F^A(\tau_i) \triangleq \{r^k | N_{ll}^k > 0 \wedge Pri(r^k, P(\tau_i)) \geq Pri(\tau_i)\} \tag{5.23}$$

This set of resources is composed by the resources to which local lower-priority tasks can issue requests ($N_{ll}^k > 0$) and whose ceiling priority is equal or greater than the priority of τ_i.

Then, the arrival blocking is the maximum value of access costs of those resources identified as potentially blocking:

$$\widehat{e_i} = \max\{(|\alpha_i^k| + 1) \cdot c^k | r^k \in F^A(\tau_i)\} \tag{5.24}$$

where $|\alpha_i^k|$ is the set of remote processors that can cause direct blocking to the local lower-priority task performing the arrival blocking access. An extra access is to be added for the access of the local task itself and multiplied for the executing time of the resource. The way $|\alpha_i^k|$ is calculated is as follows:

$$\alpha_i^k \triangleq \{P_m | NS_{x,m}^k(l) - N_i^k > 0 \wedge P_m \neq P(\tau_i)\} \cup P(\tau_i) \tag{5.25}$$

As shown in equation 5.25, a remote processor is considered as a potential arrival blocker if there are accesses to that processor not already accounted as direct or indirect blocking in the τ_i analysis.

This analysis shows the required convergence as its values are iteratively calculated. As in the original RTA for single core systems, an initial value of C_i is used as R_i and over each iteration R_i is updated to recalculate the interference and blocking until all of them remain unmodified and thus the response time after further iterations.

5.3.2 Migrations

The task model and behaviour of MrsP is not altered due to the use of this improved analysis approach. As such, tasks are able to migrate to remote processors when they are locally preempted while holding a shared resource lock. The improved analysis previously presented also provides better mechanisms to analyse the migration costs.

With the current definition of the protocol and analysis, there is a paradox with the helping mechanism: while it is supposed to "help" the task, from the response time analysis point of view, the response time is not reduced as a result of the task making progress while being interfered, but it is increased to reflect the cost of migrating. While from the point of view of waiting tasks the helping mechanism provides a great improvement, as it bounds the spin waiting time, both waiting and holding tasks response times are benefited when the number of migrations is reduced.

The number of migrations can be reduced if:

- the number of preemptions is reduced, as tasks need to migrate a lower number of times.
- the number of migration targets is reduced to zero. If a task has nowhere to migrate it just can wait to be redispatched on its host processor.

While there is nothing that can be done, from the analysis point of

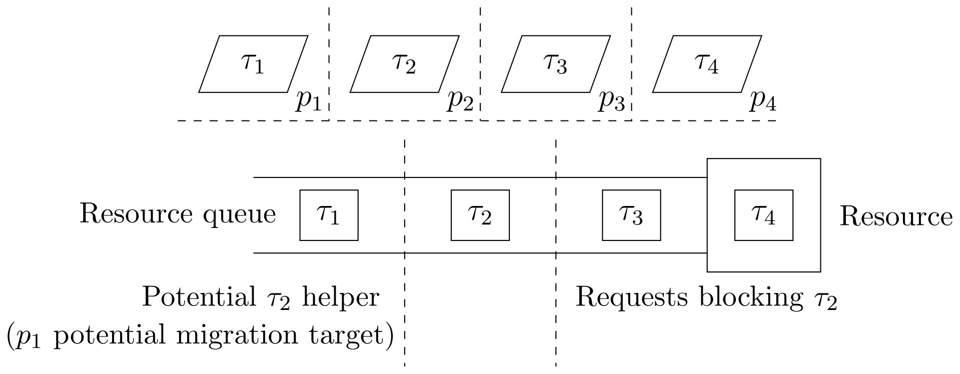


Figure 5.6: Graphical representation of a resource queue, identifying blocking tasks as potential helpers.

view, to reduce the number of preemptions apart from tightening response times, the improved analysis approach can reduce the identified migrations targets. Following a similar approach as for the blocking, potential migration target processors can be discarded if all possible access request to the resource have already been accounted for. In fact, we can say that potential blocking processors are those whose requests are placed before the analysed task in the FIFO queue to access the resource, while potential migration target processors are those whose requests are placed after the analysed task. Of course, a processor can act as both: it can be ahead in the FIFO queue when τ_i request is issued and, after releasing the resource, issue another request, being this time placed at the back of the FIFO queue (migration target section on figure 5.6).

Equation 5.26 composes the set of migration targets for the n-th access of τ_i to resource r^k, as those processors from where access requests can still be issued.

$$mt_x^k(l)(n) \triangleq \{P_m | P_m \neq P(\tau_x) \land NS_{x,m}^k(l) - n + 1 > 0\} \cup P(\tau_x) \quad (5.26)$$

When a task is migrated, it continues its execution on the migration target processor with the local ceiling priority of the resource tried to be accessed by the helper. This is, for non nested resources, the same resource held by the analysed task. As such, we can define a subset of the migration targets where the task may be preempted again, $mtp(mt, r^k)$ as:

$$mtp(mt, r^k) \triangleq \{P_m | P_m \in mt \wedge hpt(r^k, P_m) \neq \emptyset\} \tag{5.27}$$

Using both sets of processors, three specific cases are identified in [138] where the number of possible migrations can be bounded for requests issued from a task τ_i allocated at processor P_m:

- $Nmig = 0$ if $P_m \notin mtp(mt, r^k)$
  There are no possible migrations if there are no potential preemptors on τ_i host processor (the resource has the highest local ceiling priority at P_m).
- $Nmig = 0$ if $\{P_m\} = mt$
  There are no possible migrations if there is no migration target to where the task can migrate to.
- $Nmig = 2$ if $\{P_m\} = mtp(mt, r^k) \wedge |mt| > 1$
  There can be only as much as two migrations if any possible migration target apart from the host processor has no possible preemptor. Then the access request would be completed on the migrated processor and the task migrated back to P_m.

For any other case, the cost of migrations during an access can be calculated as in equation 5.28. This equation calculates the maximum number of potential preemptors on each possible migration target during an access. This is a very pessimistic approach, as it accounts for each possible release, and not for releases that would actually force the task to migrate. Mainly, the pessimism is driven by the fact that there is no way to ensure to which processor the task will migrate to continue executing. This depends on the state of the migration target processors and who pulls the task when the migration situation arises. Unfortunately, the current analysis techniques do not give means to solve those issues.

$$Mhp(mt, r^k) = mig \cdot \Big(\sum_{P_m \in mtp(mt, r^k)} (\sum_{\tau_h \in hpt(r^k, P_m)} \left\lceil \frac{c^k + Mhp(mt, r^k)}{T_h} \right\rceil) + 1 \Big) \tag{5.28}$$

A less pessimistic approach is to use a non-preemptive section after a migration, as introduced in section 4.3. Again, using this approach a safe upper bound (equation 5.29) can be obtained by allowing a minimum amount of execution time to the migrated task to perform progress before being preempted again. This, at the cost of potentially increasing the blocking to higher-priority tasks, provides a less pessimistic analysis under the circumstances explained in section 4.3. Again, this approach is complementary to equation 5.28, so the lower of both M_{hp} and M_{np} values is to be used for the analysis.

$$Mnp^k = mig \cdot (\left\lceil \frac{c^k}{C_{np}} \right\rceil + 1) \tag{5.29}$$

Given the previous considerations, equation 5.30 summarizes all possible and formulas to calculate the migration cost for each access.

$$Mig(mt, r^k) = \sum_{P_m \in mt} \begin{cases} 0, & \text{if } P_m \notin mtp(mt, r^k) \vee \{P_m\} = mt \\ 2 \cdot mig, & \text{if } \{P_m\} = mtp(mt, r^k) \wedge |mt| > 1 \\ \min\{Mhp(mt, r^k), Mnp^k\}, & \text{otherwise} \end{cases} \tag{5.30}$$

Finally, the migration cost has to be included in the overall access cost calculation, that is equation 5.31.

$$e_x^k(l,\mu) = \sum_{n=1}^{N_x^k(l,\mu)} (e_x^k(l)(n) + Mig(mt_x^k(l)(n), r^k)) \tag{5.31}$$

Regarding the arrival blocking, the migration cost is integrated in $\widehat{e_i}$ analysis as follows:

$$\widehat{e_i} = \max\{(|\alpha_i^k| + 1) \cdot c^k + Mig(\alpha mt_i^k, r^k) | r^k \in F^A(\tau_i)\} \tag{5.32}$$

where αmt_i^k is the set of processors with tasks that can cause arrival blocking to τ_i, calculated as $\alpha mt_i^k \triangleq P(\tau_i) \cup \alpha_x^k$

To conclude the analysis of the B_i term, the non-preemptive section has to be included in the analysis. As equation 5.33 shows, only tasks with higher-priority than a local ceiling priority of a resource can be affected (they would have to wait for preempting a lower-priority task accessing such resource).

$$\widehat{np_i} = \begin{cases} C_{np}, & \text{if } Pri(\tau_i) \geq min_{\{r^k \text{ is global}\}} Pri(r^k, P(\tau_i)) \\ 0, & \text{otherwise} \end{cases} \tag{5.33}$$

As in the sufficient analysis, the value to consider for the response time is the maximum between the arrival blocking, the MrsP non-preemptive section and the non-preemptive time impose by the underlying kernel.

$$B_i = \max\{\widehat{e_i}, \widehat{np_i}, \widehat{b}\} \tag{5.34}$$

5.3.3 Nested Resources

As discussed in section 5.2, support for nested resources is a basic requirement nowadays for realistic high-integrity systems. The increasing complexity of the systems under control, and thus of the required approaches for the correct synchronization of processes make nested resources an unmissable tool.

In section 5.2 a reformulation of MrsP support for nested resources was presented [71]. This included a set of rules to properly and deterministically define the behaviour of MrsP tasks accessing nested shared resources.

In this work, a sufficient analysis for the mentioned behaviour was also presented. In this section a new analysis based in the improved analysis is explored.

It has to be borne in mind that an analysis approach is only a different way to evaluate the schedulability of a system using a different method, but the system is not changed due to the analysis itself. As such, this section will only cover how the improved analysis approach can be applied to the protocol definition in section 5.2 but will not change it in any way.

The sufficient analysis for MrsP nested resources (without considering the migration cost) differed from the non-nested resource analysis in two main points:

- *Resource contention*: although we are not considering yet the migration cost, in order to properly analyse the accessing cost to nested resources, it is important to beard in mind that tasks can migrate due to the helping mechanism. As demonstrated in section 5.2, this can lead to a higher number of accesses to a shared resource than just $|map(G(r^j))|$. In the case of nested resources, it was demonstrated that the number of possible concurrent accesses to a shared resource can be as high as $|V(r^j)| + |map(G(r^j))|$.
- *Transitive blocking*: as resources are accessed sequentially, and progress on a resource can depend on third party task accesses,

the problem of transitive blocking arises, as described in section 5.2. This is accounted for by the recursive form of equation 5.4, where the cost of accessing a resource is defined as the cost of accessing that resource itself plus the cost of accessing its inner resources.

Regarding the resource contention, as demonstrated in [138], the worst case happens when the higher-priority tasks suffer the maximum possible contention. This is because, even with the improved analysis, the contention suffered by higher-priority tasks is propagated to lower-priority tasks (as indirect spin delay).

For the non-nested resources improved analysis, this is addressed by considering that, for each access, there will be the maximum number of possible tasks queued, i.e. one for each processor from where at least one task can perform a task not already accounted for.

In the nested case, there are two issues with that:

- *Maximum number of access requests*: In the non nested case, an access request can only be issued in the host processor of the requesting task. In the nested case, this can potentially happen from any processor in the system, as the task can issue the access request from a processor where it is executing being helped (calling from inside of another processor). This value has been shown to be $|V(r^j)| + |map(G(r^j))|$. However, this can be pessimistic when this value is greater than the number of tasks that can access the resource. It is clear that, no matter from where the resource is accessed, there can be no more simultaneous requests than the number of tasks that at any time during its execution can request the resource. Then we can define Q^k as the maximum number of simultaneous access requests to a resource r^k as:

$$Q^k = \begin{cases} |map(G(r^k))| & \text{when } |V(r^k)| = 0 \\ \min\{|G(r^k)|, |V(r^k)| + |map(G(r^k))|\} & \text{otherwise} \end{cases} \tag{5.35}$$

where the value of Q^k is the number of processors from where it has direct accesses if the resource is purely an outer one.

- *Which requests are issued*: in the non-nested case, the maximum number of concurrent access requests to a resource was equal to the number of processors that can issue requests to that resource. One processor, one access request. As said before, this is not the case for nested resources.

  On the contrary, in nested resources, for each access request, all possible contenders can be waiting for the resource, regardless of their host processor.

  As a result, we have to redefine equation 5.20, as it cannot longer represent the contenders on a specific processor, but all contenders in general. Then, $Nh_x^k(l)$ still represents the requests performed by local higher-priority tasks to resource r^k, during time l, as defined in equation 5.19. Let now $Nr_x^k(l)$ be the number of requests performed by all other tasks in the mentioned l period, calculated as in equation 5.36:

$$Nr_x^k(l) = \sum_{\tau_j \neq \tau_x} N_j^k(l, R_j) \tag{5.36}$$

  Now, the maximum number of contenders for an access request from task τ_x to resource r^k is defined by equation 5.37:

$$NS_x^k(l) = (Nr_x^k(l) - Nh_x^k(l))_0 \tag{5.37}$$

  With this reformulation of the possible contenders, the way the actual cost of accessing the resource r^k on each access n is:

$$Q_x^k(l)(n) = (NS_x^k(l) - ((n-1)\cdot(Q^k - 1)))_0^{Q^k-1} \tag{5.38}$$

where to the total number of remaining requests after accounting those suffered by higher-priority tasks ($NS_x^k(l)$) is subtracted the number of already accounted in previous accesses of task τ_x. As the specific access under analysis is the n-th, those already performed are $n-1$. For each previous accesses, the maximum number of simultaneous access requests $Q^k - 1$ are subtracted. The minus one is because one of the simultaneous accesses is the one of the analysed task itself (not included in $NS_x^k(l)$). Finally the number of accesses that can cause spin delay has a minimum value of 0 and maximum of $Q^k - 1$ (all possible tasks that can access the resource).

As equations 5.36 to 5.38 show, if the longest possible queue is smaller than the number of tasks accessing the resource, it does not matter which specific contender tasks are accounted on each specific access when considering homogeneous access times to resources. A desirable improvement to this analysis would be to detect situations in which only one task can perform the remaining accesses to the resource. Unfortunately this is not trivial. If only one other task can ever access the resource then there is no improvement, as it is already limited by $Q^k - 1$. In any other case (where all other tasks accesses have already been accounted) finding the tighter, safe worst-case order of accesses has the form of a np-hard problem.

Having considered the different access patterns to nested shared resources resulting in equation 5.35 by calculating the worst-case queue to be found on each specific access, a way to analyse the cost of accessing the resource is still required.

As shown in section 5.2, the accessing time of a nested resource is the time required to execute the instructions belonging to that resource itself plus the accessing cost of the inner resources accessed:

$$e'^{k}_{x}(l)(n) = c^k + \sum_{r^j \in \mathbf{U}(r^k)} \sum_{n=1}^{N^j_k} e^j_x(l)(n)) \tag{5.39}$$

Equation 5.39 considers, for each resource, that the inner accesses performed are the first done by the task to the resource. It is so because no assumption is made about the order the task accesses the resources. In the nested case, this order matters, as the cost of accessing outer resources depends on inner resources, and the queues they might have. As a consequence, assuming a certain order would result in a variable, sum of net shared resources accesses. This could be alleviated if:

- There is actually a knowledge of the resource accessing order of each task in the system.
- All possible orders are calculated and the worst case of the sum of all accesses is considered.

In any of these cases, e'^{k} could be calculated as:

$$e'^{k}_{x}(l)(n) = c^k + \sum_{r^j \in \mathbf{U}(r^k)} \sum_{n=N(\tau_x, r^j)+1}^{N^j_k + N(\tau_x, r^j)} e^j_x(l)(n)) \tag{5.40}$$

where function $N(\tau_x, r^j)$ returns the number of already performed accesses to r^j by τ_x.

Regardless of the way e'^{k}_{x} is calculated, the resource accessing cost is the maximum number of contenders for accessing the resource plus one (the analysed task access) times e'^{k}_{x}, as shown by equation 5.41.

$$e^k_x(l)(n) = (Q^k_x(l)(n) + 1) \cdot e'^{k}_{x}(l)(n) \tag{5.41}$$

Equation 5.41 can now be integrated in equation 5.16 to calculate the resource accessing cost and, as a result, the direct and indirect blocking suffered by a task.

With regard to the arrival blocking, equation 5.23 still identifies the resources that can cause arrival blocking to a task τ_i. These resources are those used by local lower-priority tasks having a local ceiling priority equal or higher than τ_i.

However, the way the cost itself is calculated in equation 5.24 does not hold for nested resources. As shown before, the accessing time of a nested resource is not only the cost of executing its related instructions but also the cost of accessing all the required inner resources. As well, the way the length of the queue is calculated in equation 5.25 needs to be reviewed as, as said before, the maximum number of concurrent access requests is not limited by the number of processors from where the resource can be directly accessed but also by the number of outer resources accessing it and the total number of tasks that can issue requests to that resource.

Then, identified the resources that can cause arrival blocking to τ_i, and for those resources, $\widehat{e_i}$ can be defined as the maximum accessing costs to potentially arrival blocking resources in $F^A(\tau_i)$, considering the next access to be accounted for after all those from τ_i have been considered:

$$\widehat{e_i} = \max\{e_i^k(R_i)(N_i^k + 1) | r^k \in F^A(\tau_i)\} \tag{5.42}$$

As it can be noted, there can be the case where τ_i does not access the resource r^k. In such a case $N_i^k = 0$ and only the accesses from higher-priority tasks are deduced to the total number accesses to r^k to determine the maximum blocking suffered by an arrival blocking request. In case τ_i does access r^k the blocking to account for is that of the next access to be performed to the resource.

Migrations

As previously presented, the migration cost analysis is based on two main identifications: the migration targets (where a task can migrate to complete an access request to a resource), and the preemptor tasks on those processors triggering the migrations, including those tasks on the analysed task host processor.

With regard to the migration targets, a task can migrate to a processor from where there is still at least one access request pending on that resource to the analysed resource. Two important things are to be borne in mind while considering the migration targets for nested resources: first, for the improved analysis, we consider that each access to outer resources comprises all the required accesses to its inner resources. This means that if r^o is an outer resource that has an inner resource r^i, all accesses to r^o imply an access to r^i. The second one is, in part due to the previous fact, that no request can be performed from a processor to a resource where all possible requests have been already satisfied. This is because: a) all access requests from tasks hosted at that processor have been satisfied, b) no task can be migrated to that processor because there is no task that can be actively waiting to access the resource or any other involved on a nesting where the analysed resource is involved. As a result, the migration targets of τ_x while accessing for the n-th time resource r^k are still properly identified as $mt_x^k(l)(n)$ in equation 5.26.

In the same fashion, it is required to identify the potential preemptors on those migration targets. In section 5.2.3 a specific approach for analysing nested resources migrations is presented. In it, it is said that, as a result of the nested accesses and migrations, the active priority of a task while accessing an inner resource at a remote processor may not be the resource local ceiling priority, but in the worst case, can be as low as the lowest ceiling priority of a resource on that processor that can be held to be a potential helper for the analysed resource.

This lowest priority of a resource r^k in processor P_m is denoted as $lcp(r^k_m)$ as defined in equation 5.5. As a result, a migration target has preemptors if there are tasks with priorities higher than the mentioned $lcp(r^k_m)$.

$$mtp(mt^k_x(l)(n), r^k) \triangleq \{P_m | P_m \in mt \wedge hpt(lcp^k_m, P_m) \neq \emptyset\} \tag{5.43}$$

With the migrations targets and preemptors identified, the total number of migrations a task can suffer can be identified. Firstly, the exceptions for specific combinations of $mt^k_x(l)(n)$ and $mtp(mt^k_x(l)(n), r^k)$ identified in section 5.3.2 hold for nested cases and can be applied.

For a general case, the number of migrations a task can suffer on its access to a shared resource in a given processor is:

$$Mp'^{r^k_m}(l)(n) = \sum_{\tau_h \in hpt(m, lcp(r^k_m))} \left\lceil \frac{e^k}{T_h} \right\rceil \tag{5.44}$$

Similarly to the sufficient analysis, $Mp'^{r^k_m}$ is the total number of potential preemptions during the whole access to a resource. This includes the possible preemptions that may happen also during the access to inner resources of r^k. As these are already included due to the recursive way of calculating e^k, the total number of possible migrations to account on the specific access to e^k are those that exceed the already suffered in the accesses to inner resources, as in equation 5.45.

$$Mp^{r^k_m}(l)(n) = Mp'^{r^k_m}(l)(n) - \sum_{r^h \in U(r^k)} Mp'^{r^h_m}(l)(n) \tag{5.45}$$

Then, the total number of migrations is the sum of those $Mp^{r^j_m}$ to all possible migration targets plus the final migration to the host processor:

$$Mp^k(l)(n) = \sum_{m \in mt_x^k(l)(n)} Mp^{r_m^k}(l)(n) + 1 \tag{5.46}$$

As in the previous analysis, the number of migrations can also be calculated based on a non-preemptive section. If it is implemented, a certain amount of execution time is granted to a task when resumed on another processor due to a migration. For the holistic nested case, this is computed as:

$$Mnp'^k(l)(n) = \left\lceil \frac{e^k(l)(n)}{C_{np}} \right\rceil \tag{5.47}$$

and should again subtract those already accounted for in the inner resources:

$$Mnp^k(l)(n) = Mnp'^k(l)(n) - \sum_{r^h \in U(r^k)} Mnp'^h(l)(n) + 1 \tag{5.48}$$

As both $Mp^{r_m^k}$ and Mnp'^k are safe upper bounds, the value that should be considered for the number of migrations when calculating the overall cost of the access to r^k is the minimum between both values.

$$M^k(l)(n) = \min\{Mp^k(l)(n), Mnp^k(l)(n)\} \tag{5.49}$$

As in all previous analysis approaches, the way of calculating the migrating cost per access to the resource MC^k is by multiplying the worst case of migrations by the cost of each migration as in 4.10.

$$MC^k(l)(n) = mig \cdot M^k(l)(n) \tag{5.50}$$

Incorporating this analysis in equation 5.4, the final equation to calculate a safe upper bound to the cost of accessing a resource r^k in nested MrsP with a holistic approach is as follows:

$$e_x^k(l)(n) = (Q_x^k(l)(n) + 1) \cdot (e_x'^k(l)(n) + MC^k(l)(n)) \qquad (5.51)$$

This new way of calculating $e_x^k(l)(n)$ shall also be used in equation 5.42 to calculate the arrival blocking value of $\hat{e}_i$.

5.4 Semi-Partitioned Systems

The previous sections have addressed a protocol definition for fully partitioned systems, as defined in section 3.2.1. These systems have been the most studied and implemented. This is because of their predictability and relative ease of validation and verification. However, they present some disadvantages. One of such disadvantages is the underutilization of the processors as a result of the need to fully allocate tasks to a single processor.

On the contrary, in semi-partitioned systems 3.2.3, some tasks can be set to execute on more than one processor, obtaining the required execution time per activation from the sum of smaller processors utilizations. As such, they can "fill" the remaining execution times on each processor, increasing the overall utilization.

Approaches of this kind have been disfavoured in the past mainly due to the need of migrating tasks from one processor to another during their execution. As this migration has a cost, the schedulability analysis is affected. Moreover, the feature of migrating a task from one processor to another needs to be implemented by the underlying kernel, what is not normally the case for real-time kernels.

MrsP systems, however, are in some sense semi-partitioned, as tasks can execute (while being helped) in more processors than just their host one. As such, the notion of migrating tasks is intrinsic to MrsP systems and needs to be implemented. The proposed approaches in this section further expand this notion of migration not only to temporally migrate a task during a resource access but to change the host processor of the task during an activation.

The set of tasks that can do this kind of migrations are considered shared or split tasks as explained in section 3.2.3. A set of desired characteristics for the scheduling of shared tasks has been identified [68]:

- Maximize the processor utilization: the main objective of Semi-Partitioned systems is to maximize the utilization of available processor time not used by statically allocated tasks. As such, the scheduling algorithm should be aligned to this objective.
- Have an effective schedulability test.
- As far as possible, exploit the well-known and proven scheduling algorithms and properties developed for single-processor real-time systems. One of the main contributions of MrsP is its effective translation of well understood properties of single-processor resource control policies to multiprocessor systems.
- Produce as few migrations as possible. Although the main benefit of Semi-Partitioned systems arises from the possibility of executing specific tasks on more than one processor, migrations inherently pose an overhead. Thus, a scheduling algorithm in which shared tasks are more likely to complete each activation duty on a single processor is preferred.

The identified benefits and desired properties have to cope with two main problems when addressing semi-partitioned systems applied to approaches like MrsP:

- Task scheduling: as identified in section 3.2.3, there is a range of possibilities on how to schedule the shared tasks and how do they relate with the statically allocated tasks. MrsP on its studied version implements a FIFO within priorities scheduler with priorities assigned using ceiling priorities. The way shared tasks are scheduled should be one so that the impact on statically allocated tasks is deterministic and analysable.

- Resource access origin: one basic characteristic of MrsP is that the analysis is based on the knowledge of the system with regard to the shared resource accesses. Sufficient analysis assumes a limited knowledge of just the task allocation and resources accessed. The improved analysis bases its improvements on the knowledge of the number of times shared resources can be accessed by both local and remote tasks. In a semi-partitioned approach this posses a challenge. Since shared tasks can be hosted not being helped on different processors, the $map(G(r))$ function turns to provide a dynamically changing result. How shared tasks are added to MrsP systems should allow to know, at design time, the required information to perform these analysis.

5.4.1 Global Scheduling with EDF

An initial approach to MrsP-like semi-partitioned systems is presented in [68]. In this work, a set of different approaches to shared task is explained, followed by scheduling considerations based on global scheduling schemes. In particular, Global-EDF is suggested for the shared task scheduling, allowing statically allocated task to maintain the fixed-priority schedulers.

In these approaches, the Global-EDF scheduler is modelled as a lowest-priority task on each processor with regard to the statically allocated tasks. As such, the effects of shared task migrations and influence in the response time analysis are bounded. In fact, the simplest approach to MrsP semi-partitioned systems is to share tasks not accessing shared resources.

As shown before, one of the main advantages of MrsP is the upper bounding of the number of concurrent accesses to a shared resource. This upper bound is the number of processors from where the resource can be accessed (we will not refer here to nested resources).

As a result, this safe upper bound is not affected if tasks not sharing any resource are added to the system. Furthermore, if these tasks are given lower priorities than tasks sharing resources, the response time analysis of the latter ones is not affected, even if those tasks are allocated to their same processor.

Lemma 7. Response time analysis of tasks under MrsP is not affected by tasks with lower priorities not accessing shared resources.

Proposal 1. Tasks not sharing resources can be dynamically allocated to the same processors as shared tasks within Semi-Partitioned systems scheduled with MrsP without affecting the schedulability of statically allocated tasks as long as the latter ones are assigned a higher priority.

Even not affecting the higher-priority (statically allocated) tasks response times is a desirable property, a much higher flexibility can be acquired with a low overhead on these tasks.

Another approach, relaxing the restrictions of use of shared resources, can have a bounded effect if the upper bound of parallel accesses to shared resources, the $|map(G(r^j))|$ factor in the MrsP response time analysis is maintained.

Lemma 8. The $|map(G(r^j))|$ factor for each resource is not affected by the addition of tasks to a system as long as the added tasks do not use different shared resources than the already allocated tasks to processors where added tasks can execute.

Lemma 9. The addition of tasks not incrementing the $|map(G(r^j))|$ for any resource only affects the response time analysis of tasks executing on the same processor as added tasks.

Lemma 10. The addition of lowest-priority tasks to a processor only affects the response time analysis of the lowest-priority tasks already allocated using the same shared resources as the added task, assuming homogeneous resources access times as in [36].

Proposal 2: Shared tasks can be dynamically allocated to processors where all the resources used by the shared tasks are already used by statically allocated tasks. Shared tasks must be assigned lower priorities than any statically allocated task. Lowest priority-tasks statically allocated using the resources also used by shared tasks must add the blocking time due to the shared tasks accesses to those resources.

With this proposal, only the lowest-priority tasks among the statically allocated tasks are affected by the shared tasks. However, a greater flexibility is obtained, being possible to share tasks using shared resources.

The scheduling approach suggested for both proposals is, as said before, a Global-EDF scheduler [54] for the shared tasks. It has an effective schedulability test for the task set restrictions previously explained, specially proposal 1, where no shared resources are involved in shared tasks. The maximum processor utilization available for shared tasks can be obtained by subtracting the processor utilization of allocated tasks by the response time analysis proposed in [36] to the total processor utilization available. Another possible way to calculate the execution capacity of shared tasks under the proposed systems is proposed in [84]. Finally, in general terms, the resulting system with regard to shared tasks has similarities with levels *C and D* of the system model presented in [98]. As such, the analysis given in [93] by using the service functions for each processor proposed in [41] is also being considered.

Unfortunately, a set of facts make Global-EDF not the best choice for a strict real-time system based on MrsP. First, the main drawbacks of EDF in mono processor systems are also translated to multi processor systems: there is not specific determinism on which tasks will fail first to meet their timing requirements under EDF and once tasks start missing deadlines, there can be a domino effect by which the whole system can start missing deadlines in a waterfall fashion.

Moreover, an in-depth approach to an EDF-MrsP analysis is required to evaluate approach 2. While there are approaches to analyse systems with shared resources and EDF systems, and while MrsP is not necessarily restricted on task allocation (as demonstrated by the viability of semi-partitioned approaches) or priority assignment, a validated EDF MrsP analysis is required in order to provide a consistent proposal based on Global-EDF as scheduler for shared tasks.

5.4.2 Deterministic Allocation

As explained before, due to the MrsP nature and its schedulability analysis, a task assignment scheme in which the set of processors where the task is going to be shared is as reduced as possible and known at design time is preferred. Among those reviewed in section 3.2.3, two protocols gather the required characteristics identified to be build semi-partitioned MrsP systems based on them: DM-PM and C=D.

DM-PM, as reviewed in section 3.2.3 assigns tasks to processors until the processor capacity is filled. Then, if there are remaining tasks to be allocated that cannot fit on any processors without being split, they are allocated to consecutive processors until they are fully allocated. Two shared tasks can have part of their execution allocated to the same processor as the last fraction of a shared would rarely exactly fill the remaining capacity of the last processor where it is allocated.

Tasks are assigned priorities in Deadline Monotonic order being the shared tasks the higher-priority tasks on each processor (tasks are assigned in decreasing relative deadline, so shared tasks are ones having higher priority). When two shared tasks have fractions allocated to the same processor, the later allocated task has greater priority.

During execution, shared tasks begin the execution on the processor with lower index and execute until the assigned capacity is consumed. Then, they are migrated to the next index processor to continue the execution, and so until the execution is finished.

As a result, the execution of shared tasks is highly deterministic under DM-PM: the execution starts always at the same processor, and migrates after a controlled amount of time to the next processor, which is also constant. As a result, the affinity of resource accesses can be statically determined at design time.

Another interesting protocol to consider for building semi-partitioned MrsP systems is the C=D partitioning scheme defined in section 3.2.3. This partitioning scheme although developed for EDF scheduled systems has interesting properties for MrsP systems. Specifically, tasks are only split in two processors, as the allocation process is slightly different to DM-PM: while in DM-PM all processors are first filled with fully allocated tasks and then the remaining task are split filling the processors, in C=D each processor is filled before start allocating tasks to the next one. Once no more tasks can be fully allocated to a processor, the next one is split so that it completely fills that processor and the rest of the task is the first to be allocated on the next processor. As a result, a task is only split in two processors and only $m - 1$ task can be split in total.

Under C=D partitioning, split tasks have the highest priority on their first processor, and then are migrated to the second one. The priority on this second processor depends on the task set. This partitioning scheme has the benefit of reducing the number of processors where split tasks execute and thus the potential cardinality of the resources accessed by them.

5.5 Summary

In this chapter a number of extensions to the MrsP protocol have been presented. First, a common source of pessimism in resource sharing cost analysis in multiprocessors has been addressed. By considering all operations on shared resources equal in the access time required to complete a certain degree of pessimism is introduced in access costs calculations. This is especially true for hardware resources common in real-time systems, such as non-volatile memories. In this chapter the problem nature has been analysed and the basis for a generics solution for FIFO spin-waiting protocols have been provided.

Then, the support for nested resource accesses has been addressed. Common approaches to nested resources policy accesses have been reviewed and the need for supporting fine grained locking has been identified. Then, the consequences of that support have been analysed, with particular attention to the effects on MrsP helping mechanism. A set of rules have been defined regulating the addressed support and certain properties have been demonstrated for the resulting model. Among these properties is an access cost calculation equation that has a similar structure as that of non-nested shared resources and allows the previously presented heterogeneous resource access cost analysis to be applied.

With the complete task and resource model defined an improved scheduling analysis is attempted. This new analysis maintains a similar form to that of the sufficient MrsP response time analysis, but separating the spin delay from the task worst case execution time. This, in combination with a method to calculate the maximum number of resource access requests a task can issue during a time frame, enables the spin delay due to a given remote access request not to be accounted for more than once on a task response time calculation. Now, each possible remote access request can only cause to a task either a direct spin delay (the task spin-waits for the resource), or an indirect spin delay if it is a higher priority task which spin waits for the resource (increasing the interference generated by that higher priority task). This contrasts with the previous approach of considering that a remote processor can issue an infinite number of access requests thus pessimistically assuming a full FIFO queue on each access request. The efficiency of the analysis is further increased by considering the specific access time for each resource access request for both the analysed and remote tasks requests, as introduced in the first contribution. This improved analysis is then extended to support the analysis of nested resource accesses. This includes the means to calculate a safe upper bound on the overhead of tasks migrations as part of the helping mechanism.

Finally, two specific approaches to semi-partitioned systems using MrsP are outlined. Although in the previous contributions a fully partitioned system is considered, tasks under MrsP are allowed to migrate during resource accesses. As a consequence any MrsP implementation has to support tasks migrations. In that case, semi-partitioned systems can be implemented, alleviating the allocation problem by splitting the computation time associated to a task among a number of processors. A set of rules for the safe splitting of tasks under MrsP is given, followed by two specific possible implementations.

The next chapter is devoted to the evaluation of the protocol by applying it to a real space mission. The system will be first described and then studied for a classic monocore implementation, and then for a triplecore platform, controlling the resource sharing with MSRP and MrsP, respectively. How the proposed extensions are used is detailed during the analysis, and then the results are compared and conclusions drawn.

Chapter 6

Evaluation

In this section an evaluation of the MrsP protocol, with the improvements proposed in this thesis, is presented. This evaluation consists in the timing analysis study of an academic satellite, UPMSat-2. While the already published MrsP work has mainly addressed the protocol evaluation by comparing its performance against relevant protocols using synthetic workloads [36, 69, 138, 116], in this document the scheduling analysis of a real system will be addressed.

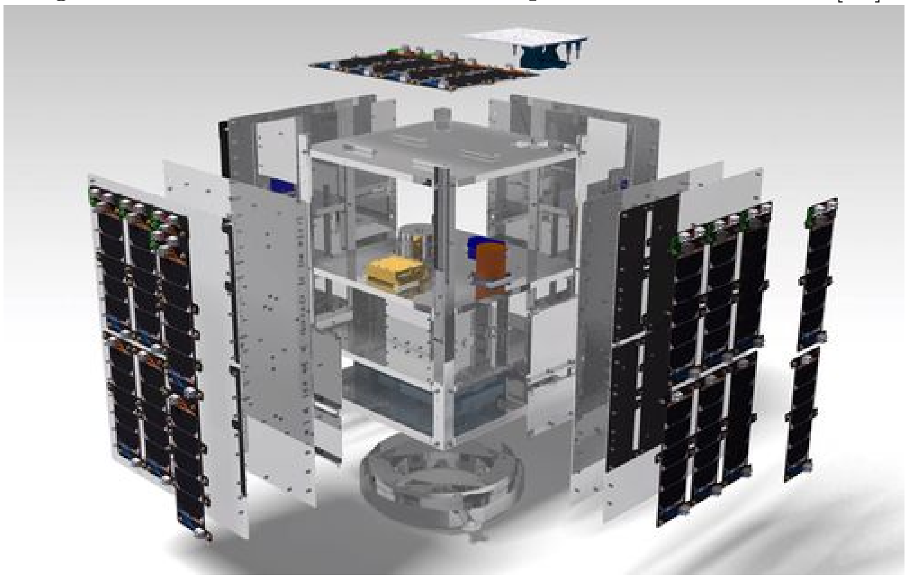
6.1 Case Study: UPMSat-2

6.1.1 Overview

UPMSat-2 [52] is a micro-satellite mission which is being developed by several research groups within UPM together with some industrial partners. Its main objective is to serve as a technology demonstrator, and it is being widely used to evaluate research developments [122]. It will serve, as well, to achieve the highest qualification of different industrial products being carried on-board as part of the satellite main subsystems or experiments.

The satellite is characterized by its 50 kg of mass, with an envelope measuring $0.5 \times 0.5 \times 0.6$ m (figure 6.1). It is envisaged to follow a low Earth noon sun-synchronous polar orbit [59], at an altitude of 700 km and 98 min period. This orbit presents an expected eclipse time of 36 min. With this orbit, the satellite will have two periods of visibility per day of 10 min each from its main ground station at UPM facilities in Madrid.

Figure 6.1: Overview of the satellite platform. Taken from [77]



During its two years of expected life span it will conduct different active and passive experiments. These experiments encompass the qualification of a set of devices from industrial partners, the validation of different techniques, methods and algorithms, as well as the observation of relevant parameters of the low Earth orbit environment and their effect on state of the art materials and parts for space use.

The data gathered during these experiments, along with housekeeping telemetry and telecommand messages will be exchanged during the mentioned periods of visibility using a dual radio link in the VHF 400 MHz and 436 MHz bands with transfer rates ranging from 1200 bauds to 9600 bauds depending on the frequency band and weather conditions. The packet format is based on the CCSDS/PUS format adopted by the European Space Agency and defined in the ECSS-E-70-41C [58] standard over an AX.25 data link layer[22].

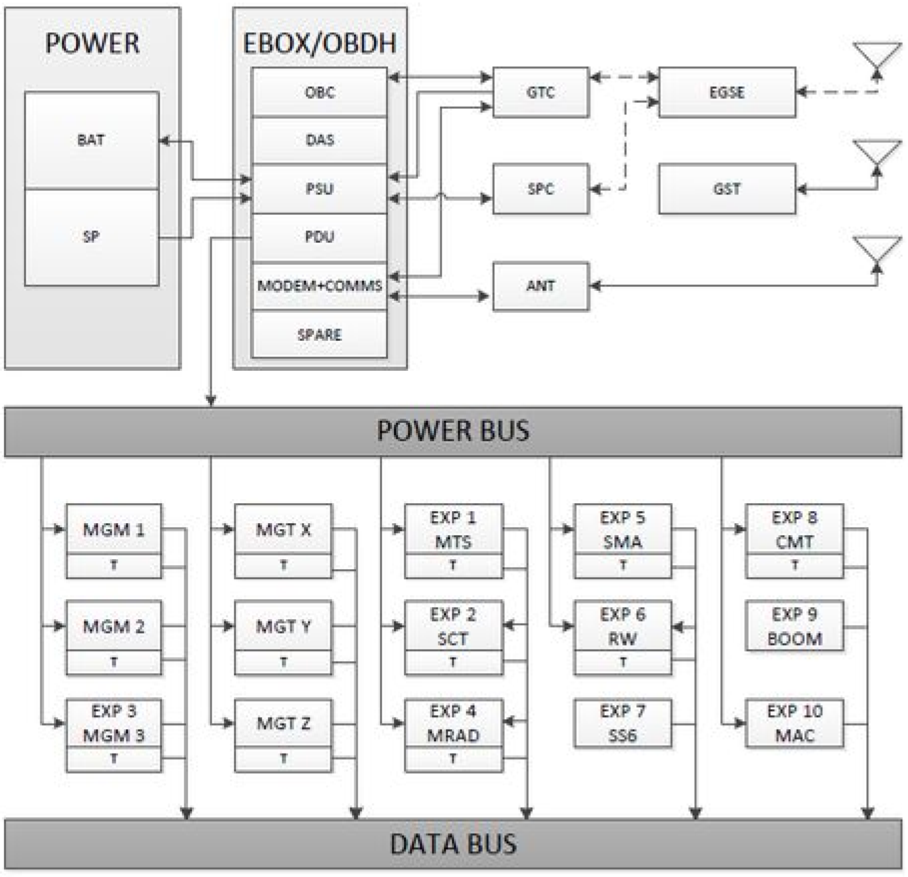
6.1.2 Monocore Design

Hardware design

The UPMSat-2 platform design is depicted in figure 6.2. Power is provided to the platform by a Saft 6S4P battery made up of four sets of six VES16 Li-Ion cells connected in parallel. This battery has a capacity of 18 Ah and provides a voltage between 17.4 and 24.3 V depending on the charging state. The battery life span is expected to be of up to 12 years, far longer than the mission expected life span (time required to complete all the experiments carried on board). Batteries are charged by means of solar panels, placed around the four lateral sides of the satellite, as well as half of the upper side. These solar panels are made up of arrays of SPVS X5 modules from Selex Galileo, comprised of 3G-28% solar cells from Azur Space.

The Electronic BOX (EBOX) or On Board Data Handling (OBDH) system is the main control system of the satellite. It is comprised of six blade slots interconnected by means of a back panel. The master one is the On Board Computer (OBC) which controls the other blades. On it, a LEON3 processor is implemented on a radiation-hardened FPGA with multiple interfaces to interact with other boards and devices: 3 UARTs, SPI and I2C ports as well as 112 digital I/O ports. It also includes interval and watchdog timers, 4 MB of RAM and 2 MB of EEPROM memory, all connected to the Central Processing Unit (CPU) via an AMBA bus. The LEON3 processor clock runs at 20 MHz and the processor chip includes 16 KB of instruction and data cache respectively.

Figure 6.2: Overview of the satellite hardware systems. Taken from [77]



The LEON family of processors are based on the SPARC-V8 architecture with a RISC instruction set. With a 32-bit CPU the processor was first developed by the European Space Agency (ESA) and then continued by Cobham Gaisler from LEON3 model onwards. It is distributed as a synthesizable VHDL model to be used in System-on-Chip (SoC) developments.

LEON3 processors have a seven-stage pipeline and symmetric multiprocessing (SMP) support that allows multiprocessor designs, as will be discussed in section 6.1.3. As mentioned before, the OBC board has a set of interfaces for which a number of GRLIB cores have been also synthesized: 32-bit PROM and SRAM controllers, SPI and I2C controllers, three UART cores (one used for programming and debugging purposes), a modular timer unit and an interrupt controller. Finally a set of four general purpose I/O port controllers have also been synthesized handling up to 32 signals each.

The OBC board commands the other boards in the EBOX by means of a back panel which is notionally the EBOX data bus. The OBC can only interact with one of the boards at a time (except for the Communications board). As a result and as will be detailed later, access to the boards needs to be carefully synchronized.

The DAS (standing for Data Acquisition Subsystem) board is mainly comprised of an analog to digital converters (ADC) and the required wiring and multiplexing. As the converter only has 8 different input channels, a 3-bit multiplexor has been implemented to support up to 64 analog signals. The sets of signals and their measured magnitude are detailed in table 6.1. Each kind of signal is measured by a specific device converting the studied magnitude to a range between 0 V and 5 V and the ADC outputs a 12-bits representation of this voltage via SPI. Then the OBC converts back the measured voltage to each engineering value applying a specific transfer function for each signal.

The PSU (standing for Power Supply Unit) board supplies power

Table 6.1: Analog signal groups.

Magnitude	Signal group
Temperature	Battery temperatures
	PSU Temperatures
	Solar Panel Temperatures
	OBC Temperatures
	Magnetometer temperatures
	Magnetorquer temperatures
	EBOX temperatures
	RW temperatures
	MTS temperatures
Voltage	Battery voltages
	PSU voltages
Current	PSU currents
	PDU currents
	Solar Panel currents
Magnetic Field	Magnetometer output
Solar radiation	Solar sensor outputs

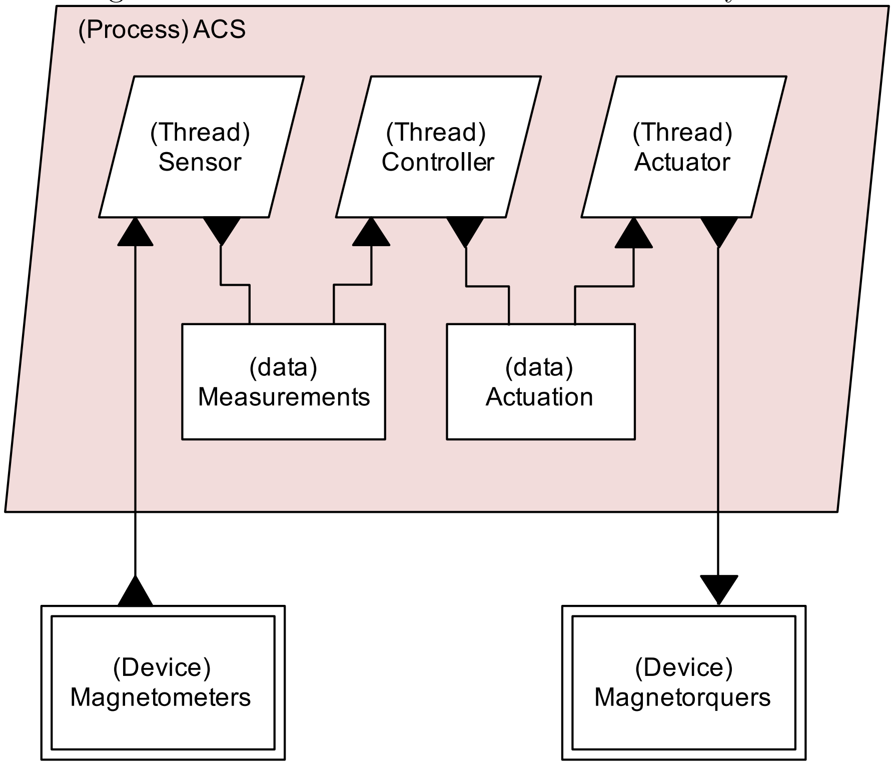
to the OBC and other boards. Directly connected to the batteries also enables monitoring the battery charge by means of 3 digital signals each of them standing for High, Low or Critical charge level. There is an implicit Nominal charge level when no other level is signalised. The board has a watchdog timer implemented by a capacitor, which can reset the whole platform in case of a software or hardware error. To avoid this, a digital signal is to be kept low by the OBC for at least 5 ms each 200 ms. Otherwise, a 24 s capacitor starts draining. When the capacitor is completely drained, the satellite enters a hibernation mode, in which all systems are powered off to allow the batteries to be recharged. Finally, it also has a mission clock, which is started at the separation from the launcher and is kept powered even during hibernations. This clock is directly connected to the OBC via a SPI bus (so it does not require the PSU board to be selected for operation on the back panel).

The PDU (standing for Power Distribution Unit) board is in charge of controlling which devices are powered during the mission. Some supplies are to be activated at start time, such as the radio board, the Attitude Control System (ACS) devices or the temperature sensors. Some others are used just eventually during the experiments and powered specifically at that time. Finally, the ACS actuators are manipulated twice every 2 s, once to set the actuation and once to deactivate it (the actuation consists in powering the actuators with positive or negative supply).

Finally, the communications board or HWComm has its own direct link to the OBC via a UART serial line at 38400 bauds. Via this link it can be configured or commanded to send telemetry messages. The board also sends via this serial line the telecommand messages when they are received. The radio equipment is half duplex, so a specific protocol has been defined to synchronize the transmission windows with the mission ground station (GS).

The satellite orientation with respect to the Earth is controlled by a specific subsystem, the Attitude Control System (ACS). This system has two kinds of sensors: a set of two magnetometers from SSBV Aerospace (MGM) that determine the magnetic field around the satellite on each of the three axis, and a set of solar cells from IES/UPM, that quantify the amount of solar radiation received on each axis. The former are used for the nominal control as well as for different experiments, while the latter are only used for experimental procedures. For the actuation, the ACS system also has two different devices: the nominal attitude is controlled via a set of three magnetorquers (MGT) from ZARM Technik AG that generate a magnetic field that, interacting with the Earth magnetic field produce a torque; and a reaction wheel (RW) from SSBV, which, by changing its rotational speed generates a counter reaction, rotating the satellite proportionally. As well as the solar cells, the reaction wheel will only be used to conduct ACS experiments.

Figure 6.3: Overview of the Attitude Control System.



Software design

The UPMSat-2 on-board software has been designed aiming to replicate the defined subsystems from the mission perspective into software packages encapsulating their related functionality. In this regard, the requirements identified in the software system specification requirements baseline document [51] have been assigned to software subsystems in the technical specification, according to the kind of mission goal fulfilled. As depicted in figure 6.4, eight higher level software packages have been identified.

Apart from the *BasicTypes* module, which only contains type specifications, all software subsystems follow the structure depicted in figure 6.5.

Figure 6.4: Overview of the onboard software design.

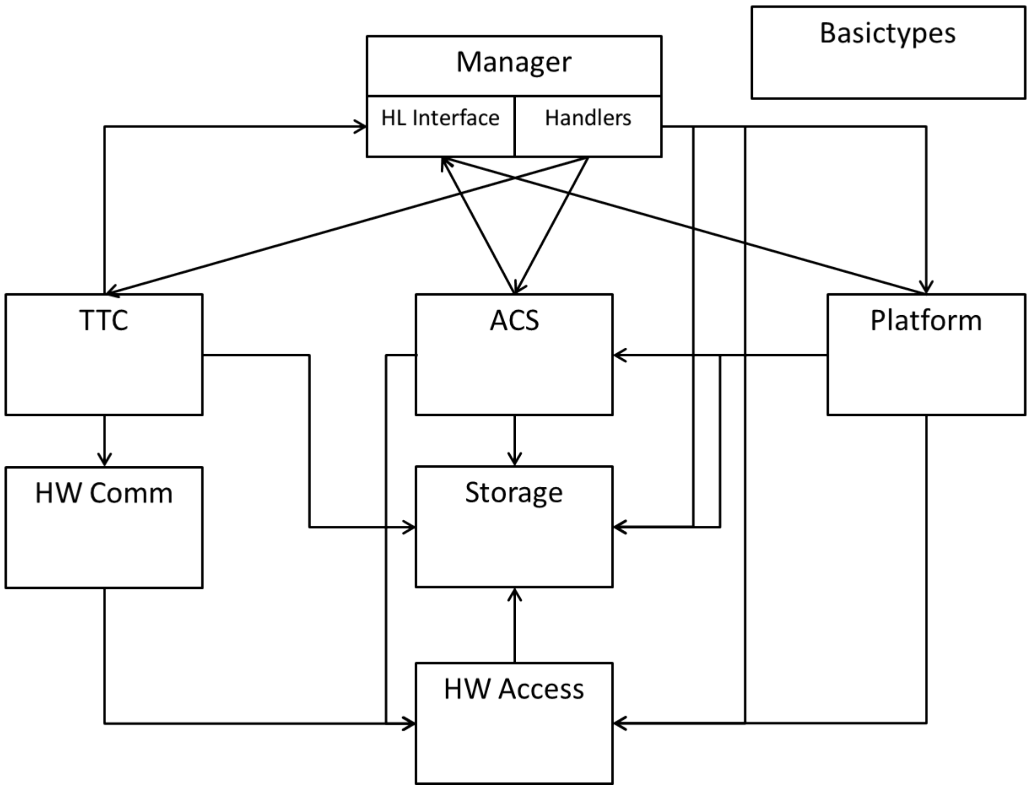


Figure 6.5: Generic subsystem software design.

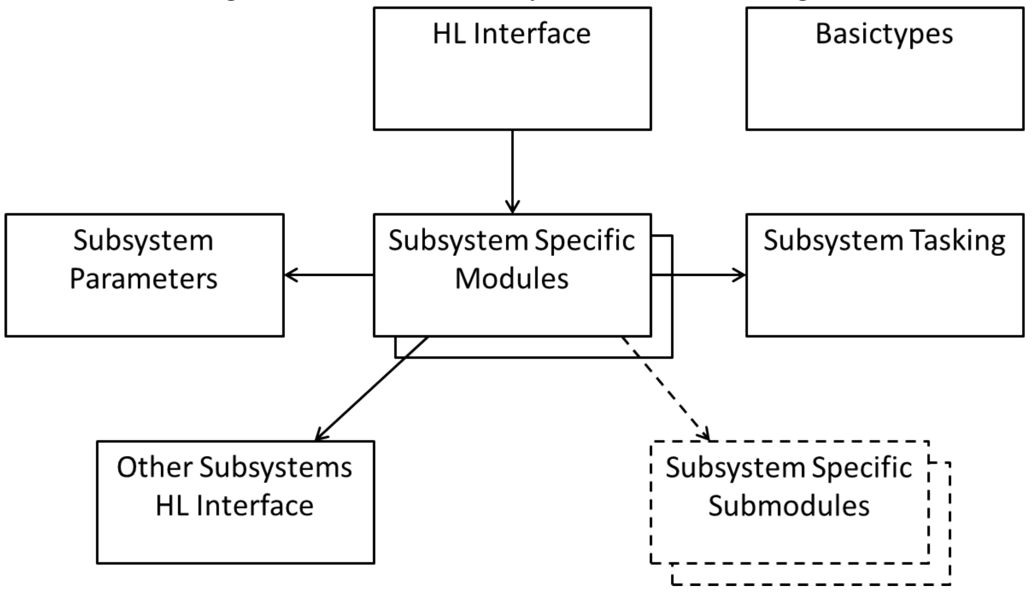
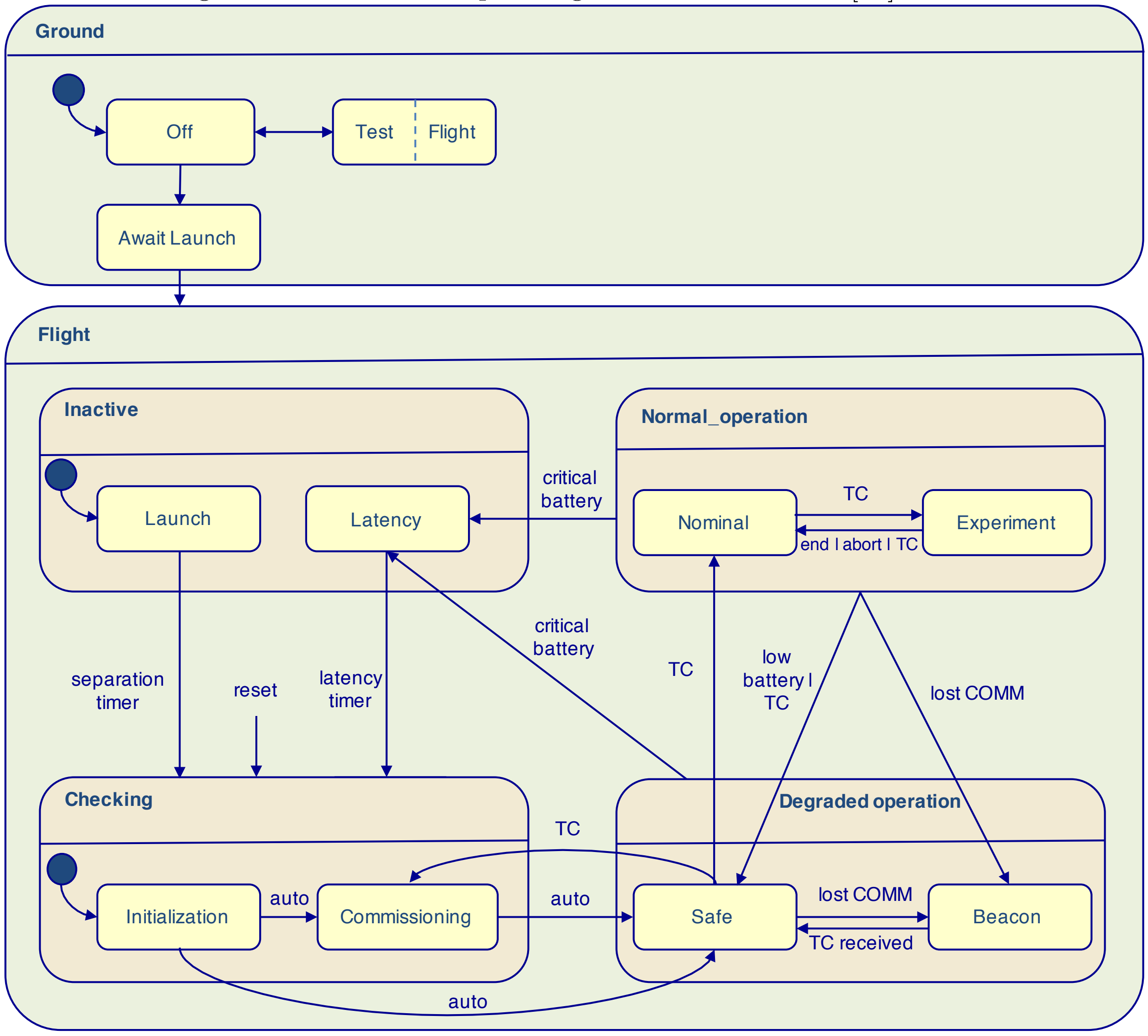
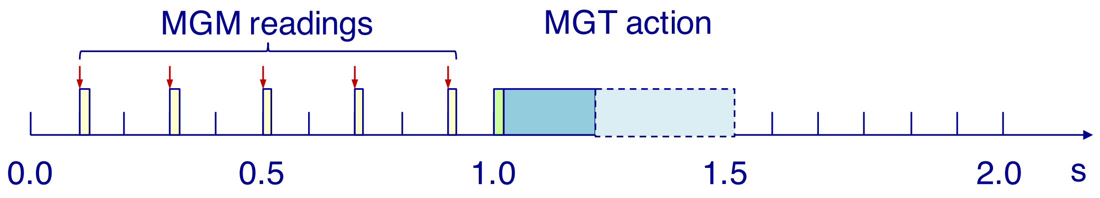

HL Interface
Basictypes
Subsystem Parameters
Subsystem Specific Modules
Subsystem Tasking
Other Subsystems HL Interface
Subsystem Specific Submodules

Figure 6.6: UPMSat-2 operating modes. Taken from [51]



The Manager subsystem is in charge of controlling the satellite general behaviour, by means of the operating mode managing. The satellite has eight operating modes defined for flight, as shown in figure 6.6. While the on-board computer is inactive in the Launch and Latency modes, each software subsystem has a defined behaviour for each of the remaining modes. The manager subsystem is in charge of maintaining the appropriate operating mode according to the mission definition based on the events detected on-board, as well as on the telecommands received from the mission control on ground. It is also in charge of the proper synchronization of the initialization and commissioning procedures, as well as logging any relevant event detected on-board, to be later sent to ground.

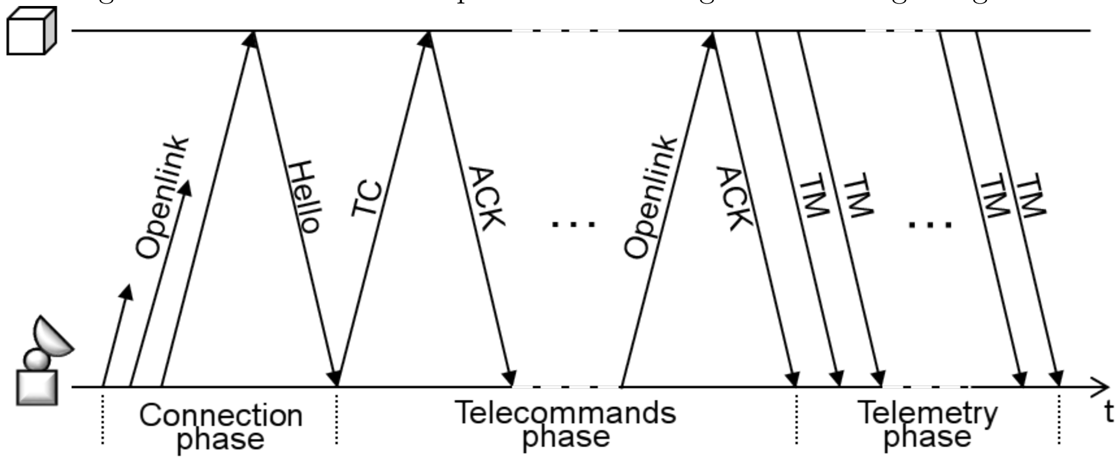
Figure 6.7: Attitude Control System cycle. Taken from [135]



The Attitude Control System (ACS) is in charge of maintaining the proper satellite attitude with respect to the Earth. The structure is that of a traditional control loop: sense, control, actuate. The control algorithm is based on an innovative modification of the B-dot strategy [48] implemented on a Simulink model, from which the source code of the ACS algorithms has been generated automatically [86, 67]. The implementation and timing analysis of this subsystem has been thoroughly reviewed and published, including its overall design [135], worst-case execution time measurement using dynamic [66] and static methods [70], as well as the schedulability analysis of the involved tasks [134]. These tasks must take 5 magnetic field measurements for each axis and magnetometer during the first second of the control loop (each measurement separated by 200 ms as shown in figure 6.7), calculate the control (green task in figure 6.7) and finally actuate on the attitude by means of the magnetorquers. As explained in [134], the magnetorquers are controlled using a Pulse-Width Modulation (PWM) mechanism. Thus, it is required that the task controlling the actuation is activated once following the control task completion to start the pulse on each magnetorquer and then up to three times to finish the actuation on each magnetorquer. As the control algorithm only produces pulse widths of $[0, 200, 300, 400, 500]$ ms, the longest activation offset can be of 500 ms.

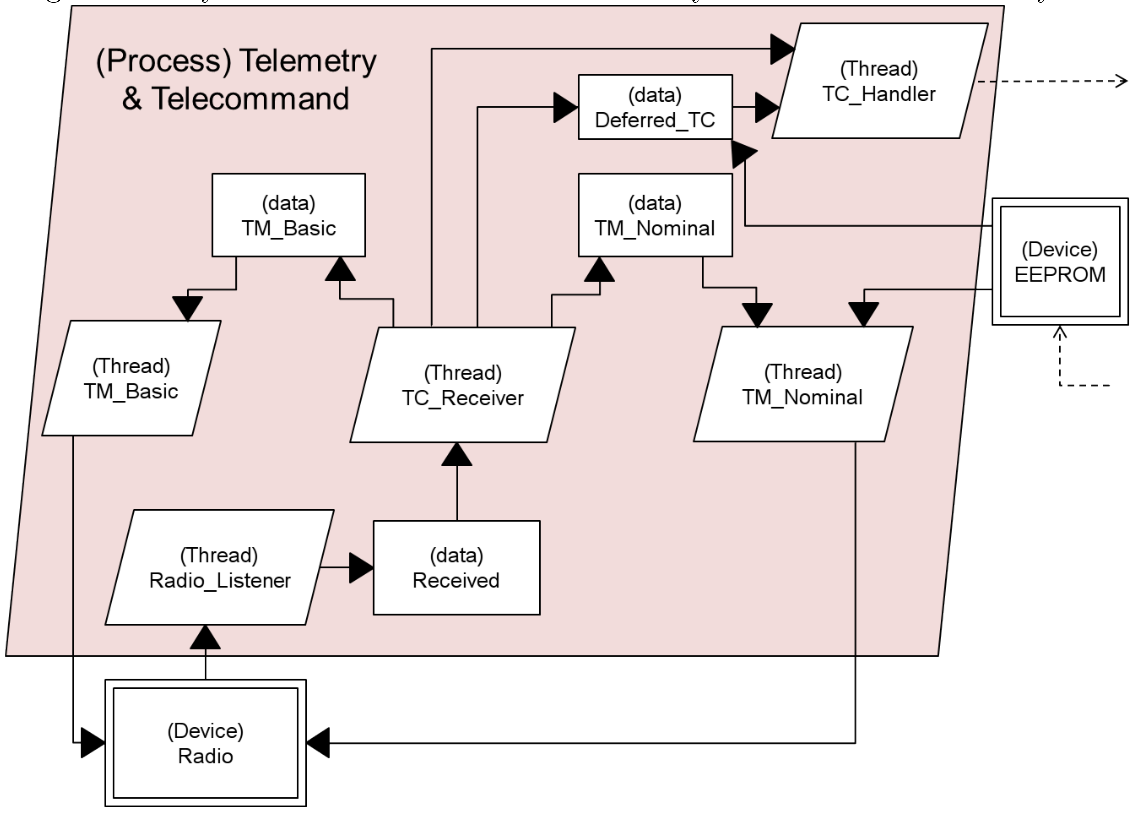
The Telemetry and Telecommand subsystem (TTC) is in charge of handling the communications between the ground and flight segments. As part of this duty the TTC controls the communication hardware behaviour by means of the HWComm package interface, as shown in figure 6.4. Following the protocol outlined in figure 6.8, the communication between both segments is initiated by the ground segment sending an initial Openlink message periodically close to the expected visibility start time on each pass. When the satellite first detects this message, it answers with a Hello message carrying the most up-to-date available data from the mission clock value, analog and digital signals, as well as the current operating mode and battery status. Then, the ground station transmits the programmed telecommands, signalling its end of transmission by another Openlink message. When the satellite receives this second Openlink message, it starts sending its stored telemetry until all has been sent or the expected visibility time is concluded. The received telecommands during the transmission window can be sent to be executed at reception time or at a later time. When the command is to be executed at a posterior time, the TTC is also responsible of holding the command until the envisaged execution time. The delayed telecommands are backed up using the Storage subsystem to prevent their lost upon an eventual reset of the OBC. Finally, the TTC subsystem is also in charge of sending, each 30 s a Hello message when not in coverage from the ground station. This message allows amateur radio operators to track the satellite and monitor its basic status as well as serving as a backup communication method in case of malfunction.

Figure 6.8: Communication protocol between ground and flight segments.



The HWComm subsystem is in charge of interfacing with the communications board from Emxys. Via a UART serial line, the radio board can receive commands as well as send the received telecommands to the OBC. Among the commands the radio can receive are: send a telemetry message, change frequency band, change the transmission baud rate, start, stop and configure a beacon message or read back the current configuration. When commanded, the radio equipment automatically sends back an ACK message with optional response values as for the configuration request. These ACK messages are sent back to the OBC when the requested operation has been performed. While a simple configuration message can be processed almost instantly by the radio, sending a long telemetry message at the lowest transmission speed (including the synchronization head and tail flags) can delay more than 1500 ms. In order to detect and minimize the effects of a malfunction of the radio board (in particular not receiving any message back), as well as to avoid wasting processor time by polling the response, a self-suspension delay is performed

Figure 6.9: Dynamic architecture of the Telemetry and Telecommand subsystem.



after sending a message. This delay depends on the kind of command sent (configuration or telemetry) as well as on the transmission speed set (1200 bauds or 9600 bauds). The HWComm also implements the interpretation of the information received from the radio equipment via the UART line. It can be of three types: an ACK message as mentioned before (optionally carrying extra information), a NACK message or a telecommand message. This interpretation can be done thanks to the delimiters placed by the radio board as well as on the data carried on each kind of messages.

The Platform subsystem is in charge of controlling the proper status and behaviour of the satellite hardware (or platform). It is in charge of periodically polling the analog and digital satellite signals to both log them to be sent to ground as well as to check if the values are in the expected range or status. If not, an event message is produced and sent to the manager, which is in control of the state transition logic (e.g. go to `Safe` mode when the low battery signal is detected). Each type of signal has its own polling period, but care has been taken in setting them so as to yield a harmonic hyper-period, easing the timing analysis. These periods can be changed by TC from ground. To do so, a schedulability analysis must be carried out first to ensure that the changes will not harm the system feasibility. It also provides an interface for the Mission_Clock to the rest of the subsystems.

The HWAccess subsystem, as its name suggests, offers a simple interface to the platform hardware to the other subsystems. In particular, it encapsulates all accesses to the PDU, PSU and DAS boards described in section 6.1.2. As mentioned in there, the access to these boards is done by means of a set of digital signals that are reused for each board. Accordingly, accesses to the auxiliary boards have to be coordinated and made in mutual exclusion. To do so, apart from the access to the mission clock, all access requests to the boards are done via a protected object, as will be mentioned is section 6.1.2 where the dynamic software architecture will be described. The HWAccess module is also responsible for resetting the watchdog timers (WDT) implemented on the PSU and OBC boards respectively.

Finally, the Storage subsystem provides a non-volatile storage service to the system. On it, apart from the resident software, there are stored the software configuration parameters of each subsystem, the telemetry information to be sent to ground, the telecommands pending to be executed as well as the experiments data. While the telemetry, telecommands and experiments information are stored using classic circular buffers, the configuration parameters are stored following a transactional model, in order to prevent data corruption. In particular, two blocks containing the configuration data are kept, one up to date and another ready to be updated by request. When

Table 6.2: Non-volatile memory configuration.

Data Type	Block Size	Number of blocks
Configuration Parameters	512 B	8
Pending TCs	512 B	8
Events & Errors TM	128 B	485
Experiments TM	128 B	1,500
Housekeeping TM	128 B	2,047

an update is requested, the latter block is written. Upon a successful write operation, a master block is written updating the up to date block identification. As only one change is allowed at a time on that block, an error writing the block (such as an OBC reset) would just not update the updated block ID and keep the previous one, and a new update request would be required. As mentioned before, the non-volatile memory chip is a 2 MB EEPROM device. This device supports byte addressing, as well as block-oriented write operations. As write operations require a wait time of 15 ms after a successful write, writing by blocks highly improves the bandwidth and overall efficiency of the system. Blocks can be defined of up to 1024 B by the user (in fact, blocks are just an abstraction since they are not implemented in any way in the hardware, i.e. a block is nothing more than a starting address and length definition). Table 6.2 presents the block size and number of blocks reserved for each kind of information.

Real-time architecture

The satellite and subsystem behaviour described above has been implemented in Ada using the restricted Ravenscar profile described in section 2.2.7. In it, a set of periodic and sporadic tasks concurrently execute to perform the defined operations with a bounded response time. Task dispatching is decided among available tasks using fixed priorities, and coordination among tasks is obtained using shared resources. These shared resources have a set of constraints to ensure timing predictability as well as to avoid concurrence issues such as deadlocks or starvation. In particular, and relevant to the UPMSat-2 software, shared resources under the Ravenscar profile can only be guarded by one entry barrier, and only one task can be blocked at a time on a given barrier, as mentioned in section 2.2.7.

The translation into tasks of the behaviour defined in section 6.1.2 is shown in table 6.3, where task names, release patterns (periodic or sporadic), priorities and the related subsystems are listed.

Due to different constraints, as well as to better mimic the encapsulating engineering system, the tasks listed in table 6.3, sometimes need to self-suspend during an activation. In order to perform an appropriate timing analysis following the Ravenscar profile, an equivalent task set has to be defined, as shown in [134], where the ACS subsystem was analysed. In this equivalent task set other three entities have to be taken into account. The TTC subsystem includes three Ada timing events, which are triggered at a specific processor clock value. Ada timing events are a way to implement this behaviour without the need of a full-fledged task. However, they have a downside, which is that they are required to be executed at the highest priority in the system. The events handled this way are:

- The expiration of the visibility time expected for each coverage period. After this time the satellite must stop sending TM messages.
- An excessive time has elapsed since the last contact with the ground station. This time has been set to be of two days. The satellite must change to *Beacon* mode to ease its detection on ground.
- A deferred TC is to be executed. TCs will be sent to be executed with a minimum separation of 2500 ms. Since the required release time is represented as a mission clock values, i.e. as a number of

250 ms of resolution, the minimum inter-arrival time of two consecutive deferred TC events has been set to 2000 ms (intentionally the same as the TCs sent to be executed immediately).

The resulting equivalent task set is listed in table 6.4.

As mentioned before, the Ravenscar model allows data-oriented synchronization among tasks by means of shared resources. As identified in sections 2.2.6 and 4, shared resources have a heavy influence on the scheduling analysis. Under the Ravenscar profile, shared resources cause two main effects to tasks: they can increase the active priority of a task, reducing the set of tasks that can preempt them (and potentially causing arrival blocking to those tasks), and can block tasks on entry barriers until the barrier condition is satisfied. Accordingly, the study of the shared resources behaviour is as necessary as that of the system tasks to perform a proper schedulability analysis. Table 6.5 identifies the UPMSat-2 software shared resources.

Table 6.3: UPMSat-2 task set.

Subsystem	Task	Pattern	Priority	Period / MIT	Deadline
OBSW	OBSW	S	First	-	
Manager	CM_Handler	S	Last-10	-	
	TC_Handler	S	Last-5	2000 ms	
	EV_Handler	S	Last-6	2000 ms	
Platform	Housekeeping_Task	P	Last-7	1000 ms	500 ms
Hwaccess	Wdt_Fpga	P	First+1	5000 ms	2500 ms
	Wdt_Psu	P	First+2	250 ms	125 ms
TTC	TM_Nominal_Task	S	Last-9	1800 ms	15000 ms
	TM_Basic_Task	P	Last-8	30000 ms	
	TC_Receiver_Task	S	Last-4	2000 ms	
HWComm	Radio_Listener	S	Last-3	2000 ms	
ADC	Sensor_Task	P	Last	2000 ms	1000 ms
	Control_Task	P	Last-1	2000 ms	1000 ms
	Actuator_task	P	Last-2	2000 ms	1000 ms

Table 6.4: UPMSat-2 equivalent task set.

Task	Job	Priority	Period	Offset	WCET	Jitter
Timing_Events	Visibility_Timer	14	5.82E+6 ms		0.4000 ms	0
Timing_Events	Lost_Comm_Timer	14	1.73E+8 ms		0.2260 ms	0
Timing_Events	Deferred_TC_Timer	14	2000 ms		17.4820 ms	0
Sensor_Task	1 to 4	13	2000 ms	200 ms	74.0930 ms	0
Sensor_Task	5	13	2000 ms	200 ms	74.2400 ms	0
Control_Task		12	2000 ms	1000 ms	4.5410 ms	0
Actuator_Task	PWM_ON	11	2000 ms	1000 ms	1.8960 ms	*R*_Control_Task
Actuator_Task	PWM_OFF_1	11	2000 ms	300 ms	1.8030 ms	*R*_PWM_ON
Actuator_Task	PWM_OFF_2	11	2000 ms	300 ms	1.8030 ms	*R*_PWM_OFF_1
Actuator_Task	PWM_OFF_3	11	2000 ms	300 ms	1.5690 ms	*R*_PWM_OFF_2
Radio_Listener		10	2000 ms		2.6300 ms	0
TC_Receiver_Task		9	2000 ms - *R*_Radio_Listener		38.0490 ms	0
TC_Handler		8	2000 ms - *R*_TC_Receiver_Task		36.5600 ms	0
EV_Handler		7	2000 ms		66.5000 ms	0
Housekeeping_Task		6	1000 ms		324.7000 ms	0
TM_Nominal_Task	Trans_Config	4	5.82E+6 ms		1.5760 ms	0
TM_Nominal_Task	Send_Post_Config	4	-	300 ms	64.9770 ms	*R*_Trans_Config
TM_Nominal_Task	Send_Post_Message	4	1800 ms		64.7810 ms	*R*_Send_Post_Config
TM_Nominal_Task	Finish	4	-		0.2070 ms	*R*_Send_Post_Message
Wdt_Psu		2	250 ms		5.5000 ms	0
Wdt_Fpga		1	5000 ms		0.0180 ms	0

Table 6.5: UPMSat-2 shared resources.

Subsystem	Protected Object	Priority
Manager	Commissioning	8
	Event_Buffer	14
	Command_Buffer	14
	Mode	8
Platform	Configuration (Platform)	8
	Table	8
	Ebox_Access	13
	I2C_Access	14
	Buffer	14
ACS	MGM_Data	13
	MGM_Signals	13
	MGT	12
	Parameters	13
TTC	Visibility_Timer	14
	Lost_Communications_Timer	14
	Deferred_Timers	14
	Configuration (TTC)	14
	Sync	10
	TM_Basic	9
	TM_Nominal	9
	Receive_Pool	10
Storage	Configuration_Parameters	14
	Storage_Buffer	14
	Backwards_Storage_Buffer	8
	EEPROM	14
	EEPROM_Update	14

6.1.3 Multicore Design

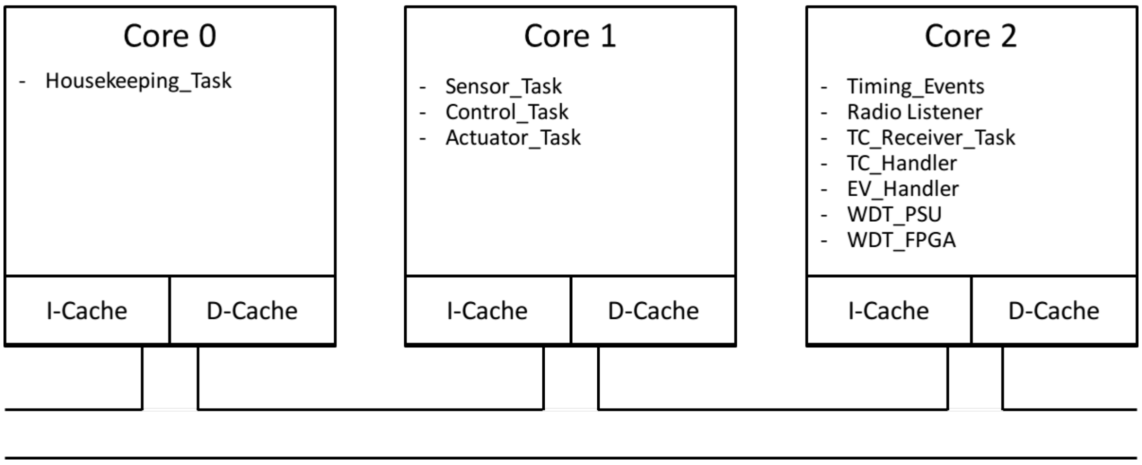
During an early development phase, a prototype dual-core platform was built for research purposes. In particular, this platform was used to exercise different techniques and procedures for the development of strict real-time systems over multi-processor platforms, based on virtualization[2].

This platform was later expanded to a three core system, based on a Virtex-5 FPGA. This board has been synthesized to mimic as close as possible the UPMSat-2 real hardware performance (apart from the number computing units).

While in previous work the dual-core design was suggested isolating the TTC subsystem on one core and assigning the rest of the software pieces to the other core, this approach later revealed to present relevant drawbacks. The most notable one is that, while the system design was aimed to provide better response times to the TTC related jobs and thus improve the amount of data that could be transmitted to ground, the TTC assigned core would be almost idle during the out-of-coverage periods, which span more than the 98% of the time. Other drawbacks include the relatively low CPU utilization by the TTC even during coverage periods, as the transmission and propagation times are orders of magnitude higher than the TTC computation times. Finally, other subsystems proved to be more sensitive to the response time behaviour and precision, but drawing no evident benefit from the new multiprocessor platform. A paradigmatic example is the ACS, which strictly depends on the timely data acquisition and actuation but whose response times were only marginally improved by the extra computation unit.

As a result, for the three core approach, a new task allocation has been proposed. This new allocation reserves one core for the two most CPU time demanding subsystems, the ACS and the Platform subsystems. The remaining Manager, TTC, HWComm, and Storage subsystems are allocated to the third CPU, as shown in figure 6.10.

Figure 6.10: UPMSat-2 3-core task allocation scheme.



This new allocation reduces both the processor contention for the isolated subsystems and the interference suffered by the other tasks. This, however comes with a cost as has been already discussed: now accesses to shared resources have to be specifically coordinated among cores. In particular, the access to the shared hardware devices becomes a relevant issue. While the HWAccess module is required (during nominal operation) by the ACS and Platform Subsystems, the Storage module is required by almost all other subsystems, allocated among the three cores. The effect of this sharing will be analysed in detail for both MrsP and MSRP approaches in sections 6.3 and 6.4, respectively.

6.1.4 Timing Data Acquisition

The timing data used for this evaluation has been acquired using different tools of the Rapita Verification Suite (RVS)[1] toolset.

The Rapita Verification Suite is a dynamic execution time analysis

[1]http://www.rapitasystems.com/products/rvs

set of tools. The analysis is based on the execution trace acquisition by executing the code to be analysed on the real platform and its post-processing offline to compute worst-case execution values. Traces and the analysis can be generated thanks to a previous static analysis of the code structure, which generates a system model, that is used to add instrumentation points at relevant points of the code. The mentioned post-processing consists in analysing the trace (succession of time-stamped instrumentation points) generated at execution time. The result of the analysis is a WCET profile, with probabilistic data, for each identified block during the static analysis.

In addition to performing measurement-based execution-time analysis, the RVS also includes other tools that have been used during the evaluation. In particular the coverage tool reports provide information on which blocks have been tested and which not, and for those that have been tested, it reports the number of executions and statistics of those executions.

Finally, the rapitask feature has been of great help to fully understand the task interaction and concurrence on such a relatively complex system.

6.2 Analysis of a Mono Core Implementation

Given the design and real-time architecture defined in section 6.1.2, the monoprocessor schedulability analysis can be done using the response time analysis techniques introduced in section 2.2.6. Since this is a mature technique, and the reasoning on the system design leading to these results has been already addressed, only relevant details will be presented.

Out-of-coverage phase analysis

As mentioned earlier, the satellite is most of the time out of coverage from a ground station. During this phase, the satellite communications are reduced to sending a basic message every 30 s. However, it cannot be excluded that the satellite might receive data from the TTC. This data can be received as a result of a radio hardware malfunction, or a malicious message reception, among others. In the worst case, this data might be considered a TC by the first filter at the Radio_Listener and then discarded (as not being authenticated) by the TC_Receiver_Task. The minimum inter-arrival time for such event will be considered to be of 2 s since, among other considerations, this value is close to the transmission time of the smallest amount of data that could be considered to be TC by the TTC board and Radio_Listener filters. The response time analysis for this phase is presented in table 6.6.

Telecommands phase analysis

The telecommands phase begins when the satellite receives an Openlink message from ground. During this phase, no telemetry is sent, resulting on a lower blocking for high-priority tasks, as shown in table 6.7.

Telemetry phase analysis

After the successful transmission of the programmed telecommands, the ground station sends a second Openlink message to signal the satellite that it can start transmitting the stored TM. The response time analysis is shown in table 6.8.

Table 6.6: UPMSat-2 response time analysis during out-of-coverage phase.

Task	Job	Priority	Period	Offset	WCET	Jitter	B	R
Timing_Events	Visibility_Timer	14	5.82E+6 ms		0.400 ms		71.016 ms	71.416 ms
Timing_Events	Lost_Comm_Timer	14	1.73E+8 ms		0.226 ms		71.190 ms	71.416 ms
Timing_Events	Deferred_TC_Timer	14	2000 ms		17.482 ms		53.934 ms	71.416 ms
Sensor_Task	1 to 4	13	2000 ms	200 ms	74.093 ms		37.500 ms	129.701 ms
Sensor_Task	5	13	2000 ms	200 ms	74.240 ms		37.500 ms	129.848 ms
Control_Task		12	2000 ms	1000 ms	4.541 ms		37.500 ms	60.149 ms
Actuator_Task	PWM_ON	11	2000 ms	1000 ms	1.896 ms	38.234 ms	37.500 ms	117.653 ms
Actuator_Task	PWM_OFF_1	11	2000 ms	300 ms	1.803 ms	373.823 ms	37.500 ms	475.064 ms
Actuator_Task	PWM_OFF_2	11	2000 ms	300 ms	1.803 ms	409.319 ms	37.500 ms	532.475 ms
Actuator_Task	PWM_OFF_3	11	2000 ms	300 ms	1.569 ms	444.815 ms	37.500 ms	589.652 ms
Radio_Listener		10	2000 ms		2.630 ms		37.500 ms	206.571 ms
TC_Receiver_Task		9	1793 ms		38.049 ms		37.500 ms	241.990 ms
TC_Handler		8	1758 ms		36.560 ms		37.500 ms	240.501 ms
EV_Handler		7	2000 ms		66.500 ms		37.500 ms	421.773 ms
Housekeeping_Task		6	1000 ms		324.700 ms		37.500 ms	967.649 ms
TM_Basic_Task	Trans_Config	5	30000 ms		1.714 ms		5.300 ms	864.173 ms
TM_Basic_Task	Send_Post_Config	5	30000 ms	300 ms	38.213 ms	836.173 ms	5.300 ms	973.662 ms
TM_Basic_Task	Post_Send	5	30000 ms	1800 ms	0.306 ms	945.662 ms	5.300 ms	862.765 ms
Wdt_Psu		2	250 ms		5.500 ms		0 ms	975.576 ms
Wdt_Fpga		1	5000 ms		0.018 ms		0 ms	992.094 ms

Table 6.7: UPMSat-2 response time analysis during TC phase.

Task	Job	Priority	Period	Offset	WCET	Jitter	B	R
Timing_Events	Visibility_Timer	14	5.82E+6 ms		0.400 ms		49.101 ms	49.501 ms
Timing_Events	Lost_Comm_Timer	14	1.73E+8 ms		0.226 ms		49.275 ms	49.501 ms
Timing_Events	Deferred_TC_Timer	14	2000 ms		17.482 ms		32.019 ms	49.501 ms
Sensor_Task	1 to 4	13	2000 ms	200 ms	74.093 ms		16.283 ms	108.484 ms
Sensor_Task	5	13	2000 ms	200 ms	74.240 ms		16.283 ms	108.631 ms
Control_Task		12	2000 ms	1000 ms	4.541 ms		16.283 ms	38.932 ms
Actuator_Task	PWM_ON	11	2000 ms	1000 ms	1.896 ms	38.932 ms	16.283 ms	75.219 ms
Actuator_Task	PWM_OFF_1	11	2000 ms	300 ms	1.803 ms	75.219 ms	16.283 ms	411.413 ms
Actuator_Task	PWM_OFF_2	11	2000 ms	300 ms	1.803 ms	411.413 ms	16.283 ms	447.607 ms
Actuator_Task	PWM_OFF_3	11	2000 ms	300 ms	1.569 ms	447.607 ms	16.283 ms	483.567 ms
Radio_Listener		10	2000 ms		2.630 ms		16.283 ms	259.043 ms
TC_Receiver_Task		9	1740 ms		38.049 ms		16.283 ms	220.773 ms
TC_Handler		8	1779 ms		36.560 ms		31.498 ms	238.122 ms
EV_Handler		7	2000 ms		66.500 ms		31.498 ms	419.394 ms
Housekeeping_Task		6	1000 ms		324.700 ms		5.3 ms	862.459 ms
Wdt_Psu		2	250 ms		5.500 ms		0 ms	862.659 ms
Wdt_Fpga		1	5000 ms		0.018 ms		0 ms	879.177 ms

Table 6.8: UPMSat-2 response time analysis during TM phase.

Task	Job	Priority	Period	Offset	WCET	Jitter	B	R
Timing_Events	Visibility_Timer	14	5.82E+6 ms		0.400 ms		71.016 ms	71.416 ms
Timing_Events	Lost_Comm_Timer	14	1.73E+8 ms		0.226 ms		71.190 ms	71.416 ms
Timing_Events	Deferred_TC_Timer	14	2000 ms		17.482 ms		53.934 ms	71.416 ms
Sensor_Task	1 to 4	13	2000 ms	200 ms	74.093 ms		37.500 ms	129.701 ms
Sensor_Task	5	13	2000 ms	200 ms	74.240 ms		37.500 ms	129.848 ms
Control_Task		12	2000 ms	1000 ms	4.541 ms		37.500 ms	60.149 ms
Actuator_Task	PWM_ON	11	2000 ms	1000 ms	1.896 ms	60.149 ms	37.500 ms	117.653 ms
Actuator_Task	PWM_OFF_1	11	2000 ms	300 ms	1.803 ms	117.653 ms	37.500 ms	475.064 ms
Actuator_Task	PWM_OFF_2	11	2000 ms	300 ms	1.803 ms	475.064 ms	37.500 ms	532.475 ms
Actuator_Task	PWM_OFF_3	11	2000 ms	300 ms	1.569 ms	532.475 ms	37.500 ms	589.652 ms
Radio_Listener		10	2000 ms		2.630 ms		37.500 ms	874.878 ms
TC_Receiver_Task		9	1125 ms		38.049 ms		37.500 ms	241.990 ms
TC_Handler		8	1758 ms		36.560 ms		37.500 ms	240.501 ms
EV_Handler		7	2000 ms		66.500 ms		37.500 ms	421.773 ms
Housekeeping_Task		6	1000 ms		324.700 ms		37.500 ms	967.649 ms
TM_Nominal_Task	Trans_Config	4	5.82E+6 ms		1.5760 ms		5.3 ms	864.035 ms
TM_Nominal_Task	Send_Post_Config	4	-	300 ms	64.9770 ms	836.035 ms	5.3 ms	1369.036 ms
TM_Nominal_Task	Send_Post_Message	4	1800 ms		64.7810 ms	1369.036 ms	5.3 ms	1368.840 ms
TM_Nominal_Task	Finish	4	-		0.2070 ms	1368.840 ms	5.3 ms	862.660 ms
Wdt_Psu		2	250 ms		5.500 ms		0 ms	1370.812 ms
Wdt_Fpga		1	5000 ms		0.018 ms		0 ms	1398.330 ms

Summary

The presented response time values show that, while the TTC active tasks change for each phase, their influence in higher priority tasks is low when compared with the execution time of those higher priority tasks. However, considering implicit deadlines for every task in the system, response time values of two tasks require specific attention.

One of them is the Housekeeping_Task. Response times of this task range between 862.459 ms and 967.649 ms, while its period if of 1000 ms. Although below the implicit deadline, these values are too close to it, specially when considering that the worst-case execution time of that task is of 324.700 ms, roughly a third of the mentioned response time values. This is mainly due to the high interference suffered from higher priority tasks.

The other task to be mentioned is the Wdt_Psu. In this case, if an implicit deadline is considered, its response times ranging between 862.659 ms and 1370.812 ms would make the task unschedulable. However, due to hardware implementation details that are out of the scope of this thesis this task can be considered as a soft real-time task, for which the presented analysis yields a tolerable number and frequency of deadline misses. It would be anyway desirable to reduce those response times below the task period.

6.3 Analysis of a Triple Core Implementation Using MSRP

The scheduling analysis of a triple core implementation of the UPMSat-2 implemented using MSRP will be presented in this section. The analysis will address the telecommands phase of the satellite, as it comprises all the relevant aspects to be addressed. The other two phases can be studied using the same approach and techniques that will be used here.

To perform such analysis, a preliminary response time analysis has been carried out. This analysis will serve for the first iteration calculations of the global contention for access to shared resources accesses, as will be explained below. This analysis is shown in table 6.9.

Table 6.9: MSRP response time analysis. Preliminary analysis. Time values en ms.

	Task	Subtasks	Priority	Period	Offset	Spin	WCET	C	Jitter	B	R
C_0	Housekeeping_Task		6	1000		0	324.700	324.700	0.000	0.000	324.700
Core_1	Sensor_Task	_1 a _4	13	2000	200 ms	0	74.093	74.093	0.000	0.000	74.093
	Sensor_Task	_5	13	2000	200	0	74.240	74.240	0.000	0.000	74.240
	Control_Task		12	2000	1000	0	4.541	4.541	0.000	0.000	4.541
	Actuator_Task	_PWM_ON	11	2000	1000	0	1.896	1.896	4.541	0.000	6.437
	Actuator_Task	_PWM_OFF_1	11	2000	300	0	1.803	1.803	306.437	0.000	308.240
	Actuator_Task	_PWM_OFF_2	11	2000	300	0	1.803	1.803	308.240	0.000	310.043
	Actuator_Task	_PWM_OFF_3	11	2000	300	0	1.569	1.569	310.043	0.000	311.612
Core_2	Timing_Events	Visibility_Timer	14	5820000		0	0.400	0.400	0.000	52.829	53.229
	Timing_Events	Lost_Comm_Timer	14	1.73E+08		0	0.226	0.226	0.000	53.003	53.229
	Timing_Events	Deferred_TC_Timer	14	2000		0	17.482	17.482	0.000	35.747	53.229
	Radio_Listener		10	2000		0	2.630	2.630	0.000	35.121	55.859
	TC_Receiver_Task		9	1944		0	38.049	38.049	0.000	35.121	91.278
	TC_Handler		8	1909		0	36.560	36.560	0.000	35.121	130.468
	EV_Handler		7	2000		0	66.500	66.500	0.000	13.1	174.947
	Wdt_Psu		2	250		0	5.500	5.500	0.000	0.000	167.347
	Wdt_Fpga		1	5000		0	0.018	0.018	0.000	0.000	207.598

First iteration

Core 0

Core 0 analysis is relatively simple, since it only hosts one task, so no interference or arrival blocking can occur. Since only one activation per task is allowed at any given time, the only perturbation to the Housekeeping_Task is the shared resources synchronization. Table 6.10 summarizes the Housekeeping_Task shared resource usage per worst-case activation.

Then, for each shared resource, the usage from all other tasks has to be analysed. During the TC phase, the Manager.Event_Buffer is accessed as described by table 6.11. As can be seen the resource is accessed from 2 different cores, i.e $|map(G(\text{Manager.Event\_Buffer}))| = 2$. This implies that, in the worst case, each access can suffer at most 1 request spin delay (a request coming from the other core). The general access cost is then $e^{\text{Manager.Event\_Buffer}} = 2 \times 0.026\,\text{ms} = 0.052\,\text{ms}$. However, given task periods, consider this access cost for all Housekeeping_Task requests would be pessimistic, since tasks from Core 2 only issue up to 6 access requests during a Housekeeping_Task activation. As a result, the worst case spin delay suffered by Housekeeping_Task while accessing Manager.Event_Buffer under MSRP is $5 \times 0.026\,\text{ms} + 1 \times 0.025\,\text{ms} = 0.155\,\text{ms}$.

The Platform.Configuration shared resource usage is summarized in table 6.12. In this case, the resource is accessed from two processors (Core 0 and Core 2). The access from the TC_Handler includes a nested access to the EEPROM memory shared resource. The EEPROM resource can be accessed via nested calls from Core 0 and Core 2. As a result, it might seem that each access could lead to spin delay. However the only task that can cause spin delay to the TC_Handler accessing the EEPROM is the Housekeeping_Task. Thus considering that transitive blocking for the Housekeeping_Task is clearly pessimistic, as the Housekeeping_Task cannot be accessing the EEPROM and spinning to access the Platform.

Table 6.10: Housekeeping_Task resource usage.

Resource	Interface	n	WCET
Manager.Event_Buffer	Put	23	0.026 ms
Platform.Configuration	Is_Enable	22	0.025 ms
Platform.Table	Save	1	31.498 ms
	Set_State	22	0.025 ms
	Set_Value	56	0.025 ms
Ebox.Ebox_Access	Get	47	3.900 ms
	Is_Enabled	44	0.381 ms
	Battery_warnings	1	0.040 ms
I2C.I2C_Access	Mission_Clock	1	0.831 ms
ACS.MGM_Data	Get	1	0.147 ms

Table 6.11: Manager.Event_Buffer resource usage during TC phase.

map(G)	Task / Resource	Interface	n	WCET	Inner Resources
Core 0	Housekeeping_Task	Put	23	0.026 ms	$\emptyset$
Core 2	TC_Handler	Put	1	0.026 ms	$\emptyset$
	EV_Handler	Get	1	0.025 ms	$\emptyset$
	Manager.Mode	Put	1	0.026 ms	$\emptyset$
	TC_Receiver_Task	Put	1	0.026 ms	$\emptyset$
	Lost_Communication_TE	Put	1	0.026 ms	$\emptyset$
	Deferred_timers.Schedule	Put	1	0.026 ms	$\emptyset$

Table 6.12: Platform.Configuration resource usage during TC phase.

map(G)	Task / Resource	Interface	n	WCET	Inner Resources
Core 0	Housekeeping_Task	Is_Enable	22	0.025 ms	$\emptyset$
Core 2	TC_Handler	Enable/Disable	1	15.660 ms	EEPROM

Table 6.13: Platform.Table resource usage during TC phase.

map(G)	Task / Resource	Interface	n	WCET	Inner Resources
Core 0	Housekeeping_Task	Save	1	31.498 ms	$2 \cdot$ EEPROM
		Set_State	22	0.025 ms	$\emptyset$
		Set_Value	56	0.025 ms	$\emptyset$
Core 2	Manager.Mode	Set_Op_Mode	1	0.025 ms	$\emptyset$

Configuration at the same time. As a result, the TC_Handler can only delay the Housekeeping_Task as much as its own access time. In consequence, the spin delay suffered due to accessing the Platform.Configuration resource is 15.660 ms due to the TC_Handler remote access.

Regarding the Platform.Table resource, it is only accessed by Housekeeping_Task and the Mode shared resources as described by table 6.13. The Mode resource is, in turn, accessed by the TC_Handler and EV_Handler when a satellite operating mode change is to be logged in the system state table. As a result, the Set_Operating_Mode interface can be accessed twice per Housekeeping_Task activation, causing as much as 2×0.025 ms $=$ 0.050 ms of spin delay. The access to the Platform.Table can also cause spin delay to the Housekeeping_Task due to its Save access, which triggers two write accesses to the EEPROM memory. Since, as previously described, the EEPROM memory is accessed also by Core 2 tasks, each access can suffer a spin delay equal to a write access cost, i.e. 2×15.500 ms $= 31.000$ ms. As a result, the total spin delay due to accessing the Platform.Table is 31.050 ms.

The Ebox.Ebox_Access resource is, as can be seen in table 6.14, accessed from the three cores. The spin delay caused by Core 2 comes only from task WDT_PSU, but this task has a period smaller than the Housekeeping_Task response time. In this case, the number of times it can cause spin delay to Housekeeping_Task is to be calculated using equation 5.17, where the period of study is the Housekeeping_Task response time, resulting:

$$N_{\text{WDT\_PSU}}^{\text{Ebox.Ebox\_Access}}(R_{\text{Housekeeping\_Task}}) = \left\lceil \frac{R_{\text{Housekeeping\_Task}}}{T_{\text{WDT\_PSU}}} \right\rceil \cdot 1 = 2$$

Thus, the maximum spin delay from Core 0 is $2 \times 5.300\,\text{ms} = 10.600\,\text{ms}$. Regarding spin delay due to Core 1, the ACS subsystem behaviour has to be recalled. The ACS works in cycles of 2 s. During the first second the magnetic field measurements are taken by Sensor_Task (1 to 5). During the latter second the execution is calculated and performed (Control_Task, PWM_ON and PWM_OFF (1 to 3)). As the Housekeeping_Task and ACS tasks do not have any offsets, to calculate the worst possible case, the worst of the two ACS seconds is to be taken in to account. As can be seen in table 6.14, the worst case is clearly represented by the Sensor_Tasks (first second of the ACS cycle). Again, using equation 5.17 we can calculate the spin delay suffer accessing each interface. In this case, the periodicity of requests to be considered is not the period of each equivalent Sensor_Task but the offsets between them. Then, the Get interface causes spin delay as much as:

$$\begin{aligned}\text{N}_{\text{Sensor\_Tasks}}^{\text{Ebox.Ebox\_Access(Get)}}(R_{\text{Housekeeping\_Task}}) &= \left\lceil \frac{R_{\text{Housekeeping\_Task}}}{\text{Offset}_{\text{Sensor\_Tasks}}} \right\rceil \cdot 9 \\ &= 18\end{aligned}$$

times per Housekeeping_Task activation, yielding $18 \times 3.900\,\text{ms} = 70.200\,\text{ms}$. In the same way, the spin delay caused by the Is_Enabled interface is:

$$\begin{aligned}\text{N}_{\text{Sensor\_Tasks}}^{\text{Ebox.Ebox\_Access(Is\_Enabled)}}(R_{\text{Housekeeping\_Task}}) &= \left\lceil \frac{R_{\text{Housekeeping\_Task}}}{\text{Offset}_{\text{Sensor\_Tasks}}} \right\rceil \cdot 4 \\ &= 8\end{aligned}$$

Table 6.14: Ebox.Ebox_Access resource usage during TC phase.

map(G)	Task / Resource	Interface	n	WCET	Inner Resources
Core 0	Housekeeping_Task	Get	47	3.900 ms	$\emptyset$
		Is_Enabled	44	0.381 ms	$\emptyset$
		Battery_Warnings	1	0.040 ms	$\emptyset$
Core 1	Sensor_Tasks (x5)	Get	9	3.900 ms	$\emptyset$
		Is_Enabled	4	0.381 ms	$\emptyset$
	PWM_ON	Actuate_MGT	3	0.414 ms	$\emptyset$
	PWM_OFF (x3)	Actuate_MGT	3	0.414 ms	$\emptyset$
Core 2	WDT_PSU	Reset_WDT	1	5.300 ms	$\emptyset$

Table 6.15: I2C.I2C_Access resource usage during TC phase.

map(G)	Task / Resource	Interface	n	WCET	Inner Resources
Core 0	Housekeeping_Task	Mission_Clock	1	0.831 ms	$\emptyset$
Core 2	TC_Handler	Mission_Clock	1	0.831 ms	$\emptyset$
	EV_Handler	Mission_Clock	1	0.831 ms	$\emptyset$

times per activation or $8 \times 0.381\,\text{ms} = 3.048\,\text{ms}$.

The access to the Mission_Clock via the I2C bus is affected by the usage of the resource from Core 0, as can be seen in table 6.15. As $|map(G(\text{I2C.I2C\_Access}))| = 2$, each access can only suffer 1 spin delay from the remote processor. As the Housekeeping_Task only accesses the Mission_Clock once per activation, the spin delay suffered is $1 \times 0.831\,\text{ms} = 0.831\,\text{ms}$.

Finally, the spin delay due to accessing to the ACS.MGM_Data is equal to the access time of the ACS Sensor_Task_5 as is the only remote task accessing the resource, as can be seen in table 6.16, i.e. 0.147 ms.

With these results, the response time of the Housekeeping_Task under

Table 6.16: ACS.MGM_Data resource usage during TC phase.

map(G)	Task / Resource	Interface	n	WCET	Inner Resources
Core 0	Housekeeping_Task	Get	1	0.147 ms	$\emptyset$
Core 1	Sensor_Task_5	Put	1	0.147 ms	$\emptyset$

Table 6.17: Housekeeping_Task spin delay suffered under MSRP. First iteration.

Resource	Spin Delay	WCET	Access Cost
Manager.Event_Buffer	0.155 ms	0.598 ms	0.753 ms
Platform.Configuration	15.660 ms	0.550 ms	16.210 ms
Platform.Table	31.050 ms	33.448 ms	64.498 ms
Ebox.Ebox_Access	83.848 ms	200.104 ms	283.652 ms
I2C.I2C_Access	0.831 ms	0.831 ms	1.662 ms
ACS.MGM_Data	0.147 ms	0.147 ms	0.294 ms
Total	131.691 ms	235.678 ms	367.369 ms

MSRP can be calculated as the sum of its WCET plus the spin delay suffered summarized in table 6.17:

$$R_{\text{Housekeeping\_Task}} = 324.700\,\text{ms} + 131.391\,\text{ms} = 456.091\,\text{ms}$$

Core 1

The analysis of Core 1 is rather similar to Core 0. Although the equivalent task set shown in table 6.4 shows up to 10 equivalent tasks related with the ACS subsystem, in practice they can be analysed as a whole for most regards. As explained in section 6.1.2, activations of ACS tasks cannot overlap by construction and thus do not cause themselves arrival blocking or interference. As a result, without considering accesses to globally shared resources, ACS response times are equal to their WCET values.

The ACS shared resource usage is rather simple, as can be seen in table 6.18. It only accesses the Ebox.Ebox_Access resource to interact with the external devices (magnetometer sensors and magnetorquers for the actuation), the ACS.Parameters to consistently update its configuration and ACS.MGM_Data to share the magnetometer readings with the Housekeeping_Task. It has local shared resources to coordinate its inter-

Table 6.18: ACS tasks resource usage.

Task	Resource	Interface	n	WCET
Sensor_Tasks	Ebox.Ebox_Access	Get	9	3.900 ms
		Is_Enabled	4	0.381 ms
	ACS.Parameters	Get_Calibration	1	0.026 ms
Sensor_Task_5	ACS.MGM_Data	Put	1	0.147 ms
Control_Task	ACS.Parameters	Get_Control	1	0.026 ms
Actuator_Tasks	Ebox.Ebox_Access	Actuate_MGT	3	0.040 ms

nal execution and to pass the required data among tasks, but since all execute on the same processor they do not affect the timing analysis.

All Sensor_Tasks access the Ebox.Ebox_Access resource 13 times per activation (9 to the Get interface and 4 to the Is_Enabled interface). As was presented in table 6.14 the resource is accessed from the other two cores. From Core 2 it is accessed by the WDT_PSU once each 250 ms only interfering once per Sensor_Task activation. On Core 0 it is accessed by the Housekeeping_Task. From all it accesses, the most time consuming one is the Get access performed up to 47 times per activation. As a result, the worst-case spin delay caused by Core 0 is 13×3.900 ms. In total, the 13 accesses to the Ebox.Ebox_Access can cause to Sensor_Tasks up to $1 \times 5.300\,\text{ms} + 13 \times 3.900\,\text{ms} = 56.000\,\text{ms}$ of spin delay. Furthermore, task Sensor_Task_5 also accesses the ACS.MGM_Data resource also shared with the Housekeeping_Task. As shown previously in table 6.16, this access can generate an extra spin delay of 0.147 ms to that specific activation of the Sensor_Task_5.

The Actuator_Tasks access the Ebox.Ebox_Access up to three times each activation. Following the same reasoning as previously, the spin delay suffered per activation is $1 \times 5.300\,\text{ms} + 3 \times 3.900\,\text{ms} = 17.000\,\text{ms}$.

Finally, both the Sensor_Tasks and Control_Task access the resource ACS .Parameters, also shared with the TC_Handler task, as shown in table 6.19. As a result, each sensor and control task can suffer one spin delay of 0.026 ms due to accesses from Core 2.

Table 6.19: ACS.Parameters resource usage during TC phase.

map(G)	Task / Resource	Interface	n	WCET	Inner Resources
Core 1	Sensor_Tasks	Get_Calibration	1	0.026 ms	$\emptyset$
	Control_Task	Get_Control	1	0.026 ms	$\emptyset$
Core 2	TC_Handler	Get_Calibration	1	0.026 ms	$\emptyset$
		Get_Control	1	0.026 ms	$\emptyset$
		Put	1	0.024 ms	$\emptyset$

Table 6.20: ACS tasks spin delay suffered under MSRP.

Task	Resource	Spin Delay	WCET	Access Cost
Sensor_Tasks	Ebox.Ebox_Access	56.000 ms	36.624ms	92.624 ms
	ACS.Parameters	0.026 ms	0.026 ms	0.052 ms
	Total	56.026 ms	36.650 ms	92.676 ms
Sensor_Task_5	ACS.MGM_Data	+0.147 ms	+0.147 ms	+0.294 ms
	Total	56.173 ms	36.997 ms	92.970 ms
Control_Task	ACS.Parameters	0.026 ms	0.026 ms	0.052 ms
Actuator_Tasks	Ebox.Ebox_Access	17.000 ms	0.120 ms	17.120 ms

The spin delay suffered by the ACS tasks is summarized in table 6.20. These values do not push the response time of any task over the 250 ms, what would increase the spin delay suffered due to the WDT_PSU task, so there is no need to reiterate on the calculations.

Core 2

The analysis of Core 2, which hosts the remaining tasks is somehow more complex: as more than one task can be concurrently active, tasks can suffer interference and arrival blocking. Furthermore, those values can be modified due to the spin delay suffered by other tasks. A first iteration response time analysis was presented in table6.9.

The highest-priority executing entities allocated to Core 2 are the Timing_Event handlers Visibility_Timer, Lost_Comm_Timer and Deferred_TC_Timer. As explained before, their response time is function of the sum of their WCET values (since they can prevent each other from executing) plus

Table 6.21: Deferred_TC_Timer resource usage.

Resource	Interface	n	WCET	Inner Resources
Manager.Command_Buffer	Put	1	0.026 ms	∅
TTC.Configuration	Save	1	16.283 ms	∅
Storage.TC_Storage_Buffer	Get	1	0.548 ms	∅

Table 6.22: Lost_Comm_Timer resource usage.

Resource	Interface	n	WCET	Inner Resources
Manager.Event_Buffer	Put	1	0.026 ms	∅

the priority inversion suffered by lower-priority tasks resource accesses. This worst-case arrival blocking is caused, under MSRP, by the longest shared resource access of any lower-priority task in the system. Initially, considering each access cost equal to the resource access time, this highest-access cost is the TTC.Configuration.Save access performed by the Deferred_TC_Timer and TC_Handler, of 16.283 ms.

The Visibility_Timer does not access any shared resource accessed from any other core. The Deferred_TC_Timer, on the contrary, accesses the three shared resources listed in table 6.21. The Manager.Command_Buffer is not globally shared and thus causes no spin delay. The TTC.Configuration and Storage.TC_Storage_Buffer are not globally shared, either. They do, however, perform a nested access to the EEPROM memory. This memory is also accessed by the Housekeeping_Task on Core 0. As can be seen in table 6.10, the Housekeeping_Task performs two nested write accesses to the EEPROM memory per activation, causing 15.500 ms of spin delay each (note that the Get access can also be delayed, as reading is not acceptable either during the EEPROM memory stabilization period). The total spin delay suffered by the Deferred_TC_Timer is then 31.000 ms.

The Lost_Comm_Timer only accesses a remotely accessed shared resource, the Manager.Event_Buffer to notify that contact with the ground station has been lost. As described in table 6.11 this resource is re-

Table 6.23: Radio_Listener resource usage.

Resource	Interface	n	WCET	Inner Resources
HWComm.Receive_Pool	TC/ACK_Received	1	0.094 ms	∅
UART.Modem_UART	Read	4	0.268 ms	∅

Table 6.24: HWComm.Receive_Pool resource usage during TC phase.

map(G)	Task / Resource	Interface	n	WCET	Inner Resources
Core 2	Radio_Listener	TC/ACK_Received	1	0.094 ms	∅
	TC_Receiver_Task	Get_TC	1	0.095 ms	∅

motely accessed by the Housekeeping_Task. Given its usage, the Manager. Event_Buffer causes one access spin delay to all the accessing tasks from Core 2, adding 0.026 ms to their response times.

Regarding regular tasks, the highest-priority one among them is the Radio_Listener task. As previously indicated, it is sporadically activated as a result of data reception from the TTC radio via a UART serial line. The data received, if correct, is passed to higher-level tasks by means of the HWComm.Receive_Pool. The task resource usage is depicted in table 6.23. The UART.Modem_UART used to communicate with the radio is solely used by the Radio_Listener task during the TC phase, and thus does not cause any delay.

The HWComm.Receive_Pool usage is shown in table 6.24. As can be seen, the only other access is from the TC_Receiver_Task that waits for TCs to be processed. As a result, the Radio_Listener does no suffer any spin delay from HWComm.Receive_Pool.

The TC_Receiver_Task does access some other resources, apart from HWComm.Receive_Pool as can be seen in table 6.25. One of its main duties is to control the coverage periods on board, by monitoring the reception of Openlink messages. This includes coordinating with the TM sending tasks by means of the TTC.TM_Basic and TTC.TM_Nominal shared resources. These resources are only accessed by other tasks of the TTC subsystem (allocated on the same processor) and thus do not cause any spin delay. The TC_Receiver_Task is also in charge of reseting the Visibility_Timer and Lost_Comm_Timer, whose associated timing events are also allocated to the same processor as detailed previously, not causing any spin delay. In the same fashion, the TC_Receiver_Task is in charge of programming the delayed execution of telecommands, by scheduling a timing event associated to the Deferred_Timer shared resource. Again, this resource is only accessed from the same core, not causing spin delay. This process, however, includes two accesses to shared resources with nested accesses to the EEPROM memory. These are the TTC.Configuration Save and the Storage.TC_Storage_Buffer Put accesses listed in table 6.25. These accesses, under a traditional MSRP analysis approach, would be considered to cause spin delay to the TC_Receiver_Task. However, applying the holistic analysis described in section 3.2.1 can provide a tighter, less pessimistic analysis. The notion behind this analysis approach is to consider all possible shared resources access patterns in the system that can lead to spin delay and consider the safest worst case one. This is somehow opposite to the traditional approach of analysing each task in an isolated way and consider that all tasks would suffer the worst-case spin delay. As previously shown in Core 0 analysis with the Ebox.Ebox_Access analysis, the number of times the Housekeeping_Task can cause Core 2 tasks spin delay during an activation of the TC_Receiver_Task is:

$$N^{\text{EEPROM(Write)}}_{\text{Housekeeping\_Task}}(R_{\text{TC\_Receiver\_Task}}) = \left\lceil \frac{R_{\text{TC\_Receiver\_Task}}}{T_{\text{Housekeeping\_Task}}} \right\rceil \cdot 2 = 2$$

times. This is exactly the number of times it has caused spin delay to the higher priority Deferred_TC_Timer, whose spin delay is part of the TC_Receiver_Task response time calculation (as interference). As a result, considering spin delay on the TC_Receiver_Task accesses would be unnecessarily pessimistic.

The telecommands received to be executed immediately are passed to the TC_Handler via the Manager.Command_Buffer, which, once again, does

Table 6.25: TC_Receiver_Task resource usage.

Resource	Interface	n	WCET	Inner Resources
HWComm.Receive_Pool	Get_TC	1	0.095 ms	∅
Manager.Event_Buffer	Put	1	0.026 ms	∅
Manager.Command_Buffer	Put	1	0.025 ms	∅
TTC.TM_Basic	Put	1	0.025 ms	∅
TTC.TM_Nominal	Put	1	0.025 ms	∅
TTC.Visibility_Timer	Reset	1	0.077 ms	∅
TTC.Lost_Comm_Timer	Reset	1	0.077 ms	∅
TTC.Deferred_Timer	Schedule	1	4.621 ms	∅
TTC.Configuration	Save	1	16.283 ms	EEPROM
Storage.TC_Storage_Buffer	Put	1	15.608 ms	EEPROM

not cause spin delay. Finally, if there is any issue with the processing of a TC (such as not being properly authenticated, among other eventualities), the event is to be propagated to the Event_Handler via the Manager.Event_Buffer. This resource can cause spin delay, as an be derived from table 6.11. As can be seen in that table, an access request from Core 2 can suffer spin wait as being issued concurrently with a request from the Housekeeping_Task. This spin is bounded to the 0.026 ms of a Put operation on such resource.

The next task in priority order is the TC_Handler that executes the received telecommands, regardless of whether they were scheduled to be executed immediately or at a later time. The shared resource usage of TC_Handler is expressed in table 6.26. The TC_Handler is activated as a result of the reception a command via the Manager.Command_Buffer Get interface. Then, TC_Handler, depending on the satellite operating mode decides whether the command received is to be executed (potentially accessing a plethora of shared resources) or discarded (generating an event to be processed by the Event_Handler). In the latter case, this implies incorporating the event to the Manager.Event_Buffer. This access, as for the TC_Receiver_Task can cause up to 0.026 ms of spin delay.

Regarding the TC execution, only accesses causing spin delay will

Table 6.26: TC_Handler resource usage.

Resource	Interface	n	WCET	I. Resources
Manager.Command_Buffer	Get	1	0.095 ms	∅
Manager.Event_Buffer	Put	1	0.026 ms	∅
Manager.Mode	Handle	1	35.121 ms	2 · EEPROM
Platform.Configuration	Enable/Disable	1	15.660 ms	EEPROM
I2C.I2C_Access	Mission_Clock	1	0.831 ms	∅
ACS.Parameters	Get_Calibration	1	0.026 ms	∅
	Get_Control	1	0.026 ms	∅
	Put	1	0.024 ms	∅
TTC.Configuration	Set_Lost_Comm	1	0.025 ms	∅
	Save	1	16.283 ms	EEPROM
Storage.Config_Param	Update_Platform_Conf	1	15.553 ms	EEPROM
	Get_Platform_Conf	1	0.488 ms	∅
	Update_ACS_Conf	1	15.653 ms	EEPROM
	Get_ACS_Config	2	0.476 ms	∅
Storage.Event_Buffer	Put	1	15.608 ms	EEPROM

be detailed. Changing the platform configuration implies an access to the Platform.Configuration shared resource, whose usage was outlined in table 6.12. As can be seen in that table, accesses from Core 0 can suffer at most 1 spin delay of 0.025 ms as a result of the Housekeeping_Task access. Similar case is that of the I2C.I2C_Access accessing the Mission_Clock (table 6.15) which causes a spin delay equal to 1 access cost (0.831 ms).

The ACS.Parameters usage detailed in table 6.19 presents an interesting case study from the TC_Handler point of view. As has been previously explained, ACS Sensor_Tasks and Control_Task execute with an offset of 200 ms. Thus, concurrence with more than one activation of those tasks happens if the response time of the analysed task is greater than those 200 ms minus the longest response time of the ACS tasks. For the first iteration analysis, it is not the case. Thus, each activation, the TC_Handler can suffer at most 1 spin delay of 0.026 ms coming from the ACS.Parameters resource.

The TC_Handler can also trigger a mode change if the telecommand received indicates so. This implies accessing the Manager.Mode shared resource to process the change, including notifying the other subsystems. This includes the platform subsystem, in which the value is updated in the System_State_Table guarded in the Platform.Table shared resource, i.e. the Manager.Mode includes a nested call to Platform.Table. This can cause spin delay, as the resource might be in use by the Housekeeping_Task, as can be seen in table 6.13. In that table, the worst case access performed from the remote Core 0 is of 31.498 ms. This access includes 2 write accesses to the EEPROM memory. Since, as explained before, all the possible spin delay caused by those remote accesses has already been accounted for, the only extra spin delay to include in the analysis is the remaining Platform.Table access time.

$$\begin{aligned} spin^{\text{Platform.Table}}_{\text{TC\_Handler}} &= c^{\text{Platform.Table}}_{\text{Housekeeping\_Task}} - 2 \times c^{\text{EEPROM}}(\text{Write}) \\ &= 31.498\,\text{ms} - 2 \times 15.500\,\text{ms} \\ &= 0.498\,\text{ms} \end{aligned}$$

It is also relevant to note that the accesses listed in table 6.26 include all accesses that can be triggered by the TC_Handler. However, not all of them are to be done on each activation. On the contrary, the specific access pattern is highly dependant on the telecommand to be processed. The worst case is considered for the present analysis.

The Event_Handler study is somehow similar to that of the TC_Handler. While able to access the relation of shared resources presented in table 6.27, only a subset of them are accessed on each activation depending on the event to be processed. An access that is always performed is the access to the Manager.Event_Buffer to get the event to be processed. Since, as shown in table 6.11 the resource can be accessed by the Housekeeping_Task on Core 0, it can cause 0.026 ms of spin delay. In the same way, the Manager.Mode access includes the same nested call to the Platform.Table as in the TC_Handler case. In this case, since the worst-case access by the Housekeeping_Task has been accounted for, the second worst is to be taken into account. As can be seen in table 6.13 regardless of

Table 6.27: Event_Handler resource usage.

Resource	Interface	n	WCET	Inner Resources
Manager.Event_Buffer	Get	1	0.026 ms	$\emptyset$
Manager.Mode	Handle	1	35.121 ms	$2 \cdot$ EEPROM
I2C.I2C_Access	Mission_Clock	1	0.831 ms	$\emptyset$
Storage.Event_Buffer	Put	1	15.608 ms	EEPROM
Storage.Conf_Param	Update_Storage_Conf	1	15.585 ms	EEPROM

the access considered this value is of 0.026 ms. Finally, also similar to the TC_Handler, the two extra nested accesses to the EEPROM memory as part of the Storage.Event_Buffer and Storage.Update_Storage_configuration are not to be accounted for any extra delay, as it is limited to two accesses as explained before.

The watchdog timer implemented in the EBOX board, updated by the WDT_PSU task can cause this spin delay due to its usage of the Ebox. Ebox_Access shown in table 6.14. The WDT_PSU accesses once per activation to the resource, and thus its maximum spin delay is equal to the worst case accesses from remote processors. The maximum spin delay is then equal to two Get accesses, performed by the ACS Sensor_Tasks and Housekeeping_Task respectively, i.e. $2 \times 3.900\,\text{ms} = 7.800\,\text{ms}$.

Finally, the WDT_FPGA task refreshes a watchdog timer implemented on the computer board, directly accessible from software which does not require any mutual exclusion and thus does not cause any spin delay.

Table 6.28 summarizes the spin delay suffered by tasks allocated to Core 2 based on the first analysis iteration.

Second iteration

The results presented in table 6.29 imply that some of the spin delays calculated based on remote processors usage periods and local response

Table 6.28: Core 2 tasks spin delay suffered under MSRP. First iteration.

Task	Resource	Spin Delay	WCET	Acces Cost
Visibility_Timer	-	-	-	-
Lost_Comm_Timer	Manager.Event_Buffer	0.026 ms	0.026 ms	0.052 ms
Deferred_TC_Timer	EEPROM	31.000 ms	16.831 ms	47.831ms
Radio_Listener	-	-	-	-
TC_Receiver_Task	Manager.Event_Buffer	0.026 ms	0.026 ms	0.052 ms
TC_Handler	ACS.Parameters	0.026 ms	0.026 ms	0.052 ms
	I2C.I2C_Access	0.831 ms	0.831 ms	1.662 ms
	Platform.Table	31.121 ms	0.498 ms	35.619 ms
	Total	31.147 ms	1.344 ms	37.333 ms
Event_Handler	Manager.Event_Buffer	0.026 ms	0.026 ms	0.052 ms
	Manager.Mode	0.025 ms	35.121 ms	35.146 ms
	Storage.Event_Buffer	0.000 ms	15.608 ms	15.608 ms
	Storage.Conf_Param	0.000 ms	15.585 ms	15.585 ms
	Total	0.051 ms	66.340 ms	66.391 ms
WDT_PSU	Ebox.Ebox_Access	7.800 ms	5.300 ms	13.100 ms
WDT_FPGA	-	0.000 ms	0.000 ms	0.000 ms

times have to be reevaluated. In particular, the Ebox.Ebox_Access spin delay suffered by the Housekeeping_Task in Core 0 and the ACS.Parameters spin delay suffered by the TC_Handler task in Core 2 have to be recalculated.

Core 0

With the worst-case response time obtained in the previous iteration, the spin delay suffered accessing the Ebox.Ebox_Access has to be recalculated. As $\left\lceil \frac{R_{\text{Housekeeping\_Task}}}{Offset_{\text{Sensor\_Tasks}}} \right\rceil$ is now 3 instead of 2, the spin delay caused by the Sensor_Tasks is $3 \times 9 \times 3.900\,\text{ms} = 105.300\,\text{ms}$ and $3 \times 4 \times = 4.572\,\text{ms}$ being in total 109.872 ms. This, added to the 10.600 ms of spin delay due to the WDT_PSU (not to be recalculated) results in a total spin delay of 120.472 ms. Updating the values in table 6.30, the second iteration of Housekeeping_Task response time is:

Table 6.29: MSRP response time analysis. First iteration.

	Task	Subtasks	Period	Offset	Spin	WCET	C	Jitter	B	R	
C_0	Housekeeping_Task		6	1000		131.691	324.700	456.391	0.000	0 .000	456.391
Core_1	Sensor_Task	_1 a _4	13	2000	200	56.026	74.093	130.119	0.000	0.000	130.119
	Sensor_Task	_5	13	2000	200	56.173	74.240	130.413	0.000	0.000	130.413
	Control_Task		12	2000	1000	0.026	4.541	4.567	0.000	0.000	4.567
	Actuator_Task	_PWM_ON	11	2000	1000	17.000	1.896	18.896	4.567	0.000	23.463
	Actuator_Task	_PWM_OFF_1	11	2000	300	17.000	1.803	18.803	323.463	0.000	342.266
	Actuator_Task	_PWM_OFF_2	11	2000	300	17.000	1.803	18.803	342.266	0.000	361.069
	Actuator_Task	_PWM_OFF_3	11	2000	300	17.000	1.569	18.569	361.069	0.000	379.638
Core_2	Timing_Events	Visibility_Timer	14	5820000		0.000	0.400	0.400	0.000	83.8550	84.255
	Timing_Events	Lost_Comm_Timer	14	1.73E+08		0.026	0.226	0.252	0.000	84.003	84.255
	Timing_Events	Deferred_TC_Timer	14	2000		31.000	17.482	48.482	0.000	35.773	84.255
	Radio_Listener		10	2000		0.000	2.630	2.630	0.000	35.121	86.885
	TC_Receiver_Task		9	1913		0.026	38.049	38.075	0.000	35.121	122.330
	TC_Handler		8	1878		1.355	36.560	37.915	0.000	35.121	162.875
	EV_Handler		7	2000		0.051	66.500	66.551	0.000	13.100	207.405
	Wdt_Psu		2	250		7.800	5.500	13.300	0.000	0.000	207.605
	Wdt_Fpga		1	5000		0.000	0.018	0.018	0.000	0.000	247.856

Table 6.30: Housekeeping_Task spin delay suffered under MSRP. Second iteration.

Resource	Spin Delay	WCET	Access Cost
Manager.Event_Buffer	0.155 ms	0.598 ms	0.753 ms
Platform.Configuration	15.660 ms	0.550 ms	16.210 ms
Platform.Table	31.050 ms	33.448 ms	64.498 ms
Ebox.Ebox_Access	120.472 ms	200.104 ms	320.576 ms
I2C.I2C_Access	0.831 ms	0.831 ms	1.662 ms
ACS.MGM_Data	0.147 ms	0.147 ms	0.294 ms
Total	168.315 ms	235.678 ms	403.993 ms

$$R_{\text{Housekeeping\_Task}} = 324.700\,\text{ms} + 168.315\,\text{ms} = 493.015\,\text{ms}$$

Core 2

The TC_Handler spin delay suffered due to the ACS.Parameters has to be recalculated as it can now collide with more than one activation of the ACS tasks. In particular, it can collide with two of them (the end of one plus the beginning of the following one). As a result ACS.Parameters can cause now $2 \times 0.026\,\text{ms} = 0.052\,\text{ms}$ spin delay to TC_Handler, yielding the final spin delay shown in table 6.31 and the response time analysis of the second iteration presented in table 6.32. These values are final, as a third iteration on the analysis yields exactly the same response time values. This means that the response time equation, given the temporal properties of the task set, have reached a fix-point at these response time values, i.e. converges at these response time values.

Summary

The presented response times for the MSRP present a notable improvement for the tasks highlighted for the monocore implementation. The Housekeeping_Task has reduced its response time value from

Table 6.31: Core 2 tasks spin delay suffered under MSRP. Second iteration.

Task	Resource	Spin Delay	WCET	Acces Cost
Visibility_Timer	-	-	-	-
Lost_Comm_Timer	Manager.Event_Buffer	0.026 ms	0.026 ms	0.052 ms
Deferred_TC_Timer	EEPROM	31.000 ms	16.831 ms	47.831ms
Radio_Listener	-	-	-	-
TC_Receiver_Task	Manager.Event_Buffer	0.026 ms	0.026 ms	0.052 ms
TC_Handler	ACS.Parameters	0.052 ms	0.026 ms	0.078 ms
	I2C.I2C_Access	0.831 ms	0.831 ms	1.662 ms
	Platform.Table	31.121 ms	0.498 ms	35.619 ms
	Total	31.147 ms	1.344 ms	37.333 ms
Event_Handler	Manager.Event_Buffer	0.026 ms	0.026 ms	0.052 ms
	Manager.Mode	0.025 ms	35.121 ms	35.146 ms
	Storage.Event_Buffer	0.000 ms	15.608 ms	15.608 ms
	Storage.Conf_Param	0.000 ms	15.585 ms	15.585 ms
	Total	0.051 ms	66.340 ms	66.391 ms
WDT_PSU	Ebox.Ebox_Access	7.800 ms	5.300 ms	13.100 ms
WDT_FPGA	-	0.000 ms	0.000 ms	0.000 ms

862.459 ms to 493.015 ms, leaving a clear margin with its implicit deadline of 1000 ms. This improvement is consequence of the chosen task allocation, in which this task is executed alone on its host processor. As a result, it no longer suffers from interference or arrival blocking, highly reducing its response time. The difference now between its response time and its WCET value is solely due to the overhead consequence of the resource sharing among processors.

The other highlighted task during the monocore analysis was the Wdt_Psu. This task response time has notably reduced its response time to 207.631 ms, ensuring that its implicit deadline is always met.

It is worth nothing to mention that some tasks have now longer response times. These tasks are, in general, the tasks assigned the highest priorities in the monocore implementation. This is so due to the increase of the priority inversion as a result of the non-preemptable access policy to globally shared resources under MSRP. Examples of

Table 6.32: MSRP response time analysis. Second iteration.

	Task	Subtasks	Priority	Period	Offset	Spin	WCET	C	Jitter	B	R
C_0	Housekeeping_Task		6	1000		168.315	324.700	493.015	0.000	0.000	493.015
Core_1	Sensor_Task	_1 a _4	13	2000	200	56.026	74.093	130.119	0.000	0.000	130.119
	Sensor_Task	_5	13	2000	200	56.173	74.240	130.413	0.000	0.000	130.413
	Control_Task		12	2000	1000	0.026	4.541	4.567	0.000	0.000	4.567
	Actuator_Task	_PWM_ON	11	2000	1000	17.000	1.896	18.896	4.567	0.000	23.463
	Actuator_Task	_PWM_OFF_1	11	2000	300	17.000	1.803	18.803	323.463	0.000	342.266
	Actuator_Task	_PWM_OFF_2	11	2000	300	17.000	1.803	18.803	342.266	0.000	361.069
	Actuator_Task	_PWM_OFF_3	11	2000	300	17.000	1.569	18.569	361.069	0.000	379.638
Core_2	Timing_Events	Visibility_Timer	14	5820000		0.000	0.400	0.400	0.000	83.855	84.255
	Timing_Events	Lost_Comm_Timer	14	1.73E+08		0.026	0.226	0.252	0.000	84.003	84.255
	Timing_Events	Deferred_TC_Timer	14	2000		31.000	17.482	48.482	0.000	35.773	84.255
	Radio_Listener		10	2000		0.000	2.630	2.630	0.000	35.121	86.885
	TC_Receiver_Task		9	1913		0.026	38.049	38.075	0.000	35.121	122.330
	TC_Handler		8	1878		1.381	36.560	37.941	0.000	35.121	162.901
	EV_Handler		7	2000		0.051	66.500	66.551	0.000	13.100	207.431
	Wdt_Psu		2	250		7.800	5.500	13.300	0.000	0.000	207.631
	Wdt_Fpga		1	5000		0.000	0.018	0.018	0.000	0.000	247.882

this effect are the Timing_Events, that have increased their response time from values below 50 ms in the monocore implementation for the TC phase to values over 84 ms in the triple core implementation under MSRP for the same phase.

6.4 Analysis of a Triple Core Implementation Using MrsP

In this section the analysis of the UPMSat-2 three core implementation during the telecommand phase, being the shared resource access policy MrsP, will be presented. This analysis will be later compared in section 6.5 with those of the monocore implementation and the multicore implementation under MSRP.

The preliminary analysis of the system is similar to that of MSRP presented in table 6.9, but with the notable difference of the blocking presented by tasks on Core 2. As was discussed in previous sections, the blocking under MrsP is ruled by the ceiling blocking policy rather than the non-preemptive policy. This limits the effect of arrival blocking on higher-priority tasks, as can be seen in table 6.33.

In particular, it is important to note that the initial blocking and thus response time of the higher-priority entities, the timing event handlers, are equal to their respective values for the monocore implementation, shown in table 6.7. For any other task, the response time is lower, since the interference has been reduced thanks to the allocation of tasks on three different cores. It is also important to note that, as a result of the new allocation, the shared resources get assigned a different ceiling priority on each core, depending on the tasks allocated to each of them. The local ceiling priorities used for this study are presented in table 6.34. Note that, as explained before, and will be detailed later, Cores 0 and 1 can only have one active task at

Table 6.33: MrsP response time analysis. Preliminary analysis. All time values in ms.

	Task	Subtasks	Priority	Period	Offset	I	E	WCET	C	Jitter	B	R
C_0	Housekeeping_Task		6	1000			0	324.700	89.022	0.000	0.000	324.700
	Sensor_Task	_1 a _4	13	2000	200		0	74.093	37.443	0.000	0.000	74.093
Core_1	Sensor_Task	_5	13	2000	200		0	74.240	37.443	0.000	0.000	74.240
	Control_Task		12	2000	1000		0	4.541	4.515	0.000	0.000	4.541
	Actuator_Task	_PWM_ON	11	2000	1000		0	1.896	1.776	4.541	0.000	6.437
	Actuator_Task	_PWM_OFF_1	11	2000	300		0	1.803	1.683	306.437	0.000	308.240
	Actuator_Task	_PWM_OFF_2	11	2000	300		0	1.803	1.683	308.240	0.000	310.043
	Actuator_Task	_PWM_OFF_3	11	2000	300		0	1.569	1.449	310.043	0.000	311.612
Core_2	Timing_Events	Visibility_Timer	14	5820000			0	0.400	0.400	0.000	33.293	33.693
	Timing_Events	Lost_Comm_Timer	14	1.73E+08			0	0.226	0.226	0.000	33.467	33.693
	Timing_Events	Deferred_TC_Timer	14	2000			0	17.482	16.934	0.000	16.211	33.693
	Radio_Listener		10	2000			0	2.630	1.464	0.000	15.585	36.323
	TC_Receiver_Task		9	1964			0	38.049	1.187	0.000	15.585	71.742
	TC_Handler		8	1928			0	36.560	0.582	0.000	35.121	130.468
	EV_Handler		7	2000			0	66.500	0.160	0.000	0.000	161.847
	Wdt_Psu		2	250			0	5.500	0.200	0.000	0.000	167.347
	Wdt_Fpga		1	5000			0	0.018	0.018	0.000	0.000	207.598

a time by construction. As a result, there is little benefit in assigning any specific priority to those lone tasks. Hence, and for the sake of simplicity and clarity, the same priority as in the monocore implementation has been maintained for this study. This does not affect the response time analysis results. Furthermore, since resource priorities do only affect one core (Core 2), ceiling priorities of that processor can be assigned globally (some systems might require that shared resources have a single ceiling priority, such as those implemented using Ada).

The rest of this section will be devoted to the study of the resource sharing among processors and its effect on the response times.

Table 6.34: UPMSat-2 shared resources local priorities under MrsP.

Subsystem	Protected Object	Core 0	Core 1	Core 2
Manager	Commissioning	-	-	8
	Event_Buffer	6	-	14
	Command_Buffer	-	-	14
	Mode	-	-	8
Platform	Configuration (Platform)	6	-	8
	Table	6	-	8
	Ebox_Access	6	13	2
	I2C_Access	6	-	14
	Buffer	-	-	14
ACS	MGM_Data	6	13	-
	MGM_Signals	-	13	-
	MGT	-	12	-
	Parameters	-	12	8
TTC	Visibility_Timer	-	-	14
	Lost_Communications_Timer	-	-	14
	Deferred_Timers	-	-	14
	Configuration (TTC)	-	-	14
	TM_Basic	-	-	9
	TM_Nominal	-	-	9
	Receive_Pool	-	-	10
Storage	Configuration_Parameters	-	-	14
	Storage_Buffers	-	-	14
	EEPROM	-	-	14

First iteration

Core 0

As previously presented, Core 0 only hosts the Housekeeping_Task, whose resource access pattern was depicted in table 6.10. The access cost for each of the six globally shared resources will be now calculated.

The first resource to be analysed is the Manager.Event_Buffer. Table 6.11 summarized the relevant information to analyse the overhead caused by this resource under MSRP. However, as thoroughly reviewed in chapter 4, this information is not enough for MrsP. While under MSRP accesses performed directly from tasks or under shared resources were considered equal, under MrsP nested resources require a specific analysis [71]. Table 6.35 presents such information. The direct spin delay suffered during the first access to the resource is calculated as follows: the resource is accessed from two processors, i.e. $|map(G(\text{Manager.Event\_Buffer}))| = 2$ plus is an inner resource of other two resources, i.e. $|V(\text{Manager.Event\_Buffer})| = 2$. As discussed in section 4, the resource FIFO queue under MrsP can be as long as $|map(G)|+|V|$. Since it is the first access and Housekeeping_Task has no higher-priority tasks, all those accesses can potentially cause direct spin delay. Thus, the first access cost is the sum of the Housekeeping_Task access time, plus the worst access time from Core 2 plus a nested call from resources Manager.Mode and Deferred_Timers.Schedule:

Table 6.35: Manager.Event_Buffer resource usage summary during TC phase under MrsP.

map(G)	Task	Interface	n	WCET
Core 0	Housekeeping_Task	Put	23	0.026 ms
Core 2	TC_Handler	Put	1	0.026 ms
	EV_Handler	Get	1	0.025 ms
	TC_Receiver_Task	Put	1	0.026 ms
	Lost_Communication_TE	Put	1	0.026 ms
	Resource	Interface	n	WCET
	Manager.Mode	Put	1	0.026 ms
	Deferred_Timers.Schedule	Put	1	0.026 ms

$map(G)$	$\|map(G)\|$	$\|V(G)\|$	Q
Core 0, Core 2	2	2	4

$$
\begin{aligned}
e^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(1(\text{Put})) &= e'^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(1(\text{Put})) \\
&\quad + e'^{\text{Manager.Event\_Buffer}}_{\text{TC\_Receiver\_Task}}(1(\text{Put})) \\
&\quad + e'^{\text{Manager.Event\_Buffer}}_{\text{Manager.Mode}}(1(\text{Put})) \\
&\quad + e'^{\text{Manager.Event\_Buffer}}_{\text{Deferred.Timers}}(1(\text{Put})) \\
&= 0.026\,\text{ms} + 0.026\,\text{ms} + 0.026\,\text{ms} + 0.026\,\text{ms} \\
&= 0.104\,\text{ms}
\end{aligned}
$$

The second access, however, presents a different analysis for nested resource calls. The Manager.Mode resource is accessed by two tasks, TC_Handler and Event_Handler. Both can cause spin delay to Housekeeping_Task once, as discussed for the MSRP analysis. Resource Deferred_Timers is, on the contrary, accessed by only one task, the TC_Handler. As

$$
\text{N}^{\text{Deferred\_Timers.Schedule}}_{\text{TC\_Handler}}(R_{\text{Housekeeping\_Task}}) = \left\lceil \frac{R_{\text{Housekeeping\_Task}}}{T_{\text{TC\_Handler}}} \right\rceil = 1
$$

the Deferred_Timers resource can only cause one spin delay per activation due to accessing the Manager.Event_Buffer as an inner resource, and it was already included in the first access cost calculation. As a result, the second access cost to the Manager.Event_Buffer is of only:

$$\begin{aligned}
e^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(2(\text{Put})) &= e'^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(2(\text{Put})) \\
&\quad + e'^{\text{Manager.Event\_Buffer}}_{\text{TC\_Handler}}(1(\text{Put})) \\
&\quad + e'^{\text{Manager.Event\_Buffer}}_{\text{Manager.Mode}}(1(\text{Put})) \\
&= 0.026\,\text{ms} + 0.026\,\text{ms} + 0.026\,\text{ms} \\
&= 0.078\,\text{ms}
\end{aligned}$$

The third access can only suffer spin delay due to the remaining Put access from Core 2, as all nested resource accesses have been accounted for. The fourth access can also be delayed by Core 2, but due to a Get access, which is slightly shorter. Finally, the remaining accesses do not suffer any spin delay, and thus the access cost is equal to the Housekeeping_Task access time:

$$\begin{aligned}
e^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(3(\text{Put})) &= e'^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(3(\text{Put})) \\
&\quad + e'^{\text{Manager.Event\_Buffer}}_{\text{Lost\_Comm}}(1(\text{Put})) \\
&= 0.026\,\text{ms} + 0.026\,\text{ms} \\
&= 0.052\,\text{ms}
\end{aligned}$$

$$\begin{aligned}
e^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(4(\text{Put})) &= e'^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(4(\text{Put})) \\
&\quad + e'^{\text{Manager.Event\_Buffer}}_{\text{EV\_Handler}}(1(\text{Get})) \\
&= 0.026\,\text{ms} + 0.026\,\text{ms} \\
&= 0.051\,\text{ms}
\end{aligned}$$

Table 6.36: Platform.Configuration resource usage summary during TC phase under MrsP.

map(G)	Task	Interface	n	WCET
Core 0	Housekeeping_Task	Is_Enable	22	0.025 ms
Core 2	TC_Handler	Enable/Disable	1	15.660 ms

$map(G)$	$\|map(G)\|$	$\|V(G)\|$	Q
Core 0, Core 2	2	0	2

$$\begin{aligned} e^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(5..23(\text{Put})) &= e'^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(5..23(\text{Put})) \\ &= 18 \times 0.026\,\text{ms} \\ &= 0.468\,\text{ms} \end{aligned}$$

being as a result the total access time to the resource during a Housekeeping_Task activation:

$$\begin{aligned} e^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}} &= e^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(1(\text{Put})) \\ &\quad + e^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(2(\text{Put})) \\ &\quad + e^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(3(\text{Put})) \\ &\quad + e^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(4(\text{Put})) \\ &\quad + e^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(5..23(\text{Put})) \\ &= 0.104\,\text{ms} + 0.078\,\text{ms} + 0.052\,\text{ms} \\ &\quad + 0.051\,\text{ms} + 0.468\,\text{ms} \\ &= 0.753\,\text{ms} \end{aligned}$$

The Platform.Configuration shared resource is only accessed directly by tasks as shown in 6.36. As discussed for MSRP, each access to the resource can be concurrent with one coming from Core 2, remaining the rest unaffected due to remote calls. However, the calls from Core 2 include a nested call to Storage.Configuration_Parameters resource. Fortunately, that resource is only accessed from calls invoked by tasks

Table 6.37: Shared resources accessing the EEPROM memory and local ceiling priorities.

Resource	Core 0	Core 1	Core 2
Platform.Table	14	-	-
Platform.Configuration	14	-	8
TTC.Configuration	14	-	14
Manager.Mode	-	-	8
Storage.Configuration_Parameters	-	-	14
Storage.Event_Buffer	-	-	14
Storage.TC_Buffer	-	-	14

allocated to Core 2 during TC phase. As a result, the access times of remote tasks to Platform.Configuration is just the sum of the Platform. Configuration.Enable/Disable interface and Storage.Update_Platform_Parameters interface execution. The Storage.Update_Platform_Parameters interface, however, accesses the EEPROM memory, which is a globally shared resource, receiving the access summarized in table 6.37. This usage, given the general analysis rules provided in section 4 could lead to a very pessimistic analysis. As the EEPROM memory is accessed from up to 7 shared resources, general analysis equations would indicate that as many as 7 accesses could be queued at a time to access the EEPROM memory. To yield such pessimistic result, a set of conditions are required for the outer resources accessing it: the outer resources have to be accessed from more than one processor and they cannot have the highest priority in the processor (so they can be preempted and migrations triggered). In any other case, they act as a regular task accessing the resource: they effectively prevent any other access to the resource from their host processor until they release the associated lock.

As can be seen in table 6.37, only three shared resources are accessed from more than one core (during the TC phase). From them, only the Platform.Configuration is preemptable on its host processor. This means that, up to two access request to the EEPROM memory could be issued at a time from Core 2: if an access to Platform.Configuration is preempted (with priority) on Core 2 and migrated to Core 0, a higher-priority task can request access to the EEPROM memory, before or during the access requested by the Platform.Configuration accessing task. This would present a worst-case spin delay of two accesses for tasks on Core 0.

However, a wise modification of Core 2 local ceiling priorities to prevent such a pessimistic analysis can be devised: as the arrival blocking suffered for high-priority tasks on Core 2 is higher than the Platform.Configuration access time, the response time analysis would not be affected if the local ceiling priority of Platform.Configuration is artificially increased to 14 (being non-preemptable in practice). As no EEPROM outer resource can now be preempted, its access cost is now equal to that of MSRP.

As now the access to the EEPROM by TC_Handler cannot be delayed, the maximum spin delay suffered by the Housekeeping_Task accessing the Platform.Configuration is just the access time of 15.660 ms previously identified. The access cost is then:

$$e^{\text{Platform.Configuration}}_{\text{Housekeeping\_Task}}(1) = 0.025\,\text{ms} + 15.660\,\text{ms} = 15.685\,\text{ms}$$

$$e^{\text{Platform.Configuration}}_{\text{Housekeeping\_Task}}(2..22) = 0.025\,\text{ms}$$

$$e^{\text{Platform.Configuration}}_{\text{Housekeeping\_Task}} = 15.685\,\text{ms} + 21 \times 0.025\,\text{ms} = 16.210\,\text{ms}$$

The Platform.Table shared resource is only accessed directly by the Housekeeping_Task as shown in 6.36. It is also accessed as an inner resource by the Manager.Mode resource. This resource is only accessed by tasks

TC_Handler and EV_Handler allocated on Core 2, so it is not strictly a globally shared resource, and thus behaves as a direct task access with regard to spin delay. It can then generate one spin delay to be accounted for during the worst-case Housekeeping_Task access. This access is to the Save interface of the resource, which includes two EEPROM accesses. These accesses, as discussed above can suffer a spin delay equal to the longest access to the EEPROM from Core 2.

$$\begin{aligned} e^{\text{Platform.Table}}_{\text{Housekeeping\_Task}}(1(\text{Save})) = {} & e'^{\text{Platform.Table}}_{\text{Manager.Mode}}(1) \\ & + c^{\text{Platform.Table}}_{\text{Housekeeping\_Task}}(1(\text{Save})) \\ & + e^{\text{EEPROM}}_{\text{Platform.Table}}(1) \\ & + e^{\text{EEPROM}}_{\text{Platform.Table}}(2) \end{aligned}$$

where:

$$\begin{aligned} e^{\text{EEPROM}}_{\text{Platform.Table}}(1) = e^{\text{EEPROM}}_{\text{Platform.Table}}(2) &= 2 \times c^{\text{EEPROM}} \\ &= 2 \times 15.500\,\text{ms} \\ &= 31.000\,\text{ms} \end{aligned}$$

being thus the Save access cost:

$$\begin{aligned} e^{\text{Platform.Table}}_{\text{Housekeeping\_Task}}(1(\text{Save})) &= 0.025\,\text{ms} + 31.498\,\text{ms} + 31.000\,\text{ms} \\ &= 62.523\,\text{ms} \end{aligned}$$

Having considered one spin delay on the worst access to the resource, there is one spin delay remaining to be considered. Following accesses are executed without contention, and are their costs equal their access times:

$$\begin{aligned} e^{\text{Platform.Table}}_{\text{Housekeeping\_Task}}(1(\text{Set\_State})) &= e'^{\text{Platform.Table}}_{\text{Manager.Mode}}(2) \\ &\quad + c^{\text{Platform.Table}}_{\text{Housekeeping\_Task}}(1(\text{Set\_State})) \\ &= 0.025\,\text{ms} + 0.025 ms \\ &= 0.050\,\text{ms} \end{aligned}$$

$$
\begin{aligned}
e^{\text{Platform.Table}}_{\text{Housekeeping\_Task}}(2..22(\text{Set\_State})) &= c^{\text{Platform.Table}}_{\text{Housekeeping\_Task}}(2..22(\text{Set\_State}))\\
&= 21 \times 0.025\,\text{ms}\\
&= 0.525\,\text{ms}
\end{aligned}
$$

$$
\begin{aligned}
e^{\text{Platform.Table}}_{\text{Housekeeping\_Task}}(1..56(\text{Set\_Value})) &= c^{\text{Platform.Table}}_{\text{Housekeeping\_Task}}(1..56(\text{Set\_Value}))\\
&= 56 \times 0.025\,\text{ms}\\
&= 1.400\,\text{ms}
\end{aligned}
$$

The total access cost of the Housekeeping_Task to the is Platform.Table resource is then:

$$
\begin{aligned}
e^{\text{Platform.Table}}_{\text{Housekeeping\_Task}} &= e^{\text{Platform.Table}}_{\text{Housekeeping\_Task}}(1(\text{Save}))\\
&\quad + e^{\text{Platform.Table}}_{\text{Housekeeping\_Task}}(1(\text{Set\_State}))\\
&\quad + e^{\text{Platform.Table}}_{\text{Housekeeping\_Task}}(2..22(\text{Set\_State}))\\
&\quad + e^{\text{Platform.Table}}_{\text{Housekeeping\_Task}}(1..56(\text{Set\_Value}))\\
&= 62.523\,\text{ms} + 0.050\,\text{ms} + 0.525\,\text{ms} + 1.400\,\text{ms}\\
&= 64.498\,\text{ms}
\end{aligned}
$$

The Ebox.Ebox_Access shared resource usage was presented in table 6.14. As can be seen in table 6.41, the resource is only accessed directly by tasks, not being involved in nested calls. As was presented for the MSRP analysis, the initial analysis presented in table 6.33 shows that the worst-case concurrence from Core 1 is suffered when the Housekeeping_Task execution is executed in parallel with up to two Sensor_Tasks. These tasks, as shown in table 6.41 access 9 times the resource to measure an analog signal and 4 times to check whether its required devices are enabled. The concurrence from Core 2 is reduced to two activation of the WDT_PSU task. Given the intensive use of the Ebox.Ebox_Access via the Get interface, all Housekeeping_Task spin delay will be accounted for in those accesses, since are the longest. The first access also suffers the spin delay from Core 2:

$$\begin{aligned}
e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1,2(\text{Get})) &= e'^{\text{Ebox.Ebox\_Access}}_{\text{Sensor\_Tasks}}(1,2(\text{Get})) \\
&\quad + e'^{\text{Ebox.Ebox\_Access}}_{\text{WDT\_PSU}}(1,2(\text{Reset\_WDT})) \\
&\quad + c^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1,2(\text{Get})) \\
&= 2 \times (3.900\,\text{ms} + 5.300\,\text{ms} + 3.900\,\text{ms}) \\
&= 26.200\,\text{ms}
\end{aligned}$$

while the next 16 accesses are concurrent with those of the remaining Get accesses from Core 1:

$$\begin{aligned}
e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(3..18(\text{Get})) &= e'^{\text{Ebox.Ebox\_Access}}_{\text{Sensor\_Tasks}}(3..18(\text{Get})) \\
&\quad + c^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(3..18(\text{Get})) \\
&= 16 \times (3.900\,\text{ms} + 3.900\,\text{ms}) \\
&= 124.800\,\text{ms}
\end{aligned}$$

and the following 8 accesses are concurrent with the Is_Enabled accesses from Core 1:

$$\begin{aligned}
e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(19..26(\text{Get})) &= e'^{\text{Ebox.Ebox\_Access}}_{\text{Sensor\_Tasks}}(1..8(\text{Is\_Enabled})) \\
&\quad + c^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(19..26(\text{Get})) \\
&= 8 \times (0.381\,\text{ms} + 3.900\,\text{ms}) \\
&= 34.248\,\text{ms}
\end{aligned}$$

the remaining Housekeeping_Task access to the resource do not suffer any spin delay:

$$\begin{aligned}
e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(27..47(\text{Get})) &= c^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(27..47(\text{Get})) \\
&= 21 \times 3.900\,\text{ms} \\
&= 81.900\,\text{ms}
\end{aligned}$$

$$\begin{aligned}
e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1..44(\text{Is\_Enabled})) &= c^{\text{Ebox.Ebox\_Access}}_{\text{WDT\_PSU}}(1..44(\text{Is\_Enabled})) \\
&= 44 \times 0.381\,\text{ms} \\
&= 16.764\,\text{ms}
\end{aligned}$$

$$e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1(\text{Battery\_Warnings})) = c^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1(\text{Battery\_Warnings}))$$
$$= 0.040\,\text{ms}$$

being the total Housekeeping_Task access to the resource:

$$\begin{aligned}
e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}} &= e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1,2(\text{Get})) \\
&\quad + e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(3..18(\text{Get})) \\
&\quad + e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(19..26(\text{Get})) \\
&\quad + e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(27..47(\text{Get})) \\
&\quad + e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1..44(\text{Is\_Enabled})) \\
&\quad + e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1(\text{Battery\_Warnings})) \\
&= 26.200\,\text{ms} + 124.800\,\text{ms} + 34.248\,\text{ms} \\
&\quad + 81.900\,\text{ms} + 16.764\,\text{ms} + 0.040\,\text{ms} \\
&= 283.952\,\text{ms}
\end{aligned}$$

Access to the Mission_Clock via the I2C bus is, as under MSRP, affected by the usage of the resource from Core 0, as can be seen in table 6.15. As $|map(G(\text{I2C.I2C\_Access}))| = 2$, each access can only suffer 1 spin delay from the remote processor. The access cost of the Housekeeping_Task is then:

$$\begin{aligned}
e^{\text{I2C.I2C\_Access}}_{\text{Housekeeping\_Task}}(1(\text{Mission\_Clock})) &= e'^{\text{I2C.I2C\_Acces}}_{\text{TC\_Handler}}(1(\text{Mission\_Clock})) \\
&\quad + c^{\text{I2C.I2C\_Acces}}_{\text{Housekeeping\_Task}}(1(\text{Mission\_Clock})) \\
&= 0.831\,\text{ms} + 0.831\,\text{ms} \\
&= 1.662\,\text{ms}
\end{aligned}$$

Finally, as explained previously, the ACS.MGM_Data shared resource is used to Get the MGM measurements to be logged in the housekeeping data. It is thus accessed by an ACS task on Core 0. Since no nesting is involved, the spin delay is equal to that of MSRP, being the access cost:

$$\begin{aligned}
e^{\text{ACS.MGM\_Data}}_{\text{Housekeeping\_Task}}(1(\text{Get})) &= e'^{\text{ACS.MGM\_Data}}_{\text{Sensor\_Task\_5}}(1(\text{Put})) \\
&\quad + c^{\text{ACS.MGM\_Data}}_{\text{Housekeeping\_Task}}(1(\text{Get})) \\
&= 0.147\,\text{ms} + 0.147\,\text{ms} \\
&= 0.294\,\text{ms}
\end{aligned}$$

The overall shared resources access cost (including direct spin delay) incurred by the Housekeeping_Task under MrsP is presented in table 6.40. The resulting response time, according to equation 5.13, given that the task executes alone on its host processor, is calculated as:

$$\begin{aligned}
R_{\text{Housekeeping\_Task}} &= C_{\text{Housekeeping\_Task}} + E_{\text{Housekeeping\_Task}} \\
&= 89.022\,\text{ms} + 367.369\,\text{ms} \\
&= 459.391\,\text{ms}
\end{aligned}$$

Since the Housekeeping_Task executes alone on its processor, it does not suffer any indirect spin delay or interference due to higher-priority tasks, as thus presents a fairly easy response time calculation.

Table 6.38: UPMSat-2 shared resources local priorities under MrsP with optimized Platform.Configuration priority.

Subsystem	Protected Object	Core 0	Core 1	Core 2
Manager	Commissioning	-	-	8
	Event_Buffer	6	-	14
	Command_Buffer	-	-	14
	Mode	-	-	8
Platform	Configuration (Platform)	6	-	14
	Table	6	-	8
	Ebox_Access	6	13	2
	I2C_Access	6	-	14
	Buffer	-	-	14
ACS	MGM_Data	6	13	-
	MGM_Signals	-	13	-
	MGT	-	12	-
	Parameters	-	12	8
TTC	Visibility_Timer	-	-	14
	Lost_Communications_Timer	-	-	14
	Deferred_Timers	-	-	14
	Configuration (TTC)	-	-	14
	TM_Basic	-	-	9
	TM_Nominal	-	-	9
	Receive_Pool	-	-	10
Storage	Configuration_Parameters	-	-	14
	Storage_Buffers	-	-	14
	EEPROM	-	-	14

Table 6.39: Platform.Table resource usage summary during TC phase under MrsP.

map(G)	Task	Interface	n	WCET	I. Resources
Core 0	Housekeeping_Task	Save	1	31.121 ms	EEPROM
		Set_State	22	0.026 ms	$\emptyset$
		Set_Value	56	0.025 ms	$\emptyset$
Core 2	Manager.Mode	Set_Operating_Mode	1	0.025 ms	$\emptyset$

$map(G)$	$\|map(G)\|$	$\|V(G)\|$	Q
Core 0, Core 2	2	0	2

Table 6.40: Housekeeping_Task resource access cost incurred under MrsP.

Resource	Access Cost
Manager.Event_Buffer	0.753 ms
Platform.Configuration	16.210 ms
Platform.Table	64.498 ms
Ebox.Ebox_Access	283.952 ms
I2C.I2C_Access	1.662 ms
ACS.MGM_Data	0.294 ms
Total	367.369 ms

Core 1

Core 1, as previously presented, hosts the ACS subsystem tasks, which present the resource access pattern shown in table 6.18. As also previously stated, while several tasks are hosted on the same processor, by construction, they do not overlap their activations. As a consequence, they do not cause any interference or blocking to each other. Activations of the Sensor_Tasks and Control are periodic, while the Actutor_Tasks are triggered by the Control completion as well as the length of the PWM to be applied on each axis.

Sensor_Tasks access the Ebox.Ebox_Access resource the measure the magnetic field around the satellite, measuring each axis per MGM (resulting in a total of 9 accesses). Before requesting the measurements, the correct configuration of the MGMs is checked (whether their power supply Is_Enabled or not). The Ebox.Ebox_Access resource details was presented in table 6.41, where it was shown that the resource is not involved in nested calls. The resource, as discussed previously for Core 0 is intensively accessed by the Housekeeping_Task. This causes tasks on Core 1 to have a possible worst-case delay of 13 Get accesses from Core 0 as $N^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(\text{Get}) \gg N^{\text{Ebox.Ebox\_Access}}_{ACS}$. Regarding Core 2, the delay from WDT_PSU is of just 1 Reset_WDT as discussed previously:

$$\begin{aligned}
e^{\text{Ebox.Ebox\_Access}}_{\text{Sensor\_Tasks}}(1(\text{Get})) &= e'^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1(\text{Get})) \\
&\quad + e'^{\text{Ebox.Ebox\_Access}}_{\text{WDT\_PSU}}(1(\text{Reset\_WDT})) \\
&\quad + c^{\text{Ebox.Ebox\_Access}}_{\text{Sensor\_Tasks}}(1(\text{Get})) \\
&= 3.900\,\text{ms} + 5.300\,\text{ms} + 3.900\,\text{ms} \\
&= 13.100\,\text{ms}
\end{aligned}$$

Table 6.41: Ebox.Ebox_Access resource usage during TC phase.

map(G)	Task / Resource	Interface	n	WCET	Inner Resources
Core 0	Housekeeping_Task	Get	47	3.900 ms	∅
		Is_Enabled	44	0.381 ms	∅
		Battery_Warnings	1	0.040 ms	∅
Core 1	Sensor_Tasks (x5)	Get	9	3.900 ms	∅
		Is_Enabled	4	0.381 ms	∅
	PWM_ON	Actuate_MGT	3	0.414 ms	∅
	PWM_OFF (x3)	Actuate_MGT	3	0.414 ms	∅
Core 2	WDT_PSU	Reset_WDT	1	5.300 ms	∅

$map(G)$	$\|map(G)\|$	$\|V(G)\|$	Q
Core 0, Core 1	2	0	2

$$\begin{aligned} e^{\text{Ebox.Ebox\_Access}}_{\text{Sensor\_Tasks}}(2..9(\text{Get})) &= e'^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(2..9(\text{Get})) \\ &\quad + c^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(2..9(\text{Get})) \\ &= 8 \times (3.900\,\text{ms} + 3.900\,\text{ms}) \\ &= 62.400\,\text{ms} \end{aligned}$$

and the Is_Enabled accesses:

$$\begin{aligned} e^{\text{Ebox.Ebox\_Access}}_{\text{Sensor\_Tasks}}(1..4(\text{Is\_Enabled})) &= e'^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(10..13(\text{Get})) \\ &\quad + c^{\text{Ebox.Ebox\_Access}}_{\text{Sensor\_Tasks}}(1..4(\text{Is\_Enabled}) \\ &= 4 \times (3.900\,\text{ms} + 0.381\,\text{ms}) \\ &= 17.124\,\text{ms} \end{aligned}$$

All Sensor_Tasks do also access the ACS.Parameters shared resource on each activation. These accesses can suffer spin delay due to accesses from Core 2 as can be seen in table 6.42:

$$\begin{aligned} e^{\text{ACS.Parameters}}_{\text{Sensor\_Tasks}}(1(\text{Get\_Calibration})) &= e'^{\text{ACS.Parameters}}_{\text{TC\_Handler}}(1(\text{Get\_Calibration})) \\ &\quad + c^{\text{ACS.Parameters}}_{\text{Sensor\_Tasks}}(1(\text{Get\_Calibration})) \\ &= 0.026\,\text{ms} + 0.026\,\text{ms} \\ &= 0.052\,\text{ms} \end{aligned}$$

Table 6.42: ACS.Parameters resource usage during TC phase.

map(G)	Task / Resource	Interface	n	WCET	Inner Resources
Core 1	Sensor_Tasks	Get_Calibration	1	0.026 ms	$\emptyset$
	Control_Task	Get_Control	1	0.026 ms	$\emptyset$
Core 2	TC_Handler	Get_Calibration	1	0.026 ms	$\emptyset$
		Get_Control	1	0.026 ms	$\emptyset$
		Put	1	0.024 ms	$\emptyset$

$map(G)$	$\lvert map(G)\rvert$	$\lvert V(G)\rvert$	Q
Core 0, Core 2	2	0	2

Finally, as mentioned before, the 5th (and last) activation of the sensor task passes, by means of the ACS.MGM_Data, the last measurements to the Housekeeping_Task to be logged and stored to be later sent to the ground station as part of the telemetry data.

$$\begin{aligned}
e^{\text{ACS.MGM\_Data}}_{\text{Sensor\_Task\_5}}(1(\text{Get})) &= e'^{\text{ACS.MGM\_Data}}_{\text{Housekeeping\_Task}}(1(\text{Get})) \\
&\quad + c^{\text{ACS.MGM\_Data}}_{\text{Sensor\_Task\_5}}(1(\text{Put})) \\
&= 0.147\,\text{ms} + 0.147\,\text{ms} \\
&= 0.294\,\text{ms}
\end{aligned}$$

The Control_Task does also access the ACS.Parameters resource to obtain the envisaged control behaviour and algorithm parameters. This access can also be delayed by an access from Core 2:

$$\begin{aligned}
e^{\text{ACS.Parameters}}_{\text{Control\_Task}}(1(\text{Get\_Control})) &= e'^{\text{ACS.Parameters}}_{\text{TC\_Handler}}(1(\text{Get\_Control})) \\
&\quad + c^{\text{ACS.Parameters}}_{\text{Control\_Task}}(1(\text{Get\_Control})) \\
&= 0.026\,\text{ms} + 0.026\,\text{ms} \\
&= 0.052\,\text{ms}
\end{aligned}$$

Finally, the Actuator_Tasks do also access the Ebox.Ebox_Access resource to start and stop the PWM cycle on each access, being potentially

Table 6.43: ACS tasks resource access costs incurred under MrsP.

Tasks	Resource	Access Cost
Sensor_Tasks	Ebox.Ebox_Access	92.624 ms
	ACS.Parameters	0.052 ms
	Total	92.676 ms
Sensor_Task_5	ACS.MGM_Data	+0.294 ms
	Total	92.970 ms
Control_Task	ACS.Parameters	0.052 ms
Actuator_Tasks	Ebox.Ebox_Access	17.120 ms

delayed by the WDT_PSU and Housekeeping_Task worst-case accesses:

$$\begin{aligned}
e^{\text{Ebox.Ebox\_Access}}_{\text{Actuator\_Tasks}}(1(\text{Actuate\_MGT})) &= e'^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1(\text{Get})) \\
&\quad + e'^{\text{Ebox.Ebox\_Access}}_{\text{WDT\_PSU}}(1(\text{Reset\_WDT})) \\
&\quad + c^{\text{Ebox.Ebox\_Access}}_{\text{Actuator\_Tasks}}(1(\text{Actuate\_MGT})) \\
&= 3.900\,\text{ms} + 5.300\,\text{ms} + 0.040\,\text{ms} \\
&= 9.240\,\text{ms}
\end{aligned}$$

$$\begin{aligned}
e^{\text{Ebox.Ebox\_Access}}_{\text{Actuator\_Tasks}}(2,3(\text{Actuate\_MGT})) &= e'^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(2,3(\text{Get})) \\
&\quad + c^{\text{Ebox.Ebox\_Access}}_{\text{Actuator\_Tasks}}(2,3(\text{Actuate\_MGT})) \\
&= 2 \times (3.900\,\text{ms} + 0.040\,\text{ms}) \\
&= 7.880\,\text{ms}
\end{aligned}$$

Table 6.43 summarizes the ACS tasks access costs to shared resources under MrsP.

Core 2

As for MSRP, the analysis of Core 2 is the most complex one. As it hosts a set of tasks that can be runnable at the same time, they can cause interference and arrival blocking to each other. Furthermore, as presented in section 5.3.1, the new response time analysis equation 5.13 can be used to reduce the pessimism in the analysis by limiting the effects in the indirect spin delay caused by higher-priority tasks.

While the Visibility_Timer handler does not access any shared resource, the Lost_Comm_Timer handler can access the Manager.Event_Buffer, as the loss of communications can trigger a mode change (later evaluated by the EV_Handler). This access can, as shown in table 6.35, cause direct spin delay equal to one access cost:

$$\begin{aligned} e_{\text{Lost\_Comm\_Timer}}^{\text{Manager.Event\_Buffer}}(1(\text{Put})) &= e'^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(1(\text{Put})) \\ &\quad + c_{\text{Lost\_Comm\_Timer}}^{\text{Manager.Event\_Buffer}}(1(\text{Put})) \\ &= 0.026\,\text{ms} + 0.026\,\text{ms} \\ &= 0.052\,\text{ms} \end{aligned}$$

The Deferred_TC_Timer handler presents a more complex analysis, as was shown for MSRP. The handler accesses two shared resources that include nested calls to a globally shared resource: the EEPROM memory. As shown before, this resource is also accessed by the Housekeeping_Task on Core 0. The TTC.Configuration access cost can calculated as follows:

$$\begin{aligned} e_{\text{Deferred\_TC\_Timer}}^{\text{TTC.Configuration}}(1(\text{Save})) &= c_{\text{Deferred\_TC\_Timer}}^{\text{TTC.Configuration}}(1(\text{Save})) \\ &\quad + e_{\text{Deferred\_TC\_Timer}}^{\text{EEPROM}}(1(\text{Write})) \end{aligned}$$

for what we need to calculate the EEPROM memory access cost, that can suffer spin delay due to the Housekeeping_Task. Since it is a terminal resource (makes no other inner resource access) and the Housekeeping_Task cannot be locally preempted, the worst-case delay is equal to its worst access time:

$$\begin{aligned} e_{\text{Deferred\_TC\_Timer}}^{\text{EEPROM}}(1(\text{Write})) &= c'^{\text{EEPROM}}_{\text{Housekeeping\_Task}}(1(\text{Write})) \\ &\quad + c_{\text{Deferred\_TC\_Timer}}^{\text{EEPROM}}(1(\text{Write})) \\ &= 15.500\,\text{ms} + 15.500\,\text{ms} \\ &= 31.000\,\text{ms} \end{aligned}$$

then we can substitute to calculate $e^{\text{Manager.Event\_Buffer}}_{\text{Lost\_Comm\_Timer}}(1(\text{Put}))$:

$$e^{\text{TTC.Configuration}}_{\text{Deferred\_TC\_Timer}}(1(\text{Save})) = 0.783\,\text{ms} + 31.000\,\text{ms} = 31.783\,\text{ms}$$

In the same way, the access to the Storage.TC_Storage_Buffer can suffer spin delay on its access to the EEPROM memory is calculated as:

$$\begin{aligned}
e^{\text{TTC.Configuration}}_{\text{Deferred\_TC\_Timer}}(1(\text{Save})) &= c^{\text{TTC.Configuration}}_{\text{Deferred\_TC\_Timer}}(1(\text{Save})) \\
&\quad + e^{\text{EEPROM}}_{\text{Deferred\_TC\_Timer}}(1(\text{Read}))
\end{aligned}$$

$$\begin{aligned}
e^{\text{EEPROM}}_{\text{Deferred\_TC\_Timer}}(1(\text{Read})) &= c'^{\text{EEPROM}}_{\text{Housekeeping\_Task}}(2(\text{Write})) \\
&\quad + c^{\text{EEPROM}}_{\text{Deferred\_TC\_Timer}}(1(\text{Read})) \\
&= 15.500\,\text{ms} + 0.035\,\text{ms} \\
&= 15.535\,\text{ms}
\end{aligned}$$

$$\begin{aligned}
e^{\text{TTC.Configuration}}_{\text{Deferred\_TC\_Timer}}(1(\text{Save})) &= c^{\text{TTC.Configuration}}_{\text{Deferred\_TC\_Timer}}(1(\text{Save})) \\
&\quad + e^{\text{EEPROM}}_{\text{Deferred\_TC\_Timer}}(1(\text{Read})) \\
&= 15.535\,\text{ms} + 0.513\,\text{ms} \\
&= 16.048\,\text{ms}
\end{aligned}$$

The Radio_Listener only accesses local shared resources, which are not of particular interest, since they follow the traditional monocore ceiling priority analysis. The TC_Receiver_Task, on the contrary, does access a globally shared resource, as can be seen in table 6.25. This shared resource is, as in the Lost_Comm_Timer case, the Manager.Event_Buffer. As a higher-priority task has already accessed the resource, the mechanisms presented in section 5.3.1 to reduce the pessimism in the analysis can be attempted. According to equation 5.20, the direct spin delay to be accounted for per processor ($\text{NS}^k_{x,m}(l)$) is the result of subtracting the already accounted for accesses of local higher-priority tasks ($\text{Nh}^k_x(l)$) in equation 5.19) to the number of accesses that can be issued from the analysed processor ($\text{Np}^k_m(l)$) in equation 5.18. The remote accesses in this case are from Core 0, due to Housekeeping_Task accesses:

$$\begin{aligned}
&\mathrm{Np}^{\text{Manager.Event\_Buffer}}_{Core0}(R_{\text{TC\_Receiver\_Task}}) \\
&\quad = \mathrm{N}^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}}(R_{\text{TC\_Receiver\_Task}}, R_{\text{Housekeeping\_Task}}) \\
&\quad = \left\lceil \frac{R_{\text{TC\_Receiver\_Task}} + R_{\text{Housekeeping\_Task}}}{T_{\text{Housekeeping\_Task}}} \right\rceil \cdot \mathrm{N}^{\text{Manager.Event\_Buffer}}_{\text{Housekeeping\_Task}} \\
&\quad = \left\lceil \frac{71.742\,\text{ms} + 324.700\,\text{ms}}{1000\,\text{ms}} \right\rceil \cdot 23 \\
&\quad = 23
\end{aligned}$$

Tasks with a higher priority than TC_Receiver_Task accessing the Manager .Event_Buffer only include the aforementioned Lost_Comm_Timer, for which only one access has been accounted for:

$$\begin{aligned}
&\mathrm{Nh}^{\text{Manager.Event\_Buffer}}_{\text{TC\_Receiver\_Task}}(R_{\text{TC\_Receiver\_Task}}) \\
&\quad = \mathrm{N}^{\text{Manager.Event\_Buffer}}_{\text{Lost\_Comm\_Timer}}(R_{\text{TC\_Receiver\_Task}}, R_{\text{Lost\_Comm\_Timer}}) \\
&\quad = \left\lceil \frac{R_{\text{TC\_Receiver\_Task}} + R_{\text{Lost\_Comm\_Timer}}}{T_{\text{Lost\_Comm\_Timer}}} \right\rceil \cdot \mathrm{N}^{\text{Manager.Event\_Buffer}}_{\text{Lost\_Comm\_Timer}} \\
&\quad = \left\lceil \frac{71.742\,\text{ms} + 33.693\,\text{ms}}{1.73\text{E+08}} \right\rceil \cdot 1 \\
&\quad = 1
\end{aligned}$$

The spin delay not already accounted for is then:

$$\begin{aligned}
&\mathrm{NS}^{\text{Manager.Event\_Buffer}}_{\text{TC\_Receiver\_Task,Core 0}}(R_{\text{TC\_Receiver\_Task}}) \\
&\qquad = (\mathrm{Np}^{\text{Manager.Event\_Buffer}}_{\text{Core 0}}(R_{\text{TC\_Receiver\_Task}}) \\
&\qquad\quad - \mathrm{Nh}^{\text{Manager.Event\_Buffer}}_{\text{TC\_Receiver\_Task}}(R_{\text{TC\_Receiver\_Task}}))_0 \\
&\qquad = 23 - 1 \\
&\qquad = 22
\end{aligned}$$

As the number of potential remote accesses is bigger than those already accounted for, there is still direct spin delay to be added to

TC_Receiver_Task accesses to the Manager.Event_Buffer. As can be seen in table 6.25, it only accesses once the resource with the following cost:

$$
\begin{aligned}
&e^{\text{Manager.Event\_Buffer}}_{\text{TC\_Receiver\_Task}}(R_{\text{TC\_Receiver\_Task}})(1)\\
&\quad= (\text{NS}^{\text{Manager.Event\_Buffer}}_{\text{TC\_Receiver\_Task,Core 0}}(R_{\text{TC\_Receiver\_Task}}) - 1 + 1)^{1}_{0} \cdot 0.026\,\text{ms}\\
&\qquad + 0.026\,\text{ms}\\
&\quad= (22 + 1 - 1)^{1}_{0} \cdot 0.026\,\text{ms} + 0.026\,\text{ms}\\
&\quad= 1 \cdot 0.026\,\text{ms} + 0.026\,\text{ms}\\
&\quad= 0.052\,\text{ms}
\end{aligned}
$$

The TC_Receiver_Task does also access a local shared resource that performs a nested access to a globally shared resource: the TTC. Deferred_Timer Schedule interface accesses twice the EEPROM memory to both store the TC to be later executed and to update the TTC configuration. As a higher-priority task has already suffered spin delay from the EEPROM memory we can apply again equations 5.18, 5.19 and 5.20:

$$
\begin{aligned}
&\text{Np}^{\text{EEPROM}}_{\text{Core 0}}(R_{\text{TC\_Receiver\_Task}})\\
&\quad= \text{N}^{\text{EEPROM}}_{\text{Housekeeping\_Task}}(R_{\text{TC\_Receiver\_Task}}, R_{\text{Housekeeping\_Task}})\\
&\quad= \left\lceil \frac{R_{\text{TC\_Receiver\_Task}} + R_{\text{Housekeeping\_Task}}}{T_{\text{Housekeeping\_Task}}} \right\rceil \cdot \text{N}^{\text{EEPROM}}_{\text{Housekeeping\_Task}}\\
&\quad= \left\lceil \frac{71.742\,\text{ms} + 324.700\,\text{ms}}{1000\,\text{ms}} \right\rceil \cdot 2\\
&\quad= 2
\end{aligned}
$$

$$\begin{aligned}
&\mathrm{Nh}^{\mathrm{EEPROM}}_{\mathrm{TC\_Receiver\_Task}}(R_{\mathrm{TC\_Receiver\_Task}})\\
&\quad= \mathrm{N}^{\mathrm{EEPROM}}_{\mathrm{Deferred\_TC\_Timer}}(R_{\mathrm{TC\_Receiver\_Task}}, R_{\mathrm{Deferred\_TC\_Timer}})\\
&\quad= \left\lceil \frac{R_{\mathrm{TC\_Receiver\_Task}} + R_{\mathrm{Deferred\_TC\_Timer}}}{T_{\mathrm{Deferred\_TC\_Timer}}} \right\rceil \cdot \mathrm{N}^{\mathrm{Manager.Event\_Buffer}}_{\mathrm{Lost\_Comm\_Timer}}\\
&\quad= \left\lceil \frac{71.742\,\mathrm{ms} + 33.693\,\mathrm{ms}}{1.73E+08\,\mathrm{ms}} \right\rceil \cdot 2\\
&\quad= 2
\end{aligned}$$

$$\begin{aligned}
\mathrm{NS}^{\mathrm{EEPROM}}_{\mathrm{TC\_Receiver\_Task},Core0}(R_{\mathrm{TC\_Receiver\_Task}})
&= (\mathrm{Np}^{\mathrm{EEPROM}}_{Core0}(R_{\mathrm{TC\_Receiver\_Task}})\\
&\quad - \mathrm{Nh}^{\mathrm{EEPROM}}_{\mathrm{TC\_Receiver\_Task}}(R_{\mathrm{TC\_Receiver\_Task}}))_0\\
&= 2 - 2\\
&= 0
\end{aligned}$$

This means that all the potential spin delay suffered on Core 2 during TC_Receiver_Task execution due to accessing the EEPROM memory has already been accounted for on higher-priority tasks. As a result, the TC_Receiver_Task does not have to account for any extra direct spin delay due to the EEPROM memory, i.e. the TTC.Deferred_Timer Schedule access cost is equal to its access time of 36.512 ms. The Housekeeping_Task induced spin delay will, however, be part of the indirect spin delay (interference).

The indirect spin delay is the access costs of higher-priority tasks to globally shared resources. This takes into account both the local and remote accesses. Since lower-priority tasks tend to have longer periods than higher-priority tasks, the indirect spin delay caused by a higher-priority task does not need to be equal to its direct spin delay times the number of activations during a lower-priority task activation. This is because in such a longer period it might not have as much shared resource contention as in a worst-case activation. Highest-priority tasks of TC_Receiver_Task accessing globally shared resource accesses are the Lost_Comm_Timer and Deferred_TC_Timer.

As calculated before for the $\text{Nh}_{\text{TC\_Receiver\_Task}}^{\text{Manager.Event\_Buffer}}(R_{\text{TC\_Receiver\_Task}})$, each activation can only overlap with at most one Lost_Comm_Timer activation. The induced indirect spin delay from TC_Receiver_Task is:

$$\begin{aligned}
&\text{I}_{\text{TC\_Receiver\_Task,Lost\_Comm\_Timer}}\\
&\quad = e_{\text{Lost\_Comm\_Timer}}^{\text{Manager.Event\_Buffer}}(R_{\text{TC\_Receiver\_Task}}, R_{\text{Lost\_Comm\_Timer}})\\
&\quad = e_{\text{Lost\_Comm\_Timer}}^{\text{Manager.Event\_Buffer}}(1(\text{Put}))\\
&\quad = 0.052\,\text{ms}
\end{aligned}$$

that equals the task $E_{\text{TC\_Receiver\_Task}}$, as both tasks only overlap once.

The Deferred_TC_Timer adds indirect spin delay due to its three globally shared resource accesses:

$$\begin{aligned}
&\text{I}_{\text{TC\_Receiver\_Task,Deferred\_TC\_Timer}}\\
&\quad = e_{\text{Deferred\_TC\_Timer}}^{\text{Manager.Command\_Buffer}}(R_{\text{TC\_Receiver\_Task}}, R_{\text{Deferred\_TC\_Timer}})\\
&\qquad + e_{\text{Deferred\_TC\_Timer}}^{\text{TTC.Configuration}}(R_{\text{TC\_Receiver\_Task}}, R_{\text{Deferred\_TC\_Timer}})\\
&\qquad + e_{\text{Deferred\_TC\_Timer}}^{\text{Storage.TC\_Storage\_Buffer}}(R_{\text{TC\_Receiver\_Task}}, R_{\text{Deferred\_TC\_Timer}})\\
&\quad = 0.026\,\text{ms} + 31.783\,\text{ms} + 16.048\,\text{ms}\\
&\quad = 47.857\,\text{ms}
\end{aligned}$$

Equation 5.13 is to be used to calculate the response time of TC_Receiver_Task (recall that the TC_Receiver_Task is activated at the Radio_Listener completion):

$$
\begin{aligned}
R_{\text{TC\_Receiver\_Task}} &= C_i + E_i + B_i + \sum_{\tau_h \in hpl(i)} \left(\left\lceil \frac{R_i}{T_h} \right\rceil \cdot C_h + I_{i,h} \right) \\
&= 1.187\,\text{ms} + 36.888\,\text{ms} + 15.585\,\text{ms} \\
&\quad + \left\lceil \frac{71.742\,\text{ms}}{5.82\text{E}{+}06\,\text{ms}} \right\rceil \cdot 0.400\,\text{ms} + 0.000\,\text{ms} \\
&\quad + \left\lceil \frac{71.742\,\text{ms}}{1.73\text{E}{+}08\,\text{ms}} \right\rceil \cdot 0.200\,\text{ms} + 0.052\,\text{ms} \\
&\quad + \left\lceil \frac{71.742\,\text{ms}}{2000\,\text{ms}} \right\rceil \cdot 0.625\,\text{ms} + 47.857\,\text{ms} \\
&= 1.187\,\text{ms} + 36.888\,\text{ms} + 15.585\,\text{ms} + 0.400\,\text{ms} \\
&\quad + 0.252\,\text{ms} + 48.482\,\text{ms} \\
&= 103.492\,\text{ms}
\end{aligned}
$$

The TC_Handler resource accesses were presented in table 6.26 and worst-case accesses discussed during the MSRP analysis. This includes an access to the I2C.I2C_Access, ACS.Parameters and Manager.Mode globally shared resources.

The I2C.I2C_Access shared resource is not accessed by a higher-priority task and, as a result, can cause direct spin delay to the TC_Handler. Since it is not involved in any nesting, the access cost is:

$$
\begin{aligned}
&e^{\text{I2C.I2C\_Access}}_{\text{TC\_Handler}}(1(\text{Mission\_Clock})) \\
&\qquad = c^{\text{I2C.I2C\_Access}}_{\text{TC\_Handler}}(1(\text{Mission\_Clock})) \\
&\qquad\quad + e'^{\text{I2C.I2C\_Access}}_{\text{Housekeeping\_Task}}(1(\text{Mission\_Clock})) \\
&\qquad = 0.831\,\text{ms} + 0.831\,\text{ms} \\
&\qquad = 1.662\,\text{ms}
\end{aligned}
$$

The ACS.Parameters is not accessed by higher-priority task, either.

Since for the fist iteration analysis

$$\mathrm{N}^{\text{Ebox.Ebox\_Access}}_{\text{Sensor\_Tasks}}(R_{\text{TC\_Handler}}) = \left\lceil \frac{R_{\text{ACS.Parameters}} + R_{\text{TC\_Handler}}}{\text{Offset}_{\text{Sensor\_Tasks}}} \right\rceil \cdot 1 = 1$$

the first access suffers one spin delay due to Sensor_Tasks:

$$\begin{aligned} e^{\text{ACS.Parameters}}_{\text{TC\_Handler}}(1) &= c^{\text{ACS.Parameters}}_{\text{TC\_Handler}}(1) + +e'^{\text{ACS.Parameters}}_{\text{Sensor\_Tasks}}(1) \\ &= 0.026\,\text{ms} + 0.026\,\text{ms} \\ &= 0.052\,\text{ms} \end{aligned}$$

and accesses 2 and 3 are done contention free:

$$e^{\text{ACS.Parameters}}_{\text{TC\_Handler}}(2,3) = c^{\text{ACS.Parameters}}_{\text{TC\_Handler}}(2,3) = 2 \times 0.026\,\text{ms} = 0.052\,\text{ms}$$

The Manager.Mode, as was explained for the MSRP analysis, includes 2 nested accesses to the EEPROM memory via the Platform.Table access. This resource has been already analysed for tasks allocated to Core 2. While for the Deferred_TC_Timer two accesses from the Housekeeping_Task were added to the direct spin delay, the TC_Receiver_Task was not added this direct spin delay. In the same way, the TC_Handler spin delay due to the EEPROM memory access is also accounted as indirect spin delay coming from the Deferred_TC_Timer, as $\mathrm{NS}^{\text{TC\_Handler}}_{\text{EEPROM}} = 0$:

$$\begin{aligned} \mathrm{Np}^{\text{EEPROM}}_{Core0}(R_{\text{TC\_Handler}}) &= \mathrm{N}^{\text{EEPROM}}_{\text{Housekeeping\_Task}}(R_{\text{TC\_Handler}}, R_{\text{Housekeeping\_Task}}) \\ &= 2 \end{aligned}$$

$$\begin{aligned} \mathrm{Nh}^{\text{EEPROM}}_{\text{TC\_Handler}}(R_{\text{TC\_Handler}}) &= \mathrm{N}^{\text{EEPROM}}_{\text{Deferred\_TC\_Timer}}(R_{\text{TC\_Handler}}, R_{\text{Deferred\_TC\_Timer}}) \\ &\quad + \mathrm{N}^{\text{EEPROM}}_{\text{TC\_Receiver\_Task}}(R_{\text{TC\_Handler}}, R_{\text{Deferred\_TC\_Timer}}) \\ &= 2 + 2 \\ &= 4 \end{aligned}$$

$$\begin{aligned}
\mathrm{NS}^{\mathrm{EEPROM}}_{\mathrm{TC\_Handler,Core\ 0}}(R_{\mathrm{TC\_Handler}}) &= (\mathrm{Np}^{\mathrm{EEPROM}}_{\mathrm{Core\ 0}}(R_{\mathrm{TC\_Handler}}) \\
&\quad - \mathrm{Nh}^{\mathrm{EEPROM}}_{\mathrm{TC\_Handler}}(R_{\mathrm{TC\_Handler}}))_0 \\
&= (2-4)_0 \\
&= 0
\end{aligned}$$

The response time analysis of the TC_Handler is then:

$$\begin{aligned}
R_{\mathrm{TC\_Handler}} &= C_i + E_i + B_i + \sum_{\tau_h \in hpl(i)} \left(\left\lceil \frac{R_i}{T_h} \right\rceil \cdot C_h + I_{i,h} \right) \\
&= 0.582\,\mathrm{ms} + 37.333\,\mathrm{ms} + 35.121\,\mathrm{ms} \\
&\quad + \left\lceil \frac{130.468\,\mathrm{ms}}{5.82\mathrm{E}{+}06\,\mathrm{ms}} \right\rceil \cdot 0.400\,\mathrm{ms} + 0.000\,\mathrm{ms} \\
&\quad + \left\lceil \frac{130.468\,\mathrm{ms}}{1.73\mathrm{E}{+}08\,\mathrm{ms}} \right\rceil \cdot 0.200\,\mathrm{ms} + 0.052\,\mathrm{ms} \\
&\quad + \left\lceil \frac{130.468\,\mathrm{ms}}{2000\,\mathrm{ms}} \right\rceil \cdot 0.625\,\mathrm{ms} + 47.857\,\mathrm{ms} \\
&\quad + \left\lceil \frac{130.468\,\mathrm{ms}}{2000\,\mathrm{ms}} \right\rceil \cdot 1.464\,\mathrm{ms} + 1.166\,\mathrm{ms} \\
&\quad + \left\lceil \frac{130.468\,\mathrm{ms}}{1932\,\mathrm{ms}} \right\rceil \cdot 1.187\,\mathrm{ms} + 36.888\,\mathrm{ms} \\
&= 0.582\,\mathrm{ms} + 37.333\,\mathrm{ms} + 35.121\,\mathrm{ms} + 0.400\,\mathrm{ms} \\
&\quad + 0.252\,\mathrm{ms} + 48.482\,\mathrm{ms} + 2.630\,\mathrm{ms} + 38.075\,\mathrm{ms} \\
&= 162.875\,\mathrm{ms}
\end{aligned}$$

Regarding the Event_Handler, the only two globally shared resources that can cause any direct spin delay are the Manager.Event_Buffer and the Manager.Mode. Concerning the Manager.Event_Buffer analysis, the number of remote accesses by the Housekeeping_Task is greater than that of the

Event_Handler higher-priority tasks:

$$\begin{aligned}
&e^{\text{Manager.Event\_Buffer}}_{\text{Event\_Handler}}(R_{\text{Event\_Handler}})(1)\\
&\quad = (\text{NS}^{\text{Manager.Event\_Buffer}}_{\text{Event\_Handler},Core0}(R_{\text{Event\_Handler}}) - 1 + 1)^1_0 \cdot 0.026\,\text{ms} + 0.026\,\text{ms}\\
&\quad = (21 + 1 - 1)^1_0 \cdot 0.026\,\text{ms} + 0.026\,\text{ms}\\
&\quad = 1 \cdot 0.026\,\text{ms} + 0.026\,\text{ms}\\
&\quad = 0.052\,\text{ms}
\end{aligned}$$

The access to the Manager.Mode, as mentioned before, can cause direct spin delay due to the access to the inner Platform.Table resource. Same as in the MSRP case, the access from the Event_Handler has to account the second worst access from the Housekeeping_Task as spin delay:

$$\begin{aligned}
e^{\text{Platform.Table}}_{\text{Event\_Handler}}(1) &= c^{\text{Platform.Table}}_{\text{Event\_Handler}}(1)\\
&\quad + c^{\text{Platform.Table}}_{\text{Housekeeping\_Task}}(2)\\
&= 35.121\,\text{ms} + 0.025\,\text{ms}\\
&= 35.146\,\text{ms}
\end{aligned}$$

The rest of the Event_Handler shared resource accesses costs equal to its access times (due to higher-priority tasks accesses), $E_{\text{Event\_Handler}} = 66.391\,\text{ms}$. Its arrival blocking, however, becomes notably lower since its own access to the Manager.Mode Handle is the one driving that of the higher-priority task TC_Handler:

$$\begin{aligned}
R_{\text{Event\_Handler}} &= C_i + E_i + B_i + \sum_{\tau_h \in hpl(i)} \left(\left\lceil \frac{R_i}{T_h} \right\rceil \cdot C_h + I_{i,h} \right) \\
&= 0.160\,\text{ms} + 66.391\,\text{ms} + 5.300\,\text{ms} \\
&\quad + \left\lceil \frac{161.847\,\text{ms}}{5.82\text{E}{+}06\,\text{ms}} \right\rceil \cdot 0.400\,\text{ms} + 0.000\,\text{ms} \\
&\quad + \left\lceil \frac{161.847\,\text{ms}}{1.73\text{E}{+}08\,\text{ms}} \right\rceil \cdot 0.200\,\text{ms} + 0.052\,\text{ms} \\
&\quad + \left\lceil \frac{161.847\,\text{ms}}{2000\,\text{ms}} \right\rceil \cdot 0.625\,\text{ms} + 47.857\,\text{ms} \\
&\quad + \left\lceil \frac{161.847\,\text{ms}}{2000\,\text{ms}} \right\rceil \cdot 1.464\,\text{ms} + 1.166\,\text{ms} \\
&\quad + \left\lceil \frac{161.847\,\text{ms}}{1932\,\text{ms}} \right\rceil \cdot 1.187\,\text{ms} + 36.888\,\text{ms} \\
&\quad + \left\lceil \frac{161.847\,\text{ms}}{1897\,\text{ms}} \right\rceil \cdot 0.582\,\text{ms} + 37.333\,\text{ms} \\
&= 0.160\,\text{ms} + 66.391\,\text{ms} + 5.300\,\text{ms} + 0.400\,\text{ms} \\
&\quad + 0.252\,\text{ms} + 48.482\,\text{ms} + 2.630\,\text{ms} + 38.075\,\text{ms} \\
&\quad + 37.915\,\text{ms} \\
&= 194.305\,\text{ms}
\end{aligned}$$

The WDT_PSU watchdog timer accesses the Ebox.Ebox_Access resource. Its access cost is dependant of the remote accesses of the Houskeeping_Task and ACS Sensor_Tasks:

$$\begin{aligned}
e^{\text{Ebox.Ebox\_Access}}_{\text{WDT\_PSU}}(1(\text{Reset\_WDT})) &= e'^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1(\text{Get})) \\
&\quad + e'^{\text{Ebox.Ebox\_Access}}_{\text{Sensor\_Tasks}}(1(\text{Get})) \\
&\quad + c^{\text{Ebox.Ebox\_Access}}_{\text{WDT\_PSU}}(1(\text{Reset\_WDT})) \\
&= 3.900\,\text{ms} + 3.900\,\text{ms} + 5.300\,\text{ms} \\
&= 13.100\,\text{ms}
\end{aligned}$$

Its response time follows a similar form as that of the previously analysed tasks:

$$
\begin{aligned}
R_{\text{WDT\_PSU}} &= C_i + E_i + B_i + \sum_{\tau_h \in hpl(i)} \left(\left\lceil \frac{R_i}{T_h} \right\rceil \cdot C_h + I_{i,h} \right) \\
&= 0.200\,\text{ms} + 13.100\,\text{ms} + 0.000\,\text{ms} \\
&\quad + \left\lceil \frac{167.347\,\text{ms}}{5.82\text{E+06}\,\text{ms}} \right\rceil \cdot 0.400\,\text{ms} + 0.000\,\text{ms} \\
&\quad + \left\lceil \frac{167.347\,\text{ms}}{1.73\text{E+08}\,\text{ms}} \right\rceil \cdot 0.200\,\text{ms} + 0.052\,\text{ms} \\
&\quad + \left\lceil \frac{167.347\,\text{ms}}{2000\,\text{ms}} \right\rceil \cdot 0.625\,\text{ms} + 47.857\,\text{ms} \\
&\quad + \left\lceil \frac{167.347\,\text{ms}}{2000\,\text{ms}} \right\rceil \cdot 1.464\,\text{ms} + 1.166\,\text{ms} \\
&\quad + \left\lceil \frac{167.347\,\text{ms}}{1932\,\text{ms}} \right\rceil \cdot 1.187\,\text{ms} + 36.888\,\text{ms} \\
&\quad + \left\lceil \frac{167.347\,\text{ms}}{1897\,\text{ms}} \right\rceil \cdot 0.582\,\text{ms} + 37.333\,\text{ms} \\
&\quad + \left\lceil \frac{167.347\,\text{ms}}{2000\,\text{ms}} \right\rceil \cdot 0.160\,\text{ms} + 66.391\,\text{ms} \\
&= 0.200\,\text{ms} + 13.100\,\text{ms} + 0.000\,\text{ms} + 0.400\,\text{ms} \\
&\quad + 0.252\,\text{ms} + 48.482\,\text{ms} + 2.630\,\text{ms} + 38.075\,\text{ms} \\
&\quad + 37.915\,\text{ms} + 66.526\,\text{ms} \\
&= 207.605\,\text{ms}
\end{aligned}
$$

As the response time is not greater than any higher-priority task period no reiteration is required.

Finally, the WDT_FPGA analysis, while not accessing any shared resource, presents the particularity of requiring a reiteration on the response time calculation:

$$\begin{aligned}
R_{\text{WDT\_FPGA}} &= C_i + E_i + B_i + \sum_{\tau_h \in hpl(i)} (\left\lceil \frac{R_i}{T_h} \right\rceil \cdot C_h + I_{i,h}) \\
&= 0.018\,\text{ms} + 0.000\,\text{ms} + 0.000\,\text{ms} \\
&\quad + \left\lceil \frac{207.598\,\text{ms}}{5.82\text{E+}06\,\text{ms}} \right\rceil \cdot 0.400\,\text{ms} + 0.000\,\text{ms} \\
&\quad + \left\lceil \frac{207.598\,\text{ms}}{1.73\text{E+}08\,\text{ms}} \right\rceil \cdot 0.200\,\text{ms} + 0.052\,\text{ms} \\
&\quad + \left\lceil \frac{207.598\,\text{ms}}{2000\,\text{ms}} \right\rceil \cdot 0.625\,\text{ms} + 47.857\,\text{ms} \\
&\quad + \left\lceil \frac{207.598\,\text{ms}}{2000\,\text{ms}} \right\rceil \cdot 1.464\,\text{ms} + 1.166\,\text{ms} \\
&\quad + \left\lceil \frac{207.598\,\text{ms}}{1932\,\text{ms}} \right\rceil \cdot 1.187\,\text{ms} + 36.888\,\text{ms} \\
&\quad + \left\lceil \frac{207.598\,\text{ms}}{1897\,\text{ms}} \right\rceil \cdot 0.582\,\text{ms} + 37.333\,\text{ms} \\
&\quad + \left\lceil \frac{207.598\,\text{ms}}{2000\,\text{ms}} \right\rceil \cdot 0.160\,\text{ms} + 66.391\,\text{ms} \\
&\quad + \left\lceil \frac{207.598\,\text{ms}}{250\,\text{ms}} \right\rceil \cdot 0.200\,\text{ms} + 13.100\,\text{ms} \\
&= 0.018\,\text{ms} + 0.000\,\text{ms} + 0.000\,\text{ms} + 0.400\,\text{ms} \\
&\quad + 0.252\,\text{ms} + 48.482\,\text{ms} + 2.630\,\text{ms} + 38.075\,\text{ms} \\
&\quad + 37.915\,\text{ms} + 66.526\,\text{ms} + 13.300\,\text{ms} \\
&= 247.856\,\text{ms}
\end{aligned}$$

Arrival blocking analysis

The difference in performance between MSRP and MrsP comes from the higher arrival blocking under MSRP due to the non-preemptive access to shared resources. Under MrsP ceiling locking access applies providing better results for higher-priority tasks.

For the case study concerns, the difference between MSRP and MrsP analysis is in the 5 highest-priority entities executing on Core 2. Their arrival blocking is result of the TC_Handler, with priority 8, accessing TTC.Configuration Save with a ceiling priority of 14 and an access time of 16.283 ms.

While the TTC.Configuration resource is only locally accessed by tasks allocated on Core 2, it includes, under the Save interface, a nested call to the Storage.Configuration_Parameters, that also accesses the EEPROM memory for a Write access. As has been explained along the MrsP analysis, this access can potentially cause spin delay. However, the arrival blocking accounted for is of just 16.283 ms, i.e. the TTC. Configuration Save access time. Why does it not include any spin delay then?

Lets us analyse the arrival blocking from the TC_Receiver_Task perspective, which is the first task whose worst-case arrival blocking is that of the TTC.Configuration Save access. First, the set of shared resources that can cause arrival blocking are to be identified as shown in equation 5.23. On that equation, the set of resources that have a local ceiling priority equal or higher than the analysed task and are accessed by lower-priority tasks are identified. For the TC_Receiver_Task, the F^A set includes the resources listed in table 6.44. From them, the access with a higher-access time is the mentioned TTC.Configuration Save. Next step is to calculate the spin delay the blocker task can suffer. This is done using equations 5.24 and 5.25 for non-nested shared resources and equations 5.40 and 5.42 for shared resources involved in nesting. Since the TTC.Configuration Save access is involved in nested accesses, the latter equations are to be used. Then, the arrival blocking of TC_Receiver_Task would be equal to its following access to

Table 6.44: Set of resources that can cause TC_Receiver_Task arrival blocking.

Subsystem	Protected Object	Interface	Access Cost
Manager	Event_Buffer	Put	0.025 ms
	Command_Buffer	Put	0.025 ms
Platform	Configuration	Enable/Disable/Set	15.660 ms
	I2C_Access	Mission_Clock	0.831 ms
TTC	Visibility_Timer	Reset	0.077 ms
	Lost_Comm_Timer	Reset	0.077 ms
	Deferred_Timer	Schedule	4.621 ms
	Configuration	Save	16.283 ms
Storage	Config_Parameters	Update_TTC_Config	15.808 ms
	Storage_Buffers	Put	15.608 ms
	EEPROM	Write	15.500 ms

TTC.Configuration Save:

$$\begin{aligned}&\mathrm{B}_{\text{TC\_Receiver\_Task}}\\&\quad= e_{\widehat{\text{TC\_Receiver\_Task}}}\\&\quad= e^{\text{TTC.Configuration}}_{\text{TC\_Receiver\_Task}}(R_{\text{TC\_Receiver\_Task}})(N^{\text{TTC.Configuration}}_{\text{TC\_Receiver\_Task}}+1)\end{aligned}$$

Since the TC_Receiver_Task was already accounted for one access,

$$\mathrm{N}^{\text{TC\_Receiver\_Task}}_{\text{TTC.Configuration}}+1=2$$

As shown before, as the resource is not accessed by any other resource or remotely, the access cost is calculated as follows:

$$\begin{aligned}&e^{\text{TTC.Configuration}}_{\text{TC\_Receiver\_Task}}(R_{\text{TC\_Receiver\_Task}})(2(\text{Save}))\\&\quad= c^{\text{TTC.Configuration}}_{\text{TC\_Receiver\_Task}}(2(\text{Save}))\\&\qquad+ e^{\text{Storage.Configuration\_Parameters}}_{\text{TC\_Receiver\_Task}}(R_{\text{TC\_Receiver\_Task}})(2(\text{Update}))\end{aligned}$$

Similarly, the Storage.Configuration_Parameters is not either accessed remotely (during the TC phase) or from outer resources, so its access cost is:

$$
\begin{aligned}
&e^{\text{Storage.Configuration\_Parameters}}_{\text{TC\_Receiver\_Task}}(R_{\text{TC\_Receiver\_Task}})(2(\text{Update}))\\
&\quad = c^{\text{Storage.Configuration\_Parameters}}_{\text{TC\_Receiver\_Task}}(2(\text{Update})\\
&\quad\quad + e^{\text{EEPROM}}_{\text{TC\_Receiver\_Task}}(R_{\text{TC\_Receiver\_Task}})(2(\text{Write}))
\end{aligned}
$$

As the EEPROM memory is an inner resource and remotely accessed, the calculation of the access cost is also dependant of those accesses. First, as in previous analysis, the remotely induced spin delay is calculated, as function of the remote and local accesses already accounted for (we use the recently calculated R values):

$$
\begin{aligned}
&\text{Np}^{\text{EEPROM}}_{\text{Core 0}}(R_{\text{TC\_Receiver\_Task}})\\
&\quad = \text{N}^{\text{EEPROM}}_{\text{Housekeeping\_Task}}(R_{\text{TC\_Receiver\_Task}}, R_{\text{Housekeeping\_Task}})\\
&\quad = \left\lceil \frac{R_{\text{TC\_Receiver\_Task}} + R_{\text{Housekeeping\_Task}}}{T_{\text{Housekeeping\_Task}}} \right\rceil \cdot \text{N}^{\text{EEPROM}}_{\text{Housekeeping\_Task}}\\
&\quad = \left\lceil \frac{103.492\,\text{ms} + 456.391\,\text{ms}}{1000\,\text{ms}} \right\rceil \cdot 2\\
&\quad = 2
\end{aligned}
$$

$$
\begin{aligned}
&\text{Nh}^{\text{EEPROM}}_{\text{TC\_Receiver\_Task}}(R_{\text{TC\_Receiver\_Task}})\\
&\quad = \text{N}^{\text{EEPROM}}_{\text{Deferred\_TC\_Timer}}(R_{\text{TC\_Receiver\_Task}}, R_{\text{Deferred\_TC\_Timer}})\\
&\quad = \left\lceil \frac{R_{\text{TC\_Receiver\_Task}} + R_{\text{Deferred\_TC\_Timer}}}{T_{\text{Deferred\_TC\_Timer}}} \right\rceil \cdot \text{N}^{\text{Manager.Event\_Buffer}}_{\text{Lost\_Comm\_Timer}}\\
&\quad = \left\lceil \frac{103.492\,\text{ms} + 65.417\,\text{ms}}{1.73\text{E}{+}08\,\text{ms}} \right\rceil \cdot 2\\
&\quad = 2
\end{aligned}
$$

$$\begin{aligned}
&\mathrm{NS}^{\mathrm{EEPROM}}_{\mathrm{TC\_Receiver\_Task,Core\ 0}}(R_{\mathrm{TC\_Receiver\_Task}}) \\
&\qquad = (\mathrm{Np}^{\mathrm{EEPROM}}_{\mathrm{Core\ 0}}(R_{\mathrm{TC\_Receiver\_Task}}) \\
&\qquad\quad - \mathrm{Nh}^{\mathrm{EEPROM}}_{\mathrm{TC\_Receiver\_Task}}(R_{\mathrm{TC\_Receiver\_Task}}))_0 \\
&\qquad = 2 - 2 \\
&\qquad = 0
\end{aligned}$$

As the number of access request potentially inducing spin delay not already accounted for (NS) is 0, then no spin delay is to be added to the arrival blocking calculation as:

$$\begin{aligned}
&\mathrm{Q}^{\mathrm{EEPROM}}_{\mathrm{TC\_Receiver\_Task}}(R_{\mathrm{TC\_Receiver\_Task}})(2) \\
&\qquad = (\mathrm{NS}^{\mathrm{EEPROM}}_{\mathrm{TC\_Receiver\_Task,Core\ 0}}(R_{\mathrm{TC\_Receiver\_Task}}) \\
&\qquad\quad - ((2-1)\cdot(Q^{\mathrm{EEPROM}}-1)))_0^{Q^{\mathrm{EEPROM}-1}} \\
&\qquad = (0 - ((2-1)\cdot(2-1)))_0^{Q^{\mathrm{EEPROM}-1}} \\
&\qquad = (0-2)_0^{Q^{\mathrm{EEPROM}-1}} \\
&\qquad = 0
\end{aligned}$$

As no spin delay is to be accounted for, then the access cost is equal to the access time:

$$\begin{aligned}
&e^{\mathrm{EEPROM}}_{\mathrm{TC\_Receiver\_Task}}(R_{\mathrm{TC\_Receiver\_Task}})(2(\mathrm{Write})) \\
&\qquad = c^{\mathrm{EEPROM}}_{\mathrm{TC\_Receiver\_Task}}(R_{\mathrm{TC\_Receiver\_Task}})(2(\mathrm{Write})) \\
&\qquad = 15.500\,\mathrm{ms}
\end{aligned}$$

then:

$$\begin{aligned}
&e^{\mathrm{Storage.Configuration\_Parameters}}_{\mathrm{TC\_Receiver\_Task}}(R_{\mathrm{TC\_Receiver\_Task}})(2(\mathrm{Update})) \\
&\qquad = 0.308\,\mathrm{ms} + 15.500\,\mathrm{ms} \\
&\qquad = 15.808\,\mathrm{ms}
\end{aligned}$$

and:

$$\begin{aligned} e^{\text{TTC.Configuration}}_{\text{TC\_Receiver\_Task}}(R_{\text{TC\_Receiver\_Task}})(2(\text{Save})) &= 0.475\,\text{ms} + 15.808\,\text{ms} \\ &= 16.283\,\text{ms} \end{aligned}$$

so we can finally state the arrival blocking to be accounted for as:

$$\text{B}_{\text{TC\_Receiver\_Task}} = e_{\widehat{\text{TC\_Receiver\_Task}}} = 16.283\,\text{ms}$$

Note that this is reasonable: since it was previously demonstrated that accesses from the TC_Receiver_Task ($\text{N}^{\text{EEPROM}}_{\text{TC\_Receiver\_Task}}$) did not have to be be accounted for any extra spin delay, subsequent accesses as ($\text{N}^{\text{EEPROM}}_{\text{TC\_Receiver\_Task}} + 1$) are not be accounted for that spin delay either.

It is important to bear in mind that accounting the possible spin delay as indirect spin delay due to higher-priority tasks does not mean that, for every task activation this spin delay will actually be suffered as indirect spin delay: it could be suffered as direct spin delay or arrival spin delay, or not suffer it at all. This is, however, the least pessimistic safe upper bound analysis for that spin delay.

Table 6.45: MrsP response time analysis. First iteration.

	Task	Subtasks	Priority	Period	Offset	E	WCET ms	C ms	Jitter	B	R
C_0	Housekeeping_Task		6	1000		367.369	324.700	89.022	0.000	0.000	456.391
Core_1	Sensor_Task	_1 a _4	13	2000	200	92.676	74.093	37.443	0.000	0.000	130.119
	Sensor_Task	_5	13	2000	200	92.970	74.240	37.443	0.000	0.000	130.413
	Control_Task		12	2000	1000	0.052	4.541	4.515	0.000	0.000	4.567
	Actuator_Task	_PWM_ON	11	2000	1000	17.120	1.896	1.776	4.567	0.000	23.463
	Actuator_Task	_PWM_OFF_1	11	2000	300	17.120	1.803	1.683	323.463	0.000	342.266
	Actuator_Task	_PWM_OFF_2	11	2000	300	17.120	1.803	1.683	342.266	0.000	361.069
	Actuator_Task	_PWM_OFF_3	11	2000	300	17.120	1.569	1.449	361.069	0.000	379.638
Core_2	Timing_Events	Visibility_Timer	14	5820000		0.000	0.400	0.400	0.000	65.017	65.417
	Timing_Events	Lost_Comm_Timer	14	1.73E+08		0.052	0.226	0.200	0.000	65.165	65.417
	Timing_Events	Deferred_TC_Timer	14	2000		47.857	17.482	0.625	0.000	16.935	65.417
	Radio_Listener		10	2000		1.166	2.630	1.464	0.000	16.283	68.047
	TC_Receiver_Task		9	1932		36.888	38.049	1.187	0.000	16.283	103.492
	TC_Handler		8	1897		37.333	36.560	0.582	0.000	35.121	162.875
	EV_Handler		7	2000		66.391	66.500	0.160	0.000	0.000	194.305
	Wdt_Psu		2	250		13.100	5.500	0.200	0.000	0.000	207.605
	Wdt_Fpga		1	5000		0.000	0.018	0.018	0.000	0.000	247.856

Second iteration

The results presented in table 6.45, same as in the MSRP case, imply that some of the spin delays calculated have to be reevaluated. As a result of the new response time values presented in table 6.45, some tasks can execute in parallel of more than one activation of tasks that can induce them spin delay. In particular, the Ebox.Ebox_Access spin delay suffered by the Housekeeping_Task in Core 0 and the ACS.Parameters spin delay suffered by the TC_Handler task in Core 2 have to be recalculated.

Core 0

The response time calculated for the Housekeeping_Task is notably longer than its initial value. Since it presents an intensive use of the Ebox. Ebox_Access, which allows interacting with most of the satellite physical devices, the spin delay suffered is, in comparison with its computation time, high.

In particular, this increase in its response time increases its potential parallel execution with ACS Sensor_Tasks activations to three instead of two. This has a notable effect on its spin delay and thus on its response time analysis.

The first two accesses can be concurrent with two WDT_PSU accesses plus two Sensor_Tasks. This has not changed, since the spin delay due to the WDT_PSU is not affected:

$$\begin{aligned} e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1, 2(\text{Get})) &= e'^{\text{Ebox.Ebox\_Access}}_{\text{Sensor\_Tasks}}(1, 2(\text{Get})) \\ &\quad + e'^{\text{Ebox.Ebox\_Access}}_{\text{WDT\_PSU}}(1, 2(\text{Reset\_WDT})) \\ &\quad c^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1, 2(\text{Get})) \\ &= 2 \times (3.900\,\text{ms} + 5.300\,\text{ms} + 3.900\,\text{ms}) \\ &= 26.200\,\text{ms} \end{aligned}$$

The next accesses, however, have to be analysed in a different way.

As can be seen in table 6.14, every Sensor_Task activation accesses 9 times the Get interface and 4 times the Is_Enabled interface. As a result, in three Sensor_Task activations there are 27 Get accesses and 12 Is_Enabled accesses. As a result, the following 25 accesses are analysed with a spin delay coming from Get accesses:

$$\begin{aligned} e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(3..27(\text{Get})) &= e'^{\text{Ebox.Ebox\_Access}}_{\text{Sensor\_Tasks}}(3..27(\text{Get})) \\ &\quad + c^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(3..27(\text{Get})) \\ &= 25 \times (3.900\,\text{ms} + 3.900\,\text{ms}) \\ &= 195.000\,\text{ms} \end{aligned}$$

Accounted for all 27 Sensor_Tasks Get accesses, spin delay due to the 12 Is_Enabled accesses is to be analysed:

$$\begin{aligned} e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(28..39(\text{Get})) &= e'^{\text{Ebox.Ebox\_Access}}_{\text{Sensor\_Tasks}}(1..12(\text{Is\_Enabled})) \\ &\quad + c^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(28..39(\text{Get})) \\ &= 12 \times (0.381\,\text{ms} + 3.900\,\text{ms}) \\ &= 51.372\,\text{ms} \end{aligned}$$

Finally the remaining 8 Get accesses are not expected to experience any spin delay, as well as the 44 Is_Enabled accesses:

$$\begin{aligned} e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(40..47(\text{Get})) &= c^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(40..47(\text{Get})) \\ &= 8 \times 3.900\,\text{ms} \\ &= 31.200\,\text{ms} \end{aligned}$$

$$\begin{aligned} e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1..44(\text{Is\_Enabled})) &= c^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1..44(\text{Is\_Enabled})) \\ &= 44 \times 0.381\,\text{ms} \\ &= 16.764\,\text{ms} \end{aligned}$$

The new access cost of the Ebox.Ebox_Access resource:

$$\begin{aligned}
e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}} &= e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1, 2(\text{Get})) \\
&\quad + e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(3..27(\text{Get})) \\
&\quad + e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(28..39(\text{Get})) \\
&\quad + e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(40..47(\text{Get})) \\
&\quad + e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1..44(\text{Is\_Enabled})) \\
&\quad + e^{\text{Ebox.Ebox\_Access}}_{\text{Housekeeping\_Task}}(1(\text{Battery\_Warnings})) \\
&= 26.200\,\text{ms} + 195.000\,\text{ms} + 51.372\,\text{ms} \\
&\quad + 31.200\,\text{ms} + 16.764\,\text{ms} + 0.040\,\text{ms} \\
&= 303.772\,\text{ms}
\end{aligned}$$

Including all costs to access resources, the response time of the Housekeeping_Task under MrsP during the TC phase is:

$$R_{\text{Housekeeping\_Task}} = 403.993\,\text{ms} + 89.022\,\text{ms} = 493.015\,\text{ms}$$

Core 2

As well as in the MSRP case, Core 2 tasks spin delay is also affected by the new response times. In particular, the TC_Handler access to the ACS.Parameters is to be reevaluated. Since

$$\left\lceil \frac{R_{\text{TC\_Handler}} + R_{\text{Sensor\_Tasks}}}{\mathit{Offset}_{\text{Sensor\_Tasks}}} \right\rceil = 2$$

the TC_Handler can be executed in parallel of both the end of one Sensor_Task activation and the beginning of the following, the direct spin delay is to be reevaluated.

Now, accesses 1 and 2 are considered to be concurrent with accesses from Core 1

$$\begin{aligned} e_{\text{TC\_Handler}}^{\text{ACS.Parameters}}(1,2) &= c_{\text{TC\_Handler}}^{\text{ACS.Parameters}}(1,2) \\ &\quad + e'^{\text{ACS.Parameters}}_{\text{Sensor\_Tasks}}(1,2) \\ &= 2 \times (0.026\,\text{ms} + 0.026\,\text{ms}) \\ &= 0.104\,\text{ms} \end{aligned}$$

and access 3 is done contention free:

$$e_{\text{TC\_Handler}}^{\text{ACS.Parameters}}(3) = c_{\text{TC\_Handler}}^{\text{ACS.Parameters}}(3) = 0.026\,\text{ms}$$

The update of these values is presented is table 6.46. It is important to highlight that this iteration results do not have any impact on arrival blocking. The system analysis is to be completed with the migrations overhead analysis presented in next subsection.

Table 6.46: MrsP response time analysis. Second iteration.

	Task	Subtasks	Priority	Period	Offset	E	WCET	C	Jitter	B	R
C_0	Housekeeping_Task		6	1000		403.993	324.700	89.022	0.000	0.000	493.015
Core_1	Sensor_Task	_1 a _4	13	2000	200	92.676	74.093	37.443	0.000	0.000	130.119
	Sensor_Task	_5	13	2000	200	92.970	74.240	37.443	0.000	0.000	130.413
	Control_Task		12	2000	1000	0.052	4.541	4.515	0.000	0.000	4.567
	Actuator_Task	_PWM_ON	11	2000	1000	17.120	1.896	1.776	4.567	0.000	23.463
	Actuator_Task	_PWM_OFF_1	11	2000	300	17.120	1.803	1.683	323.463	0.000	342.266
	Actuator_Task	_PWM_OFF_2	11	2000	300	17.120	1.803	1.683	342.266	0.000	361.069
	Actuator_Task	_PWM_OFF_3	11	2000	300	17.120	1.569	1.449	361.069	0.000	379.638
Core_2	Timing_Events	Visibility_Timer	14	5820000		0.000	0.400	0.400	0.000	65.017	65.417
	Timing_Events	Lost_Comm_Timer	14	1.73E+08		0.052	0.226	0.200	0.000	65.165	65.417
	Timing_Events	Deferred_TC_Timer	14	2000		47.857	17.482	0.625	0.000	16.935	65.417
	Radio_Listener		10	2000		1.166	2.630	1.464	0.000	16.283	68.047
	TC_Receiver_Task		9	1932		36.888	38.049	1.187	0.000	16.283	103.492
	TC_Handler		8	1897		37.359	36.560	0.582	0.000	35.121	162.901
	EV_Handler		7	2000		66.391	66.500	0.160	0.000	0.000	194.331
	Wdt_Psu		2	250		13.100	5.500	0.200	0.000	0.000	207.631
	Wdt_Fpga		1	5000		0.000	0.018	0.018	0.000	0.000	247.882

Migrations analysis

The benefits on the arrival blocking analysis under MrsP come with a cost: as the access to shared resources is preemptable, a mechanism is required to bound the spin delay on remote waiting tasks. This is achieved under MrsP by migrating locally preempted tasks to a processor where they can execute replacing a task spin-waiting for the locked resource.

The process of migrating a task implies certain costs, which will be calculated first. Then, the possible migrations are identified, with an upper bound on the number of those migrations. Finally the calculated costs are added to the access costs times calculated in the previous subsection to obtain the final response time analysis results.

Migration cost estimation

Under MrsP, migrating a task (only done as a result of the helping mechanism) generates an execution time overhead due to two reasons:

- Ready-queues update: migrating a task from one processor to another mainly requires moving the task information from the original scheduler ready queue to the destination (migration target) scheduler. This necessarily requires an amount of time to reorder both queues.
- Cached data invalidation: most modern computers, including those used in high-integrity and real-time systems, include cache memories, that highly reduce the average cycles required to access the computer memory by storing some of the main memory content on smaller and faster memory banks. These banks store previously accessed memory contents plus some of contents that are expected to be required in subsequent operations. If a task is

Table 6.47: Ready-queues ordering cost per processor. Cost values in ms.

Processor	Base cost	N tasks	Total Cost
Core 0	0.504	1 + 1	0.538
Core 1	0.504	3 + 1	0.572
Core 2	0.504	6	0.606

migrated to another processor, a certain amount of memory accesses will result in cache errors (as the cache contains the helper task data). The cache controller will normally update the cache content following such errors. While this might improve the performance of the helped task, it will also result in the invalidation, i.e. deallocation, of the helper task content, later resulting on new cache errors after the helping mechanism has been completed and the helper task re-dispatched.

Regarding the first source of overhead, existing implementations of MrsP have reported low costs in practice [39, 116]. In general, real-time systems run-times are implemented very efficiently on this regards, even using assembler code for some operations. This improves efficiency and allows determining more precisely the required execution time as the exact number of instructions to execute is known. In some cases, as is the case of reordering the queues, this exact value depends on the system characteristics. In this case, the amount of time required depends on the number of tasks. Based on previous metrics of the run-time system [74], this cost, on each processor is of $0.504 + 0.017 \times N$ ms where N is the number of tasks allocated on the processor. Bearing in mind that, as was explained, the ACS is implemented with just 3 tasks, the migration cost per processor due to reordering queues is shown in table 6.47. As will be explained later, only tasks from Core 2 can migrate to Cores 0 and 1. As a result, one more task has to be considered on Cores 0 and 1 for the reordering operation, while, as no task can migrate to Core 2, N is just the number of tasks it hosts.

Regarding the cache data invalidation overhead, an approach based on execution times has been followed. As mentioned before, the overhead in the execution time due to cache memory is first suffered by the migrated task, as the helping processor does not have in cache the data and instructions to be used. Then, when the migrated task leaves the helping processor, the cache content will be partially modified with content belonging to the helped task, i.e. its content has been invalidated as a result of the helping mechanism. An interesting point from the analysis perspective is that the helped task cannot invalidate more cache banks than it uses. In different words: the cache overhead caused cannot be higher than the cache overhead suffered on an execution with an empty cache, where all first accesses to a cache memory bank will result in a cache error. This appreciation can be used to measure an expectable overhead.

For the case study, the cache overhead has been calculated as follows. For each shared resource access that can lead to a migration (identification will be detailed in next subsection), three execution times have been obtained. The first one is an access with cache memory disabled, i.e. all memory accesses require an access to the main memory. The second one is an access with cache memory enabled but just flushed, i.e. the memory cache is invalidated and all banks accesses will result in a cache error on the first access. Finally the third measurement is done immediately after the second access without flushing the cache. Since the executions are the same (same interface with same parameters) the execution time difference between the second and third value is only due to the cache memory now containing the values fetched during the second execution. This is a safe upper bound for the cache overhead. Table 6.48 presents these measurements.

To calculate the final migration costs values to be taken into account for the analysis both sources of overhead have to be combined.

Table 6.48: Resource access times with different cache states.

Resource Access	Disabled	Flushed	Preloaded	Overhead
Platform Table Set_Operating_Mode	0.038	0.020	0.010	0.009
Ebox Reset_WDT	5.499	5.270	5.158	0.112
ACS.Parameters Get_Calibration	0.044	0.023	0.021	0.002

Table 6.49: Migration cost per processor and resource access.

Resource Access	$\text{mig}_{\text{Core 0}}$	$\text{mig}_{\text{Core 1}}$	$\text{mig}_{\text{Core 2}}$
Platform Table Set_Operating_Mode	0.548 ms		0.616 ms
Ebox Reset_WDT	0.650 ms	0.684 ms	0.718 ms
ACS.Parameters Get_Calibration		0.574 ms	0.608 ms

As mentioned before, the cost of a migration is the sum of incorporating the migrated task to the destination core ready queue and the effect of the cache invalidation. For each possible migration, the cost has been calculated as follows:

$$mig_m^k = q_x + mem^k \tag{6.1}$$

where k is the resource access that causes the migration, m is the destination processor, q^m is the cost of reordering the ready-queues on the destination processor (table 6.47 values) and mem^k is the cache invalidation cost of a resource k access (table 6.48 values). Table 6.49 presents the final migration costs of all possible migrations applying equation 6.1.

Migration cost analysis

The first consideration for the analysis is that, for a migration to happen, the task accessing the resource has to be preempted. That is, during the execution of a critical section a local task with higher priority than the resource ceiling priority has to be released. In the particular case of this system, this means that accesses to shared resources performed by tasks allocated on Cores 0 and 1 will not require to be migrated, since they are *de facto* non-preemptable as they execute alone on their processor. Some Core 2 accesses, however, are potentially preemptable as some tasks have higher priority than some resource local ceiling priorities. In consequence, the analysis presented below only studies Core 2 access.

Similar to tasks on Core 0 and 1, the timing event handlers are also not affected by migrations: as they are the highest-priority tasks in the system, their accessed shared resources also have the highest local ceiling priority. The Radio_Listener task cannot suffer any extra overhead due to migrations as a result of its shared resource accesses: since the HWComm.Receive_Pool and UART.Modem_UART shared resources are only shared locally, cannot induce a migration, as no remote task can be spin-waiting for those resources.

The TC_Receiver_Task does not either access any resource that can be migrated: while most of the resources it accesses have a priority equal to the timer handlers, the others, such as the mentioned HWComm.Receive_Pool, the TTC.TM_Basic and TTC.TM_Nominal resources are only accessed locally.

The TC_Handler, on the contrary, can suffer preemptions while accessing globally shared resources. Up to two resources accesses are on this situation: Platform.Table and ACS.Parameters.

The first step in the calculation of the number of possible migrations is to determine its potential migration targets. As can be seen in table 6.38, the Platform.Table access can only be migrated to Core 0, where the Housekeeping_Task is hosted. As identified in equation 5.26, to qualify as a migration target, the candidate processor needs to have requests to be accounted for on the processor of the analysed task, i.e. $\text{NS}^{k}_{x,m} \geq 1$. In this case, as it is the first access to the resource from Core 2 it is to be considered a migration target (NS = 79 in this case).

The subsequent step is to identify whether the migration targets present potential preemptors. This is done applying equation 5.27. In this case, the migration target does not present any preemptor, as the tasks is allocated alone on that processor, i.e. $mtp(mt, r^k) = \emptyset$. As a result, one of the specific cases identified in section 5.3.2 applies: as all the migration targets (in this case just one) do not present preemptors, the number of migrations in the worst-case is just 2; the initial migration as a result of being locally preempted plus the migration back to its host processor. This means that the TC_Handler migrates to Core 0 to complete its access (non-preemptively as there is no other task on Core 0) and upon completing the critical section it is migrated back to Core 2. This process, as explained before, presents a cost for both tasks: the migrated task own access time is increased, as a result of the migration overhead. The helper task (and any other waiting task, if any), in this case the Housekeeping_Task is also delayed: since the task that is causing spin delay has a longer access time, the time required for the helper task to get access to the resource is also increased. The overall cost is then the sum of migrating the WDT_PSU task to Core 0 to complete the access plus the cost of migrating it back to Core 2 after completion:

$$\begin{aligned}
\mathrm{MC}_{\text{TC\_Handler}}^{\text{Platform.Table}} &= mig_{\text{Core 0}}^{\text{Platform.Table}} + mig_{\text{Core 2}}^{\text{Platform.Table}} \\
&= 0.548\,\text{ms} + 0.616\,\text{ms} \\
&= 1.164\,\text{ms}
\end{aligned}$$

Note that this overhead is also incorporated in TC_Handler local lower-priority tasks response times as part of the TC_Handler induced indirect spin delay.

The TC_Handler can also be locally preempted while accessing the ACS.Parameters shared resource. This case requires a different analysis than the previous case. First, the remote core involved is Core 1 (ACS tasks one) instead of Core 0. Secondly, while in the previous case the remote task accesses more intensively the resource, in this case the potentially preempted task might not always find a migration target.

As can be seen in table 6.19, while Sensor_Tasks and the Control_Task access just once the resource per activation, the TC_Handler can access up to three times the resource on a single activation. This might result on cases where the TC_Handler is locally preempted but the potentially helping accesses from the remote core have already been accounted for. In those cases Core 1 is no longer considered as a migration target as defined in equation 5.26. As a result, the helping mechanism cannot take place and no overhead is to be accounted for. What does not differ from the previous analysis is the fact that when Core 1 is a valid migration target it does not present preemptors.

To calculate whether Core 1 is a valid migration target, equation 5.26 is to be applied for each of the three accesses. To do so, $\mathrm{NS}^{\mathrm{ACS.Parameters}}_{\mathrm{TC\_Handler,Core\ 1}}$ is to be calculated, as previously described for the second iteration of the analysis:

$$\begin{aligned}
\mathrm{Np}^{\mathrm{ACS.Parameters}}_{\mathrm{Core\ 1}}&(R_{\mathrm{TC\_Handler}})\\
&= \mathrm{N}^{\mathrm{ACS.Parameters}}_{\mathrm{Sensor\_Tasks}}(R_{\mathrm{TC\_Handler}}, R_{\mathrm{Sensor\_Tasks}})\\
&= \left\lceil \frac{R_{\mathrm{TC\_Handler}} + R_{\mathrm{Sensor\_Tasks}}}{\mathrm{Offset}_{\mathrm{Sensor\_Tasks}}} \right\rceil \cdot \mathrm{N}^{\mathrm{ACS.Parameters}}_{\mathrm{Sensor\_Tasks}}\\
&= \left\lceil \frac{162.901\,\mathrm{ms} + 130.119\,\mathrm{ms}}{200\,\mathrm{ms}} \right\rceil \cdot 1\\
&= 2
\end{aligned}$$

and $\mathrm{Nh}^{\mathrm{ACS.Parameters}}_{\mathrm{TC\_Handler}}(R_{\mathrm{TC\_Handler}}) = 0$ as no higher-priority task accesses the resource and thus $\mathrm{NS}^{\mathrm{ACS.Parameters}}_{\mathrm{TC\_Handler},Core1} = 2$. Then we can see that the condition to be a migration target:

$$\begin{aligned}
\mathrm{mt}^{\mathrm{ACS.Parameters}}_{\mathrm{TC\_Handler}}&(R_{\mathrm{TC\_Handler}})(n)\\
&\triangleq \{\mathrm{Core\ 1} | \mathrm{Core\ 1} \neq P(\mathrm{TC\_Handler})\\
&\quad \wedge \mathrm{NS}^{\mathrm{ACS.Parameters}}_{\mathrm{TC\_Handler,Core\ 1}}(R_{\mathrm{TC\_Handler}}) - n + 1 > 0\}\\
&\triangleq \{\mathrm{Core\ 1} | \mathrm{Core\ 1} \neq P(\mathrm{TC\_Handler}) \wedge 2 - n + 1 > 0\}
\end{aligned}$$

only holds for $n = 1$ and $n = 2$ but not for $n = 3$. As a result the helping mechanism can only be triggered twice for TC_Handler accesses to the ACS.Parameters globally shared resource. Note that this analysis is reasonable, in the sense that, for the second iteration analysis, it was already calculated that up to two accesses from Core 1 and 2 could be concurrent during each TC_Handler activation.

Each access migration cost per is calculated as previously shown:

$$\begin{aligned}\mathrm{MC}_{\mathrm{TC\_Handler}}^{\mathrm{ACS.Parameters}} &= mig_{Core1}^{\mathrm{ACS.Parameters}} + mig_{Core2}^{\mathrm{Platform.Table}} \\ &= 0.574\,\mathrm{ms} + 0.608\,\mathrm{ms} \\ &= 1.182\,\mathrm{ms}\end{aligned}$$

This is to be added once for each ACS task activation accessing the ACS.Parameters resource (Sensor_Tasks and Control_Task). As mentioned before, for the TC_Handler it is to be added twice per activation, resulting in a total overhead of 2.364 ms per activation.

The Event_Handler does not access any globally shared resource that could trigger a migration. The WDT_PSU accesses the EBOX.EBOX_Access resource that is also accessed from Cores 0 and 1. As the WDT_PSU is the only task accessing the resource on Core 2 the local ceiling priority of the resource is fairly low, being preemptable by almost any task in that core. In this case, both remote cores are possible migration targets, as $N_{\mathrm{Housekeeping\_Task}}^{\mathrm{EBOX.EBOX\_Access}}(R_{\mathrm{WDT\_PSU}}, R_{\mathrm{Housekeeping\_Task}}) = 184$ and $N_{\mathrm{Sensor\_Tasks}}^{\mathrm{EBOX.EBOX\_Access}}(R_{\mathrm{WDT\_PSU}}, R_{\mathrm{Sensor\_Tasks}}) = 26$. As discussed previously, both processors present no preemptors. However, it is not irrelevant to which processor the WDT_PSU is migrated to. As shown in table 6.49, the cost of migrating to Core 1 is slightly higher, since it hosts more tasks than Core 0. So, the worst-case migration for all three tasks is:

$$\begin{aligned}\mathrm{MC}_{\mathrm{WDT\_PSU}}^{\mathrm{Ebox.Ebox\_Access}} &= mig_{Core1}^{\mathrm{Ebox.Ebox\_Access}} + mig_{Core2}^{\mathrm{Ebox.Ebox\_Access}} \\ &= 0.684\,\mathrm{ms} + 0.718\,\mathrm{ms} \\ &= 1.402\,\mathrm{ms}\end{aligned}$$

Note that, as identified in the second iteration analysis, the Housekeep ing_task response time is longer than the WDT_PSU period and as a result up to two remote access to the resource from Core 2 are considered for the Housekeeping_Task response time. As each of those accesses can be preempted and cause a migration, the overhead per Housekeeping_Task activation is double (2.804 ms).

Incorporating the calculated migration costs to the resource access costs, the final response time analysis is obtained. Note that, as the response time of the WDT_FPGA went over the 250 ms of the WDT_PSU, it is interfered twice per activation in the worst case. The result of this analysis is presented in table 6.50.

Summary

The response times of the implementation under MrsP are comparable to those of under MSRP. In particular, the differences between them are due to the cost of migrating tasks (non existent under MSRP) and the arrival blocking (lower under MrsP). The previously highlighted Timing_Events, in particular, have reduced their response time values from over 84 ms to 65.417, ms thanks to the reduced priority inversion under MrsP, which is the main aim of the protocol. An in-depth analysis and comparison of results is presented in the following section.

Table 6.50: MrsP response time analysis. Final results including migration costs.

	Task	Subtasks	Priority	Period	Offset	MC	E	WCET	C	Jitter	B	R
C_0	Housekeeping_Task		6	1000		3.968	407.961	324.700	89.022	0.000	0.000	496.983
Core_1	Sensor_Task	_1 a _4	13	2000	200	2.584	95.260	74.093	37.443	0.000	0.000	132.703
	Sensor_Task	_5	13	2000	200	2.584	95.554	74.240	37.443	0.000	0.000	132.997
	Control_Task		12	2000	1000	2.584	2.636	4.541	4.515	0.000	0.000	7.151
	Actuator_Task	_PWM_ON	11	2000	1000	0.000	17.120	1.896	1.776	7.151	0.000	26.047
	Actuator_Task	_PWM_OFF_1	11	2000	300	0.000	17.120	1.803	1.683	326.047	0.000	344.850
	Actuator_Task	_PWM_OFF_2	11	2000	300	0.000	17.120	1.803	1.683	344.850	0.000	363.653
	Actuator_Task	_PWM_OFF_3	11	2000	300	0.000	17.120	1.569	1.449	363.653	0.000	382.222
Core_2	Timing_Events	Visibility_Timer	14	5820000		0.000	0.000	0.400	0.400	0.000	65.017	65.417
	Timing_Events	Lost_Comm_Timer	14	1.73E+08		0.000	0.052	0.226	0.200	0.000	65.165	65.417
	Timing_Events	Deferred_TC_Timer	14	2000		0.000	47.857	17.482	0.625	0.000	16.935	65.417
	Radio_Listener		10	2000		0.000	1.166	2.630	1.464	0.000	16.283	68.047
	TC_Receiver_Task		9	1932		0.000	36.888	38.049	1.187	0.000	16.283	103.492
	TC_Handler		8	1897		3.528	40.887	36.560	0.582	0.000	35.121	166.429
	EV_Handler		7	2000		0.000	66.391	66.500	0.160	0.000	0.000	197.859
	Wdt_Psu		2	250		1.402	14.502	5.500	0.200	0.000	0.000	212.561
	Wdt_Fpga		1	5000		0.000	0.000	0.018	0.018	0.000	0.000	267.514

6.5 Evaluation Summary

In this chapter the analysis of a real system has been conducted for three different implementation choices: monocore Ravenscar system, triplecore MSRP system and triplecore MrsP system. For each analysis the most advanced analysis techniques have been used, thoroughly described on each section, especially for the multicore analysis, main topic of this thesis.

The analysis of a real system is a notable contribution by itself. The lack of real-world examples coming from the industry has always been an issue within the real-time academic community. The development of the UPMSat-2 has demonstrated to be an invaluable source of both data and inspiration for research efforts as those presented in this thesis. In particular, for the best of our understanding the analysis and comparison of a real system with these three approaches has never been published before.

The analysis obtained in sections 6.2, 6.3 and 6.4 provide interesting results. The main benefits and drawbacks of each implementation approach are, to some extent, present on the response time analysis results. One initial conclusion that can be extracted from this analysis is that none of the approaches strictly dominates another in practice. Each of them yields a lower response time for a subset of the system tasks as can be seen in table 6.51.

The tasks granted the four highest priorities in the monocore implementation present the lowest response times precisely on that implementation. These are the timing event handlers plus the Sensor_Task. This is consequence of two main factors: they suffer a low arrival blocking equal to that of MrsP, and their accesses to shared resources do not incur in any spin delay. The relatively low execution time of the timing events results, for the Sensor_Task, in a lower interference than the cost of sharing resources among processors. For lower-priority

Table 6.51: UPMSat-2 response time comparison between approaches. R and Δ Mono values expressed in ms.

		Monoprocessor	MSRP			MrsP		
Task	Job	R	R	Δ Mono	%Δ	R	Δ Mono	%Δ
Housekeeping_Task		862.459	493.015	-369.444	-42.84%	496.983	-365.476	-42.38%
Sensor_Task	_1 a _4	108.484	130.119	21.635	19.94%	132.703	24.219	22.32%
Sensor_Task	_5	108.631	130.413	21.782	20.05%	132.997	24.366	22.43%
Control_Task		38.932	4.567	-34.365	-88.27%	7.151	-31.781	-81.63%
Actuator_Task	_PWM_ON	75.219	23.463	-51.756	-68.81%	26.047	-49.172	-65.37%
Actuator_Task	_PWM_OFF_1	411.413	342.266	-69.147	-16.81%	344.85	-66.563	-16.18%
Actuator_Task	_PWM_OFF_2	447.607	361.069	-86.538	-19.33%	363.653	-83.954	-18.76%
Actuator_Task	_PWM_OFF_3	483.567	379.638	-103.929	-21.49%	382.222	-101.345	-20.96%
Timing_Events	Visibility_Timer	34.391	84.255	49.864	144.99%	65.417	31.026	90.22%
Timing_Events	Lost_Comm_Timer	34.391	84.255	49.864	144.99%	65.417	31.026	90.22%
Timing_Events	Deferred_TC_Timer	34.391	84.255	49.864	144.99%	65.417	31.026	90.22%
Radio_Listener		259.741	86.885	-172.856	-66.55%	68.047	-191.694	-73.80%
TC_Receiver_Task		220.773	122.33	-98.443	-44.59%	103.492	-117.281	-53.12%
TC_Handler		238.122	162.901	-75.221	-31.59%	166.429	-71.693	-30.11%
EV_Handler		419.394	207.431	-211.963	-50.54%	197.859	-221.535	-52.82%
Wdt_Psu		862.659	207.631	-655.028	-75.93%	207.631	-650.098	-75.36%
Wdt_Fpga		879.177	247.882	-631.295	-71.81%	267.514	-611.663	-69.57%

tasks, due to the high processor utilization of the Sensor_Task, this is no longer true.

Table 6.52: UPMSat-2 priority inversion time comparison between approaches. R and Δ Mono values expressed in ms.

		Monop.	MSRP		MrsP	
Task	Job	B	B	Δ Mono	B	Δ Mono
Housekeeping_T		5.300	0.000	-5.300	0.000	-5.300
Sensor_Task	_1 a _4	16.283	0.000	-16.283	0.000	-16.283
Sensor_Task	_5	16.283	0.000	-16.283	0.000	-16.283
Control_Task		16.283	0.000	-16.283	0.000	-16.283
Actuator_Task	_PWM_ON	16.283	0.000	-16.283	0.000	-16.283
Actuator_Task	_PWM_OFF_1	16.283	0.000	-16.283	0.000	-16.283
Actuator_Task	_PWM_OFF_2	16.283	0.000	-16.283	0.000	-16.283
Actuator_Task	_PWM_OFF_3	16.283	0.000	-16.283	0.000	-16.283
Timing_Events	Visibility	33.991	83.855	49.864	65.017	31.026
Timing_Events	Lost_Comm	34.165	84.003	49.838	65.165	31.000
Timing_Events	Deferred_TC	16.909	35.773	18.864	16.935	0.026
Radio_Listener		16.283	35.121	18.838	16.283	0.000
TC_Receiver_T		16.283	35.121	18.838	16.283	0.000
TC_Handler		35.121	35.121	0.000	35.121	0.000
EV_Handler		35.121	13.100	-22.021	0.000	-35.121
Wdt_Psu		0.000	0.000	0.000	0.000	0.000
Wdt_Fpga		0.000	0.000	0.000	0.000	0.000

Apart from the Sensor_Task, the other ACS tasks and the Housekeeping_Task present the best response times under the MSRP analysis. This is an interesting result, since the main drawback of the multiprocessor approaches is the cost of sharing resources among processors, and both the Housekeeping_Task and the Actuator_Tasks make intensive use of a resource shared between them, the Ebox.Ebox_Access. This result is achieved as a consequence of having allocated those tasks in a manner such that neither of them may suffer interference or arrival blocking, overcoming the costs of the multiprocessor resource sharing.

The gap between MSRP and MrsP response times for these tasks is, however, relatively small, as can be seen in table 6.53. As was dis-

Table 6.53: UPMSat-2 response time comparison between MSRP and MrsP approaches. R and Δ values expressed in ms.

		MSRP	MrsP		
Task	Job	R	R	Δ	%Δ
Housekeeping_Task		493.015	496.983	3.968	0.80%
Sensor_Task	_1 a _4	130.119	132.703	2.584	1.99%
Sensor_Task	_5	130.413	132.997	2.584	1.98%
Control_Task		4.567	7.151	2.584	56.58%
Actuator_Task	_PWM_ON	23.463	26.047	2.584	11.01%
Actuator_Task	_PWM_OFF_1	342.266	344.85	2.584	0.75%
Actuator_Task	_PWM_OFF_2	361.069	363.653	2.584	0.72%
Actuator_Task	_PWM_OFF_3	379.638	382.222	2.584	0.68%
Timing_Events	Visibility_Timer	84.255	65.417	-18.838	-22.36%
Timing_Events	Lost_Comm_Timer	84.255	65.417	-18.838	-22.36%
Timing_Events	Deferred_TC_Timer	84.255	65.417	-18.838	-22.36%
Radio_Listener		86.885	68.047	-18.838	-21.68%
TC_Receiver_Task		122.33	103.492	-18.838	-15.40%
TC_Handler		162.901	166.429	3.528	2.17%
EV_Handler		207.431	197.859	-9.572	-4.61%
Wdt_Psu		207.631	212.561	4.93	2.37%
Wdt_Fpga		247.882	267.514	19.632	7.92%

cussed during the analysis, the access cost analysis (direct spin delay) for both protocols is equal without considering the MrsP migration overhead. Cores 0 and 1, as they do not present arrival blocking, would have shown equal response times without migration costs. These costs are quite relevant, in contrast to existing evaluations in the literature. This is because those evaluations have been carried out in modern, general-purpose high-end platforms. In contrast, the UPMSat-2 computer platform is a real-time high-integrity aimed processor architecture implemented on a FPGA. While the former commonly execute at a processor clock speed of $\approx 3\,\text{GHz}$, the UPMSat-2 processor clock speed is of $20\,\text{MHz}$. This explains why the process of incorporating the migrated task to the migration target ready queue (that depends only on computing speed) is so costly on the UPMSat-2 case study.

Even with that migration overhead MrsP yields a better response time for a set of tasks in the system. In particular, some telecommand related tasks present the best response times under MrsP. This is not a minor result. The telecommand messages are the only way to remotely control and manage the satellite system, and, as a result, the correct and timely execution of telecommands is vital. Also, the Event_Handler does also present a lower response time under MrsP, and clearly better than in the monocore implementation. Again, a shorter response time of this task enables a better response to certain critical events for a satellite system, as the detection of a low battery status. These tasks present a lower blocking than under MSRP as can be seen in table 6.52 overcoming the migrations overheads, what is the main point of the MrsP proposal.

It is relevant to note, however, that some MrsP results are benefited from the task allocation envisaged. In particular the access cost to inner nested resources gets the most benefit. As was shown in section 5.2 and discussed during MrsP analysis in this chapter, inner nested resource analysis is also benefited by the fact that, in the case study, all migration targets do not present preemptors. This avoids the case where tasks have to migrate back to their own processor to make progress in a resource being helped.

The nested resource access cost and migration overhead have also been optimized by artificially increasing the local ceiling priority of a resource. This, while not negatively affecting the response time of higher-priority tasks (as their arrival blocking was higher than the resource access cost) highly reduced the cost of accessing a time consuming resource as the EEPROM memory. This benefit was obtained not only by the local tasks accessing the resource on the processor where the ceiling priority was increased, but also remote tasks. This benefit was obtained for both spin delay and migrations analysis, demonstrating that, in practice, different improvements can be done when the system behaviour is fully understood. This kind of improvements are rarely included in evaluations using synthetic workloads.

Migration analysis could have been further improved for a Core 2 task. The TC_Handler can be migrated to Core 1, but the task on Core 1 accesses only twice the ACS.Parameters. Then, it is clear that Core 1 can cause only one of the following (considering the two accesses overlapping with those of TC_Handler): 2 spin delays, 2 migrations or one of each, but never 2 spin delays and 2 migrations (that would require 4 accesses from Core 1).

Another relevant evaluation could be made with regard to the quality of the system behaviour given the presented analysis. It is commonly assumed that shorter response times for application tasks tend to be better from the scheduling point of view. However, the systems implemented by those tasks can be sensitive to other variables, such as, for example, variability. In the UPMSat-2 case, the ACS system is particularly sensitive to that parameter. As mentioned during the case study presentation, the ACS takes measurements each 200 ms for each axis and magnetometer. These are used to infer the satellite spin speed per axis by dividing the variation of measured magnetic field by the time between measurements. This means that, if the measurements are not taken exactly separated by those 200 ms, results will be altered. If the time between 2 measurements is shorter the inferred speed will be faster than the real one, and, with the same logic, if time between 2 measurements is longer the inferred speed will be slower than the real one. In a similar way, the ACS actuation is applied by a PWM mechanism on each axis. If the real time span between activation and deactivation of each magnetorquer (effective PWM pulse) is not of the exact amount of time envisaged the actuation will not be the expected one. While the response time analysis does not, by itself, provide means to study the performance of the ACS control system, some considerations can be made. The monocore implementation yields the lowest response times for the Sensor_Tasks. The worst-case arrival blocking is of 16.283 ms, but when a Sensor_Task is dispatched it has direct access to all resources. This means that the variability between measurements is only due to blocking and interference, since no spin delay might be suffered while accessing shared resources. Furthermore, the highest-priority entities are the timing event handlers, which have a extremely high inter-arrival time (the Deffered_TC has a minimum inter arrival time equal to the ACS period, but, since only up to 8 deferred TCs can be programmed per pass over the ground station, the overall interference is reduced). As a result, it can be said that, for all Sensor_Task measurements, the variability is of 16.283 ms. In the multiprocessor case, however, all ACS tasks are free of arrival blocking or interference. They can suffer, however, spin delay due to the remote accesses of the WDT_PSU and Housekeeping_Task to the to the Ebox.Ebox_Access resource. Given that 4 accesses to the Ebox.Ebox_Access resources are done each Sensor_Task activation before actual MGM measurements are taken, the first measurement can suffer as much as:

$$1 \cdot \text{WDT\_PSU (Reset\_WDT)} + 5 \cdot \text{Housekeeping\_Task (Get)}$$

possible (but not necessarily) spin delays, which generate a variability of 24.800 ms. The ninth measurement, the last in the series, can suffer up to:

$$1 \cdot \text{WDT\_PSU (Reset\_WDT)} + 13 \cdot \text{Housekeeping\_Task (Get)} = 56.000\,\text{ms}$$

of variability. It can then be expected that the ACS subsystem will behave worst in either multicore approach than in the monocore one if no measures are taken to feed back the actual time between measurements to the control algorithm.

The results and analysis presented in this section are representative as being those of a real system rather than a synthetic case study. The conclusions obtained on the performance of each implementation approach are, however, limited by the uniqueness of each individual system. The convenience of adopting one or another seems to be specific of each task set and execution platform characteristics.

Chapter 7

Conclusions

The main objective of this thesis has been to contribute to the design and analysis of high-integrity real-time systems over multiprocessor platforms. In particular, the scheduling of multiprocessor systems with complex resource sharing schemes has been addressed.

The main contribution of the present work is a set of extensions to a promising multiprocessor systems oriented protocol: the Multiprocessor resource sharing Protocol, MrsP. This protocol unites the use of spinlock resource access policy with the ceiling priority task model to provide a bounded access cost to sharing resources with limited priority inversion.

The revision of existing major results on real-time systems presented in chapters 2 and 3 provides a meaningful context supporting the interest and relevance of the PhD contributions.

Monoprocessor real-time systems have achieved a notable maturity with regards of the scheduling of independent tasks as well as those task sets presenting shared resources. Two main dispatching policies were shown to be the most convenient considering predictability, efficiency and implementability: the Earliest Deadline First and Fixed Priority policies. These policies have associated scheduling analysis techniques that can prove the feasibility of a system in finite time and with high-utilization bounds. Furthermore, these approaches can support complex tasking models including shared resources with nested calls: the Stack Resource Policy and the Priority Ceiling Protocol are equivalent solutions for EDF and FP schedulers that enable the implementation of deadlock-free systems with bounded blocking times.

Multiprocessor platforms provide increasing computing capacity enabling the incorporation of more complex features to the final systems. These platforms can also provide means to reduce the number of parts in complex and distributed systems as cars or aeroplanes, reducing energy consumption, wiring weight and maintenance costs. As discussed in the introduction, these benefits motivate the adoption of multiprocessor platforms.

However, these platforms present a number of novel challenges for the development of real-time systems. While adding computing units may increase the computing capacity it also increases the complexity when dealing with concurrency. As stated in the introduction, the hardware architecture of multiprocessor systems introduces new scheduling issues. Even tasks not sharing logical resources do necessarily contend for physical resources to be accessed in mutual exclusion, such as memories or communication buses. This, combined with the convenience of sharing logical resources, presents a set of problems to solved: how to determine which task should execute and on which computing unit, how to arbitrate the access to shared resources so deadlocks are prevented and access costs are bounded or how to obtain worst-case execution times for those platforms are some of the most relevant as identified in the introduction.

The Multiprocessor resource sharing Protocol is a general purpose protocol that focuses on the resource sharing problem. It has been, however, mainly studied (including this thesis) for partitioned fixed priority systems. The proposed resource sharing mechanism presents a balanced compromise between low access costs bounds and a limited effect on higher-urgency tasks by means of an innovative helping mechanism. This balance is what makes MrsP outstanding when compared with other multiprocessor protocols. This thesis has contributed to the study and development of the protocol in the following aspects:

- A scheduling analysis for shared resources with heterogeneous access costs has been formalized. This contribution reduces the pessimism of assuming a general worst access time to a resource regardless of the operation to execute. Furthermore, it serves as a starting point for the improved accuracy of the new analysis methods contributed in this thesis.
- A policy for nested resource accesses has been formalized and a fine grained schedulability analysis has been provided for that policy. This contribution enables the use of a flexible and expressive task and resource model substantially increasing the protocol soundness to be industrially adopted, one of the main goals of this thesis.
- A contribution on the development of a new, less pessimistic schedulability analysis. This contribution was developed by the Real-Time Systems group at University of York during a predoctoral research stay.
- This new analysis has been adapted to the proposed nested resource access policy. As a result, the nested resource model can be analysed using the most accurate methods for FIFO spin-locking protocols supported by a helping mechanism.
- A set of possible designs for semi-partitioned systems based on MrsP have been outlined. These approaches alleviate the allocation problem inherent to multiprocessor systems, by providing a set of rules for the safe splitting of computation associated to a single task among different processors in the system.

- The protocol has been evaluated based on a space mission case study, and the results compared with a monocore implementation of the same system as well as with an implementation based on an equivalent multiprocessor protocol, MSRP. This evaluation complements the evaluations presented by the author and other research groups based on synthetic workloads. This case study has also been used to exemplify how the protocol is used in practice, and how different levels of system knowledge can be used to provide a tighter schedulabilty analysis.

The outcome of this thesis is a multiprocessor resource sharing protocol that is both flexible and expressive enough to support the development of complex systems while being able to provide the highest levels of assurance required in hard real-time systems.

Although this thesis has contributed in a number of ways to the design of multiprocessor real-time systems, the quantity and complexity of the challenges that the topic presents allows to cite a number of future and complementary research lines:

- Spin-locking has been identified as the most convenient locking approach for multiprocessor real-time systems in a number of works. The most widely studied (including this thesis) multiprocessor resource sharing protocols, MSRP and MrsP are based on this locking policy. It is to be further investigated whether this preference holds for extreme cases where resource accesses are extremely long or short compared with tasks computation times.
- Increasing the number of computing units can reduce task interference at a cost of increasing resource access overheads. Many-core architectures can nowadays provide a number of computing units comparable to the number of tasks in a system. FIFO spin-locking, as currently understood, does not scale well for that scenario. It is to be further studied whether a new approach is

required for those platforms and where is the break-even point between FIFO spinlocks and its alternatives, such as non-blocking synchronization.

- Evaluation work based on synthetic workloads published during the research leading to this PhD and the case study in section 6 have not specifically addressed the allocation problem. In particular, the system task allocation studied in section 6 is the result of previous evaluations and preliminar analysis conducted, but not necessarily the one producing the best results. All possible task allocation combinations have not been studied. Literature currently lacks of practical results on multiprocessor task allocation schemes on systems including shared resources.
- Most evaluation and comparison works on real-time scheduling protocols performance have focused the analysis on effective utilization ratios, number of schedulable systems, lower response times, etc. However, there are some other parameters that have an influence on final system performance, such as deviation in response times, that are normally excluded from evaluation work. Deeper knowledge of real-time requirements on industrial practice is required to aim future research to valuable results rather to marginal gains on profitless properties.
- The emergence of multiprocessor platforms has fostered the development of new programming paradigms and techniques. While in this thesis the study has focused on the effect of executing more than one task at a time, multiprocessor platforms can be also used to make progress on a single task on more than one processor, i.e. parallel programming. The applicability of these new techniques to real-time systems is to be further investigated.
- Scheduling analysis is based on different timing properties, with worst-case execution time (WCET) being among the most relevant ones. WCET values are difficult to obtain, and even harder

to provide evidence that they are really safe worst-case estimations. These difficulties are mostly driven by hardware improvements, that increase the average performance on detriment of timing predictability: e.g. cache memories, branch prediction mechanisms, etc. Multiprocessor platforms can be seen as a new hardware improvement towards performance at the cost of predictability. If system feasibility analysis are to be execution-time driven, new techniques are required to bound the loss of predictability on WCET measurements.

These research lines would complement the results and contributions of the present dissertation and help enabling the development of a new generation of strict real-time systems based on multiprocessor platforms.

Conclusiones

El principal objetivo de esta tesis ha sido el contribuir al diseño y análisis de sistemas de tiempo real y alta integridad sobre plataformas multiprocesador. En particular, se ha contribuido en el diseño y planificación de sistemas multiprocesador que soportan la comunicación y sincronización de diferentes hilos de ejecución mediante recursos compartidos, ya sean lógicos o físicos.

La principal contribución de la tesis es una política de compartición de recursos para sistemas multiprocesador que soporta el anidamiento en la adquisición de los mismos. Esta política se ha diseñado para complementar la ya existente para un protocolo de relevancia en el ámbito, como es el Multiprocessor resource sharing Protocol, MrsP por sus siglas en inglés. Este protocolo aúna el uso de mecanismos de espera activa junto con el bien conocido protocolo de techo de prioridad para proveer al sistema de un coste acotado para el acceso a recursos compartidos, limitando los periodos de inversión de prioridad.

En este trabajo se ha realizado una profunda revisión del estado del arte, tanto en el soporte en sistemas monoprocesador en el capítulo 2 como multiprocesador en el capítulo 3. En los mismos se ha profundizado en aquellos aspectos de relevancia para la propuesta de la tesis, demostrando su interés y relevancia en el estado actual de la técnica.

Como punto de partida, puede considerarse que los sistemas de tiempo real sobre monoprocesadores presentan hoy día un alto estado de madurez tecnológica tanto para sistemas con tareas independientes como aquellas que se coordinan mediante recursos compartidos. Las dos principales estrategias de planificación, dadas sus propiedades de predecibilidad, eficiencia e implementabilidad son Earliest Deadline First (EDF) y Fixed Priorities (FP). Ambas estrategias llevan aparejadas técnicas que permiten demostrar que los requisitos temporales del sistema se van a cumplir durante la ejecución del sistema. Finalmente, se ha de destacar que ambas estrategias soportan la compartición de recursos incluyendo su acceso anidado, objeto de estudio de esta tesis. Los protocolos de pila de recursos (Stack Resource Policy, SRP) y de techo de prioridad (Priority Ceiling Protocol, PCP) son soluciones equivalentes para planificadores EDF y FP que permiten soportar el mencionado modelo de computación.

Con respecto a las plataformas multiprocesador, estas proveen de una mayor capacidad de cómputo, al incorporar un mayor número de núcleos de ejecución. De este modo permiten la implementación de sistemas con mayores funcionalidades. Estas plataformas también permiten reducir el número de computadores embarcados en los sistemas finales, como son automóviles o aviones, reduciendo su consumo eléctrico, peso y costes de mantenimiento.

Estas plataformas multiprocesador presentan, sin embargo, diferentes dificultades para su aplicación a sistemas de tiempo real estricto. En particular, la posibilidad de ejecutar más de un hilo a la vez incrementa notablemente la dificultad de demostrar las diferentes propiedades temporales del sistema. Dado que determinados recursos del sistema no pueden multiplicarse del mismo modo que el número de núcleos de ejecución, como la memoria, incluso tareas totalmente independientes desde el punto de vista lógico pueden verse afectadas por la ejecución de más de un hilo a la vez sobre la misma plataforma. Si a esto se añade la conveniencia de compartir recursos lógicos entre tareas no necesariamente alojadas en el mismo procesador se pueden enumerar los siguientes problemas a resolver: cómo determinar que hilo ejecuta en cada momento y en que procesador, cómo arbitrar el acceso a recursos compartidos (lógicos y físicos), cómo prevenir interbloqueos, o cómo obtener tiempos de ejecución válidos para el análisis temporal son algunos de los retos para los sistemas de tiempo real sobre plataformas multiprocesadores.

El protocolo MrsP es un protocolo de compartición de recursos sobre plataformas multiprocesador de propósito general, en el sentido en que no está específicamente enfocado a sistemas EDF o FP, ni asume ninguna política de asociación de tareas a procesadores. No obstante, hasta la fecha MrsP ha sido principalmente estudiado (incluyendo la presente tesis doctoral) para sistemas con particionado estricto y prioridades fijas (FP). La principal característica del protocolo MrsP es que presenta un balanceado compromiso entre un bajo coste de acceso a recursos compartidos y una limitada duración de periodos de inversión de prioridad. Este compromiso puede alcanzarse gracias a un innovador mecanismo de ayuda, que permite continuar el progreso en un recurso compartido de una tarea localmente desalojada. Esta tesis contribuyo en el estudio y desarrollo del protocolo MrsP en los siguientes aspectos:

- Se ha formalizado un análisis que soporta el estudio de recursos compartidos cuyo tiempo de acceso no es homogéneo. Esta contribución reduce el pesimismo fruto de asumir un peor caso de tiempo de acceso igual para las diferentes operaciones que pueden realizarso sobre un determinado recurso compartido.

- Se ha definido una política de acceso anidado a recursos que permite un análisis de grano fino del coste de acceso a esos recursos. Así, se ha provisto al protocolo de un modelo de computación flexible que ofrece la expresividad necesaria para desarrollar sistemas con relaciones complejas en tareas de manera eficiente.

- Se ha contribuido al desarrollo de una estrategia de análisis de la planificabilidad del sistema que tiene en cuenta los patrones de frecuencia en el acceso a recursos compartidos. Esta contribución se ha desarrollado en el grupo de Sistemas de Tiempo Real de la Universidad de York durante una estancia investigadora.

- Se ha adaptado el nuevo análisis de MrsP a la política de recursos anidados previamente definida. De este modo el análisis más ajustado disponible para protocolos con colas FIFO y espera activa con mecanismos de ayuda puede ser utilizado para el estudio de sistemas que implementan el protocolo MrsP con soporte para recursos anidados.
- Se han adelantado diferentes posibilidades para la construcción de sistemas semiparticionados que soporten la compartición de recursos mediante MrsP. Así, se reduce de forma notable el problema de la asignación de tareas a procesadores al permitir, respetando el conjuto de reglas definido, ejecutar fracciones de una misma tarea en diferentes procesadores.
- Se ha demostrado que el protocolo MrsP puede ser usado en un sistema real como el microsatélite UPMSat-2, para el cual se ha analizado una implementación sobre una plataforma con tres núcleos de ejecución.
- Se ha evaluado el comportamiento del protocolo mediante su comparación con un protocolo multiprocesador equivalente, MSRP, y con la implementación monoprocesador siguiendo el modelo de tareas del perfil de Ravenscar. Esta evaluación complementa la ya realizada mediante casos sintéticos y demuestra que en la práctica el protocolo MrsP, con las contribuciones de esta tesis, es un protocolo de interés que permite obtener mejores tiempos de respuesta que ningún otro protocolo existente.

Si bien en esta tesis se ha contribuido de diferentes maneras al diseño de sistemas de tiempo real multiprocesador, la complejidad del tema a tratar permite identificar algunas líneas de trabajo futuro y complementario:

- A pesar de que los protocolos de espera activa han demostrado su conveniencia para sistemas multiprocesador, es posible que estos no escalen bien para la siguiente generación de plataformas, conocidas como *manycore* en las cuales el número de núcleos de computación es comparable al número de tareas en el sistema. Procede, por tanto, el estudio de otras estrategias que permitan en un futuro aprovechar plenamente dichas plataformas.
- Un aspecto fundamental de los sistemas de tiempo real sobre plataformas multiprocesador es la decisión de sobre qué núcleo de computación ejecutar cada tarea. Ya sea durante el diseño del sistema o en tiempo de ejecución esta tarea tiene una gran influencia en la planificabilidad de los sistemas, y actualmente no se conocen estrategias óptimas que puedan resolverse en tiempo lineal. En particular el estado de la técnica carece de referencias relevantes en cuanto al particionado de sistemas que comparten recursos entre procesadores.
- La mayoría de los trabajos de evaluación y comparación en el ámbito del tiempo real se centran en el estudio de tiempos de respuesta, ratios de utilización máximos, relación de sistemas planificables, etcétera. Estas comparativas obvian que en la práctica hay otros factores que afectan notablemente el comportamiento de los sistemas de tiempo real, como la variabilidiad de los tiempos de respuesta durante la ejecución. Para que se puedan desarrollar mejores protocolos en el futuro, un mayor conocimiento de los requisitos de la industria en este aspecto es necesario.
- El surgimiento de las plataformas multiprocesador ha motivado el desarrollo de diferentes técnicas y paradigmas de programación, como es el paralelismo. Este enfoque permite la ejecución

de código asociado a una misma tarea en más de un núcleo al mismo tiempo (en contraposición con el modelo presentado en esta tesis, en el cual cada procesador ejecuta código de tareas distintas). El estudio de la aplicabilidad de estos enfoques merece una especial atención, en tanto en cuanto pueden aportar notables incrementos de rendimiento en algunas aplicaciones específicas de los sistemas de tiempo real.

- El estudio de la planificabilidad de los sistemas de tiempo real se basa en diferentes parámetros temporales de las tareas a ejecutar. Entre ellos se encuentran el peor tiempo de cómputo (WCET por sus siglas en inglés). Estos valores son difíciles de obtener debido a la incorporación de diferentes mejoras hardware que incrementan el rendimiento medio del sistema a costa de la predecibilidad del peor caso de cómputo. La inclusión de más de un núcleo de ejecución en la plataforma puede verse como otro mecanismo que una vez más incrementa el redimiento a costa de la predecibilidad. Dada la necesidad de conocer los mencionados valores de peor tiempo de cómputo, son necesarias nuevas técnicas específicas de adquisición de estos datos que permitan su estudio en plataformas multiprocesador.

Estas líneas de investigación complementan los resultados y contribuciones de la tesis facilitando el desarrollo de una nueva generación de sistemas de tiempo real estricto sobre plataformas multinúcleo.

Appendix A

Ravenscar Profile Definition

The Ravenscar profile can be enforced, since Ada 2005 language revision by using the single pragma **pragma** Profile (Ravenscar). The use of this single profile pragma is equivalent to the use of the pragmas shown in listing A.1 in the current language revision Ada 2012 [9].

Ada 2012 language revision included in the Real-Time Systems annex (D) the notion of multiprocessor systems. The use of those platforms, while benefiting the average system performance, can decrease the timing predictability. To limit this effect, two new restrictions were included in the profile definition.

The first one is the No_Dynamic_CPU_Assignment restriction that, in practice, prevents explicit task migrations among processors. General Ada tasking runtime allows this by the Set_CPU statement, that can only be used at task specification blocks under the Ravenscar profile.

The second restriction is the no dependence of the System.Multiprocessors.Dispatching_Domains. Ada dispatching domains allows processors to be grouped together, and tasks allocated to dispatching domains and, optionally, to specific processors. While tasks are not allow to migrate across domains, they are allowed to do so inside their own domain. Furthermore, if a task is not allocated to a specific processor is the scheduler decision where to allocate it. By restricting the dependence to the dispatching domain package, all tasks are statically assigned a processor explicitly on their specification block or implicitly to the CPU where the environment task is allocated.

Listing A.1: Ravenscar profile equivalent pragmas.

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
            No_Abort_Statements,
            No_Dynamic_Attachment,
            No_Dynamic_CPU_Assignment,
            No_Dynamic_Priorities,
            No_Implicit_Heap_Allocations,
            No_Local_Protected_Objects,
            No_Local_Timing_Events,
            No_Protected_Type_Allocators,
            No_Relative_Delay,
            No_Requeue_Statements,
            No_Select_Statements,
            No_Specific_Termination_Handlers,
            No_Task_Allocators,
            No_Task_Hierarchy,
            No_Task_Termination,
            Simple_Barriers,
            Max_Entry_Queue_Length => 1,
            Max_Protected_Entries => 1,
            Max_Task_Entries => 0,
            No_Dependence => Ada.Asynchronous_Task_Control,
            No_Dependence => Ada.Calendar,
            No_Dependence => Ada.Execution_Time.Group_Budgets,
            No_Dependence => Ada.Execution_Time.Timers,
            No_Dependence => Ada.Synchronous_Barriers,
            No_Dependence => Ada.Task_Attributes,
            No_Dependence => System.Multiprocessors.
                                         Dispatching_Domains);
```

Bibliography

[1] Luis Almeida and Paulo Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 95–103, New York, NY, USA, 2004. ACM Press.

[2] Alejandro Alonso, Juan A. de la Puente, Juan Zamorano, Miguel A. de Miguel, Emilio Salazar, and Jorge Garrido. Safety concept for a mixed criticality on-board software system. *IFAC-PapersOnLine*, 48(10):40 – 245, 2015.

[3] AMD. AMD Ryzen Processors Catalog. `http://www.amd.com/en/ryzen`. Accessed: 2017-10-24.

[4] James H Anderson and Anand Srinivasan. Early-release fair scheduling. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 35–43. IEEE, 2000.

[5] James H Anderson and Anand Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. In *Real-Time Systems, 13th Euromicro Conference On, 2001.*, pages 76–85. IEEE, 2001.

[6] Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 193–202. IEEE, 2001.

[7] Björn Andersson and Jan Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 337–346. IEEE, 2000.

[8] Björn Andersson and Jan Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 33–40. IEEE, 2003.

[9] *Ada Reference Manual, ISO/IEC 8652:2012(E) with COR.1:2016*, 2016.

[10] N. C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.

[11] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Deadline monotonic scheduling theory. In Luc Boullart and Juan A. de la Puente, editors, *Real-Time Programming 1992. Proceedings of the IFAC/IFIP Workshop*. Pergamon Press, 1992.

[12] Alan F. Babich. Proving total correctness of parallel programs. *IEEE Transactions on Software Engineering*, (6):558–574, 1979.

[13] T. P. Baker. A stack-based resource allocation policy for real-time processes. In *[1990] Proceedings 11th Real-Time Systems Symposium*, pages 191–200, Dec 1990.

[14] T. P. Baker. Stack-based scheduling for realtime processes. *Real-Time Systems*, 3(1):67–99, January 1991.

[15] T. P. Baker and A. Shaw. The cyclic executive model and Ada. *Real-Time Systems*, 1(1), 1989.

[16] Theodore P Baker. An analysis of edf schedulability on a multiprocessor. *IEEE transactions on parallel and distributed systems*, 16(8):760–768, 2005.

[17] Theodore P Baker and Sanjoy K Baruah. Schedulability analysis of multiprocessor sporadic task systems. *Handbook of Real-Time and Embedded Systems*, 2007.

[18] Sanjoy K Baruah. Resource sharing in EDF-scheduled systems: A closer look. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 379–387. IEEE, 2006.

[19] Sanjoy K Baruah and John Carpenter. Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. *Journal of Embedded Computing*, 1(2):169–178, 2005.

[20] Sanjoy K Baruah, Neil K Cohen, C Greg Plaxton, and Donald A Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[21] Sanjoy K Baruah, Johannes E Gehrke, and C Greg Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pages 280–288. IEEE, 1995.

[22] William A. Beech, Douglas E. Nielsen, and Jack Taylor. *AX.25 Link Access Protocol for Amateur Packet Radio*, 1998. Version 2.2.

[23] Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 149–160. IEEE, 2007.

[24] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Improved schedulability analysis of EDF on multiprocessor plat-

forms. In *Real-Time Systems, 2005.(ECRTS 2005). Proceedings. 17th Euromicro Conference on*, pages 209–218. IEEE, 2005.

[25] Alessandro Biondi, Björn B Brandenburg, and Alexander Wieder. A blocking bound for nested fifo spin locks. In *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pages 291–302. IEEE, 2016.

[26] Aaron Block, Hennadiy Leontyev, Bjorn B Brandenburg, and James H Anderson. A flexible real-time locking protocol for multiprocessors. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 47–56. IEEE, 2007.

[27] Björn B Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[28] Björn B Brandenburg. A fully preemptive multiprocessor semaphore protocol for latency-sensitive real-time applications. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 292–302. IEEE, 2013.

[29] Bjorn B Brandenburg and James H Anderson. Optimality results for multiprocessor real-time locking. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 49–60. IEEE, 2010.

[30] Björn B Brandenburg, John M Calandrino, and James H Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Real-Time Systems Symposium, 2008*, pages 157–169. IEEE, 2008.

[31] Alan Burns. The Ravenscar profile. *Ada Letters*, XIX(4):49–52, 1999.

[32] Alan Burns and Sanjoy Baruah. Migrating mixed criticality tasks within a cyclic executive framework. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 203–216. Springer, 2017.

[33] Alan Burns, Robert I Davis, P Wang, and Fengxiang Zhang. Partitioned EDF scheduling for multiprocessors using a C=D task splitting scheme. *Real-Time Systems*, 48(1):3–33, 2012.

[34] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the Ada Ravenscar profile in high integrity systems. *Ada Letters*, XXIV:1–74, June 2004.

[35] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages.* Addison-Wesley, 4th edition, 2009.

[36] Alan Burns and Andy J Wellings. A schedulability compatible multiprocessor resource sharing protocol–MrsP. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 282–291. IEEE, 2013.

[37] John M Calandrino, James H Anderson, and Dan P Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Real-time Systems, 2007. ECRTS'07. 19th Euromicro Conference On*, pages 247–258. IEEE, 2007.

[38] John M Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. LITMUSˆ RT: A testbed for empirically comparing real-time multiprocessor schedulers. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 111–126. IEEE, 2006.

[39] Sebastiano Catellani, Luca Bonato, Sebastian Huber, and Enrico Mezzetti. Challenges in the implementation of MrsP. In

Juan Antonio de la Puente and Tullio Vardanega, editors, *Reliable Software Technologies – Ada-Europe 2015: 20th Ada-Europe International Conference on Reliable Software Technologies, Madrid Spain, June 22-26, 2015, Proceedings*, pages 179–195. Springer International Publishing, 2015.

[40] Francisco J. Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Code Lo, and Dorin Maxim. PROARTIS: Probabilistically analyzable real-time systems. *ACM Trans. Embed. Comput. Syst.*, 12(2s):94:1–94:26, May 2013.

[41] Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE*, volume 3, page 10190. Citeseer, 2003.

[42] Yi-Hsiung Chao, Shun-Shii Lin, and Kwei-Jay Lin. Schedulability issues for EDZL scheduling on real-time multiprocessor systems. *Information Processing Letters*, 107(5):158–164, 2008.

[43] Hyeonjoong Cho, Binoy Ravindran, and E Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 101–110. IEEE, 2006.

[44] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.

[45] Sébastien Collette, Liliana Cucu, and Joël Goossens. Algorithm and complexity for the global scheduling of sporadic tasks on multiprocessors with work-limited parallelism. *RTNS'07*, page 123, 2007.

[46] Sébastien Collette, Liliana Cucu, and Joël Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106(5):180–187, 2008.

[47] Alfons Crespo, Alejandro Alonso, Marga Marcos, Juan A. de la Puente, and Patricia Balbastre. Mixed-criticality in control systems. In Edward Boje and Xiaohua Xia, editors, *Proc. 19th IFAC World Congres*, pages 12261–12271. IFAC-PapersOnLine, 2014.

[48] Javier Cubas, Assal Farrahi, and Santiago Pindado. Magnetic attitude control for satellites in polar or sun- synchronous orbits. *Journal of Guidance, Control, and Dynamics*, 38(10):1947–1958, aug 2015.

[49] J. Daintith and E. Wright. *A Dictionary of Computing.* Oxford Paperback Reference. OUP Oxford, 2008.

[50] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computer Surveys*, 43(4):35:1–35:44, October 2011.

[51] Juan A. de la Puente, Juan Zamorano, and Alejandro Alonso. UPMSAT-2 software. requirements baseline — Software System Specification. Technical report, Universidad Politécnica de Madrid, October 2014. Version 2.2.

[52] Juan A. de la Puente, Juan Zamorano, Alejandro Alonso, Jorge Garrido, Emilio Salazar, and Miguel A. de Miguel. Experience in spacecraft on-board software development. *Ada User Journal*, 35(1):55–60, March 2014.

[53] UmaMaheswari C Devi and James H Anderson. Tardiness bounds under global edf scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189, 2008.

[54] Sudarshan K Dhall and Chung Laung Liu. On a real-time scheduling problem. *Operations research*, 26(1):127–140, 1978.

[55] Jeff Edmonds and Kirk Pruhs. Scalably scheduling processes with arbitrary speedup curves. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 685–692. SIAM, 2009.

[56] Dario Faggioli, Giuseppe Lipari, and Tommaso Cucinotta. The multiprocessor bandwidth inheritance protocol. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 90–99. IEEE, 2010.

[57] Nathan Fisher and Sanjoy Baruah. The global feasibility and schedulability of general task models on multiprocessor platforms. In *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, pages 51–60. IEEE, 2007.

[58] European Cooperation for Space Standardization. Ground systems and operations - Telemetry and telecommand packet utilization ECSS-E-70-41C, 2016.

[59] Peter Fortescue, Graham Swinerd, and John Stark. *Spacecraft Systems Engineering.* Wiley, 4 edition, 2011.

[60] Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. In *Real-Time Systems, 2008. ECRTS'08. Euromicro Conference on*, pages 13–22. IEEE, 2008.

[61] Shelby Funk, Joel Goossens, and Sanjoy Baruah. On-line scheduling on uniform multiprocessors. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 183–192. IEEE, 2001.

[62] Shelby Funk and Vijaykant Nanadur. LRE-TL: An optimal multiprocessor scheduling algorithm for sporadic task sets. In *17th International Conference on Real-Time and Network Systems*, pages 159–168, 2009.

[63] Paolo Gai, Marco Di Natale, Giuseppe Lipari, Alberto Ferrari, Claudio Gabellini, and Paolo Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*, pages 189–198. IEEE, 2003.

[64] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*. IEEE Computer Society, 2001.

[65] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, 1979.

[66] Jorge Garrido, Daniel Brosnan, Juan A. de la Puente, Alejandro Alonso, and Juan Zamorano. Analysis of WCET in an experimental satellite software development. In Tullio Vardanega, editor, *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 81–90. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012.

[67] Jorge Garrido, Juan A. de la Puente, Juan Zamorano, Miguel A. de Miguel, and Alejandro Alonso. Timing analysis tools in a model-driven development environment. *IFAC-PapersOnLine*, 50(1):5875 – 5880, 2017. 20th IFAC World Congress.

[68] Jorge Garrido, A Juan, and Juan Zamorano. MrsP on semi-partitioned systems. *28th Euromicro Conference on Real-Time Systems (ECRTS16) - WiP Session proceedings*, page 4, 2016.

[69] Jorge Garrido, Juan Zamorano, Alejandro Alonso, and Juan A. de la Puente. Evaluating MSRP and MrsP with the multiprocessor Ravenscar profile. In Johann Blieberger and Markus Bader, editors, *Reliable Software Technologies — Ada-Europe 2017*, pages 3–17. Springer, 2017.

[70] Jorge Garrido, Juan Zamorano, and Juan A. de la Puente. Static analysis of WCET in a satellite software subsystem. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 87–96, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[71] Jorge Garrido, Shuai Zhao, Alan Burns, and Andy Wellings. Supporting nested resources in MrsP. In Johann Blieberger and Markus Bader, editors, *Ada-Europe International Conference on Reliable Software Technologies*, pages 73–86. Springer, 2017.

[72] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–15. IEEE, 2013.

[73] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-time systems*, 25(2):187–205, 2003.

[74] Grupo de Sistemas de Tiempo Real y Arquitectura de Servicios Telematicos. GNATforLEON / ORK+ User Manual. http://www.dit.upm.es/ str/ork/documents/opm-2.1.0.pdf.

[75] P. K. Harter. Response times in level-structured systems. *ACM Tr. on Computer Systems*, 5(3), 1987.

[76] Philip Holman and James H Anderson. Adapting pfair scheduling for symmetric multiprocessors. *Journal of Embedded Computing*, 1(4):543–564, 2005.

[77] IDR. UPMSat-2 website. `http://www.idr.upm.es/tec_espacial/upmsat2/01_UPMSAT2.html`. Accessed: 2017-11-03.

[78] Intel. Intel Core X-series Processors. `https://ark.intel.com/products/series/123588/Intel-Core-X-series-Processors`. Accessed: 2017-10-24.

[79] ISO. *Ada Reference Manual ISO/IEC 8652:1995(E)/TC1(2000)/AMD1(2007)*, 2007. Available at `http://www.adaic.com/standards/ada05.html`.

[80] David S Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[81] David S Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8(3):272–314, 1974.

[82] Mathai Joseph and Paritosh K. Pandya. Finding response times in real-time systems. *BCS Computer Journal*, 29(5):390–395, 1986.

[83] Shinpei Kato and Nobuyuki Yamasaki. Global EDF-based scheduling with efficient priority promotion. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA'08. 14th IEEE International Conference on*, pages 197–206. IEEE, 2008.

[84] Shinpei Kato and Nobuyuki Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 23–32. IEEE, 2009.

[85] Shinpei Kato, Nobuyuki Yamasaki, and Yutaka Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on*, pages 249–258. IEEE, 2009.

[86] Beatriz Lacruz, Jorge Garrido, Juan Zamorano, and Juan A. de la Puente. Análisis de herramientas de generación automática de código. In Javier Gutiérrez and Michael González Harbour, editors, *Actas del V Simposio de Sistemas de Tiempo Real — CEDI 2016*, 2016.

[87] Karthik Lakshmanan, Ragunathan Rajkumar, and John Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on*, pages 239–248. IEEE, 2009.

[88] David Lammers. Intel cancels Tejas, moves to dual-core designs. *EETimes, May 7th*, 2004.

[89] Butler W Lampson and David D Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, 1980.

[90] Hugh C Lauer and Edwin H Satterthwaite. The impact of Mesa on system design. In *Proceedings of the 4th international conference on Software engineering*, pages 174–182. IEEE Press, 1979.

[91] Suk Kyoon Lee. On-line multiprocessor scheduling algorithms for real-time tasks. In *TENCON'94. IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology. Proceedings of 1994*, pages 607–611. IEEE, 1994.

[92] Hennadiy Leontyev and James H Anderson. A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. *Real-Time Systems*, 43(1):60–92, 2009.

[93] Hennadiy Leontyev and James H Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Systems*, 44(1-3):26–71, 2010.

[94] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4), 1982.

[95] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

[96] Chung Laung Liu. Scheduling algorithms for multiprocessors in a hard-real-time environment. *JPL Space Programs Summary*, 2:37–60, 1969.

[97] Aloysius. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. PhD thesis, MIT, 1983.

[98] M.S. Mollison, J.P. Erickson, J.H. Anderson, S.K. Baruah, and J.A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1864 –1871, July 2010.

[99] Tatsuo Nakajima, Yuki Kinebuchi, Hiromasa Shimada, Alexandre Courbot, and Tsung-Han Lin. Temporal and spatial isolation in a virtualization layer for multi-core processor based information appliances. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, pages 645–652. IEEE Press, 2011.

[100] J Carlos Palencia, Michael González Harbour, J Javier Gutiérrez, and Juan M Rivas. Response-time analysis in hierarchically-scheduled time-partitioned distributed systems.

IEEE Transactions on Parallel and Distributed Systems, 28(7), 2017.

[101] Cynthia A Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 140–149. ACM, 1997.

[102] Luis Miguel Pinho and Stephen Michell. Session summary: Parallel and multicore systems. *Ada Lett.*, 36(1):83–90, July 2016.

[103] Luis Miguel Pinho, Eduardo Quiñonez, and Sara Royuela. Combining the tasklet model with openmp. 2018.

[104] Erhard Ploedereder and Jorge Garrido. Ada-Europe 2017 panel session summary: The future of safety-minded languages. *Ada User Journal*, 38(2):97–98, 2017.

[105] Eduardo Quinones, Emery D Berger, Guillem Bernat, and Francisco J Cazorla. Using randomized caches in probabilistic real-time systems. In *Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on*, pages 129–138. IEEE, 2009.

[106] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Systems Symposium*, 1988.

[107] Ragunathan Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 116–123. IEEE, 1990.

[108] Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A priority Inheritance Approach.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991.

[109] K. Ramamritham, J. A. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Tr. on Software Engineering*, 38(8), 1989.

[110] B. Communautes europeennes. Bureau ESPRIT Randell. *Predictably dependable computing systems*. ESPRIT basic research series. Springer, Berlin, London, 1995. On cover: ESPRIT.

[111] Sara Royuela, Alejandro Duran, Maria A Serrano, Eduardo Quiñones, and Xavier Martorell. A functional safety openmp ˆ{*} for critical real-time embedded systems. In *International Workshop on OpenMP*, pages 231–245. Springer, 2017.

[112] RTCA. *DO-178C/ED12C — Software Considerations in Airborne Systems and Equipment Certification*, 2012.

[113] J. M. Rushby. Design and verification of secure systems. *SIGOPS Oper. Syst. Rev.*, 15:12–21, December 1981.

[114] José Emilio Salazar Marsá. *A framework for mixed-criticality partitioned real-time embedded systems*. PhD thesis, ETSI Telecomunicación, Universidad Politécnica de Madrid, 2015.

[115] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Tr. on Computers*, 39(9), 1990.

[116] Junjie Shi, Kuan-Hsun Chen, Shuai Zhao, Wen-Hung Huang, Jian-Jia Chen, and Andy Wellings. Implementation and evaluation of multiprocessor resource synchronization protocol (MrsP) on LITMUSRT. 13th Workshop on Operating Systems Platforms for Embedded Real-Time Applications, 2017.

[117] Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Real-Time Systems, 2008. ECRTS'08. Euromicro Conference on*, pages 181–190. IEEE, 2008.

[118] David Simchi-Levi. New worst-case results for the bin-packing problem. *Naval Research Logistics*, 41(4):579, 1994.

[119] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information processing letters*, 84(2):93–98, 2002.

[120] Hiroaki Takada and Ken Sakamura. Real-time scalability of nested spin locks. In *Real-Time Computing Systems and Applications. Proceedings., Second International Workshop on*, pages 160–167. IEEE, 1995.

[121] Hiroaki Takada and Ken Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 134–143. IEEE, 1997.

[122] leaded by Ignacio Da Riva institute UPMSat-2 research groups. UPMSat-2 selection of research publications. https://www.researchgate.net/project/MUSE-Master-in-Space-Systems-at-Universidad-Politecnica-de-Madrid-UPM.

[123] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 239 –243, December 2007.

[124] Yang Wang, Nan Guan, Jinghao Sun, Mingsong Lv, Qingqiang He, Tianzhang He, and Wang Yi. Benchmarking openmp programs for real-time scheduling. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2017 IEEE 23rd International Conference on*, pages 1–10. IEEE, 2017.

[125] Bryan C Ward and James H Anderson. Supporting nested locking in multiprocessor real-time systems. In *24th Euromicro Conference on Real-Time Systems*, pages 223–232. IEEE, 2012.

[126] Bryan C Ward and James H Anderson. Fine-grained multiprocessor real-time locking with improved blocking. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, pages 67–76. ACM, 2013.

[127] Bryan C Ward and James H Anderson. Multi-resource real-time reader/writer locks for multiprocessors. In *Parallel and Distributed Processing Symposium, IEEE 28th International*, pages 177–186. IEEE, 2014.

[128] Alexander Wieder and Björn B. Brandenburg. On spin locks in AUTOSAR: blocking analysis of FIFO, Unordered, and Priority-Ordered Spin Locks. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December 3-6, 2013*, pages 45–56, 2013.

[129] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

[130] Jungwoo Yang, Hyungseok Kim, Sangwon Park, Changki Hong, and Insik Shin. Implementation of compositional scheduling framework on virtualization. *ACM SIGBED Review*, 8(1):30–37, 2011.

[131] S. J. Young. *Real-Time Languages: Design and Development.* Ellis Horwood, Chichester, England, 1982.

[132] Juan Zamorano, Alejandro Alonso, and Juan Antonio de la Puente. Building safety critical real-time systems with reusable cyclic executives. *Control Engineering Practice*, 5(7), July 1997.

[133] Juan Zamorano, Alejandro Alonso, and Juan Antonio de la Puente. Automatic generation of cyclic schedules. In Jacques Skubich, editor, *Real-Time Programming 1997 — Proceedings of the 22nd IFACIFIP Workshop on Real-Time Programming*, 1998.

[134] Juan Zamorano and Jorge Garrido. Schedulability analysis of PWM tasks for the UPMSat-2 ADCS. In J.A. de la Puente and T. Vardanega, editors, *Reliable Software Technologies — Ada-Europe 2015*, volume 9111 of *Lecture Notes in Computer Science*, pages 85–99. Springer Berlin Heidelberg, 2015.

[135] Juan Zamorano, Jorge Garrido, Javier Cubas, Alejandro Alonso, and Juan A. de la Puente. The design and implementation of the UPMSAT-2 Attitude Control System. *IFAC-PapersOnLine*, 50(1):11245 – 11250, 2017. 20th IFAC World Congress.

[136] Fengxiang Zhang and Alan Burns. Schedulability analysis for real-time systems with EDF scheduling. Technical Report YCS 426. Technical report, University of York, 2008.

[137] Fengxiang Zhang and Alan Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Transactions on Computers*, 58:1250–1258, 2009.

[138] Shuai Zhao, Jorge Garrido, Alan Burns, and Andy Wellings. New schedulability analysis for MrsP. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2017 IEEE 23rd International Conference on*, pages 1–10. IEEE, 2017.

[139] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Tr. on Computers*, 36(8), 1987.

[140] W. Zhao, K. Ramamritham, and J. A. Stankovic. Schedul-

ing tasks with resource requirements in hard real-time systems. *IEEE Tr. on Software Engineering*, 13(5), 1987.

[141] Dakai Zhu, Daniel Mossé, and Rami Melhem. Multiple-resource periodic scheduling problem: how much fairness is necessary? In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 142–151. IEEE, 2003.