Institut Supérieur de l'Aéronautique
et de l'Espace

Integration and validation of embedded flight software on space-qualified multicore architectures

Filière SiSy - Stage de Fin d'Études

Report

Student(s) :
Joan MOYA RIERA

Supervisor(s) :
Javier FERNANDEZ SALGADO
Pedro BARRIOS GARCÍA

October 10, 2022

Abstract

In the recent decades, the importance of software on space missions has notably increased, reflecting the need to integrate advanced on-board functionalities. With multicore processors being lately introduced to host critical high-performance applications, the complexity to validate software has significantly raised with respect to single core architectures. While there has been a big step forward in avionics after the publication of the CAST-32A paper, the ECSS-E-ST-40C software engineering standard used by the European Space Agency (ESA) is still not providing validation support for multicore processors. Hence, it is expected that standardising guidelines to develop software on such platforms will become a recurring topic in the industry to match the demands of future space exploration missions.

Contents

List of Figures

List of Tables

1 Introduction

In order to conclude my aerospace and telecommunication engineering studies, I did a 6-month internship spanning from the beginning of April until the end of September, within the framework of the flight software systems section (TEC-SWF) of the European Space Agency (ESA/ESTEC). The TEC-SWF is responsible for providing support to all projects of the programme directorates in the area of flight software development, verification and validation operations.

Month	**Tasks**	**Documents**
April (remotely)	Review of objectives and bibliography	Report v.1 (state of the art)
May (remotely)	Introduction of the activity and expected results	Report v.2 (objectives and hypotheses)
June (on-site)	Board configuration and software implementation	Report v.3 (design of the multicore testbench)
July (on-site)	Performance analysis methodology	Report v.4 (instrumentation of the code and processing)
August (on-site)	Identification of interfering channels and mitigation	Report v.5 (validation of mitigation mechanisms)
September (on-site)	Conclusions and validation of hypotheses	Report v.6 (results and conclusions)

Table 1: Structure of the tasks and the documents to deliver during the internship

Due to the COVID-19 pandemic, the internship was unable to be done entirely in presence. During April and May the work was carried out remotely and in June I moved to the Netherlands to work four months on-site. Following this organization, different milestones and deliverables were defined taking into account the limitations of not being in presence at ESTEC. In Table 1 it is shown the structure of the work per month with the associated tasks and deliverables. Various meetings were scheduled every week with the supervisors to provide iterative feedback and a correction phase was set at the end of each month to verify the completeness of each activity. Concerning the presentation of the results, a first talk in front of an evaluation jury at ISAE-SUPAERO, and a second one to the TEC-SWF section in the scope of an OBOSO to conclude the internship.

In view of the specific tasks to perform and the expected deliverables, the report has been organized as follows: in section 2, it is defined the state of the art to understand the challenges related to verification and validation for multicore processors and the existing work on multicore certification is reviewed. In section 3, the design process of a multicore software testbench for a space use-case is described, together with the tools and hardware that has been used to launch and analyse the experiments. In section 4, the results of the different experiments are discussed, including the identification of interfering channels, the analysis of mitigation mechanisms and the validation of real-time requirements. Finally, in section 5, the goals of the activity are reviewed and the hypotheses are validated.

1.1 Context

Since the earliest days of space exploration, several missions have been doomed by software-related causes (Mariner I, Ariane V, Mars Polar Lander...). For a space mission to succeed, and in general for all critical systems, it is essential to conduct rigorous software development processes [17] to avoid regression bugs and ensure that the desired functionalities are provided with negligible probability of failure. Following the evolution in software technology, space agencies have been able to face the criticality of this sector by means of strict verification and validation (V&V) activities [16]. Thereby, spacecrafts have been enabled to safely operate and recover from faulty events and possible hardware defects induced by the harsh outer space environment.

In the recent decades, the impact of software on space systems has increased significantly, reflecting the need of on-board autonomy (e.g. high-precision landings, autonomous navigation or demanding orbital rendezvous), as well as the increased amount of mission data to be collected and processed. Moreover, the introduction of innovative machine learning methods are starting to allow spacecrafts to make their own intelligent decisions for both flight control and payload subsystems.
Due to the rapid growth in algorithmic complexity, modern space applications can compromise the mission in terms of size, weight and power consumption (SWaP) when they run on existing space-qualified hardware [18]. To solve this problem, the industry has set its focus on the use of embedded processors with multiple cores [14]. The first generation of satellites for Earth observation ran on single core processors with 16 kilobytes of memory, while the subsequent generation of telecommunication satellites embarked around 1 or 2 megabytes. Nowadays, the spacecrafts managed by ESA run on multicore versions of space-qualified LEON processors with up to 512 megabyte memory spaces.

Despite the improvements offered by this technology, having multiple cores poses significant problems for V&V activities that ultimately increase development costs and complexity. In particular, when tasks execute simultaneously on a multicore processor, they need to share a certain amount of resources, creating variations on the execution times that are difficult to predict. Consequently, having an accurate model to analyse if a particular scheduler is able to fulfill timing requirements is much harder than in single processors, which have been successfully used in critical applications for several years.
The trend of using multicore processors has become very attractive to increase the on-board performance and enable new spacecraft capabilities. For that reason, the community has recently started tailoring standards for these architectures. However, space applications still run on a unique core, as nowadays there is no sufficient guidance to address multicore verification and validation issues [21].

Because software components running on the same platform interfere each other, the most common practice to provide co-runner interference isolation between software components has been time-space partitioning (TSP). It enables following an incremental validation and integration approach to reduce V&V development times and and costs. Unfortunately, as this leads to a significant waste of the total capacity, other options may be considered, such as symmetric multiprocessing (SMP) to exploit the availability of other cores.

To verify and validate software functions that have been integrated with SMP, it is crucial to take into account the effect of interference in terms of resource contention and task migrations. Existing work on multicore V&V focuses on the need to characterise interference channels and apply mitigation techniques so that predictability can be recovered over the application's behaviour. Therefore, sending multicore-based applications into space appears to rely on the accurate identification of worst-case interference scenarios, so that more precise models can be designed to make a better and reliable use of the available resources.

1.2 Objectives

The aim of this work is to study possible solutions for flight software V&V in multicore processors within the scope of the ECSS standards and prove that the Leon 4 GR740 board provides sufficient mechanisms to mitigate inter-core interference in order to ensure the compliance of hard real-time constraints. The following points describe in further detail the contribution that is expected to be provided at the end of this activity:

1. Develop a testbench using specific algorithms from existing off-the-shelf benchmarks to emulate a real space application.
   Results: Selection of the algorithms included in the testbench, definition of the task model and timing requirements.

2. Integrate the application's software components in SMP configuration using the space pre-qualified version of RTEMS 6.
   Results: Configuration of the setup and integration of the software components.

3. Study possible alternatives to measure the performance of the designed application minimising the overhead introduced in the system.
   Results: Instrumentation of the software to extract timestamps and core-specific events and code scripts to post-process the information.

4. Following the guidelines proposed in the CAST-32A paper, identify the interfering channels and propose mitigation solutions.
   Results: List of the shared resources of the GR740 adding interference to the system and evaluation of possible mitigation mechanisms.

5. Study the response of the mitigation mechanisms and provide proof of evidence that the designed application can be partially validated through the CAST-32A.
   Results: Ensure the compliance of hard real-time requirements using an empirical confidence-based approach.

1.3 Hypothesis

The ECSS-E-ST-40C is the software engineering standard of the ECSS family, which specifies the required deliverables to achieve software certification for space missions managed by ESA. This standard has been developed specifically for single core processors and therefore, it does not support software running on a multicore system.

According to the ECSS-E-ST-40C, the verification of timing constraints needs to be done through schedulability analysis, respecting specific margins defined between the customer and the supplier. However, multicore processors are complex to accurately model, as well as the runtime behaviour of the software application. Consequently, safety margins need to be further increased to account for a bigger set of possible pathological cases. Overly-pessimistic margins imply using a small fraction of the total computational capacity, which ultimately questions the necessity of using multicore processors for critical applications.

During the ECSS's software development life cycle, verification of timing requirements needs to be performed analytically. Because the standard only lists scheduling models for single core processors (cyclic, preemptive, Ravenscar and partitioned systems) it may not be possible to apply analytic methods when more than one core is employed. Instead, measurement-based or hybrid tools can be good alternatives to perform timing analysis on multicore systems. In multicore architectures, the presence of inter-core interference compromises the timing composability and compositionality properties of an application, which eliminates the possibility of having an incremental development. This problem has been solved by applying time-space partitioning techniques, as it is commonly done nowadays in most applications that are based on an Integrated Modular Avionics (IMA) architecture.

Nonetheless, time-space partitioning does not fully exploit the computational capacity available on a multicore system. To make a more efficient use of the resources, SMP may be employed to implement applications, at the expense of having to deal with more exhaustive and complex validation activities. The CAST-32A position paper [19] proposes a set of guidelines in order to have multicore certification for avionics applications, not only for platforms with robust partitioning (IMA-TSP) but also for those without. Identifying channels introducing interference in the system becomes key in order to apply the correct mitigation mechanisms to increase the predictability of the application.

Following these statements, the hypothesis of this work is written as follows: Space applications based on SMP architectures may benefit from the guidelines proposed in the CAST-32A, in order to provide proof of evidence that the software timing requirements are fulfilled without compromising the integrity of the mission.

2 State of the Art

In the following sections the elements that will be taken into consideration to implement the multicore validation benchmark and analyse the outcome of the experiments are presented.

2.1 Critical real-time applications in multicore processors

One of the most significant changes in the sector of critical systems has been the introduction of multicore processors [14]. The increased performance and the reduction on power consumption that characterises this technology is crucial to meet the computational needs of modern space applications. However, extending single core to multicore reduces the predictability of the system's behaviour, which ultimately increases the complexity of verification and validation activities.

While verification activities are performed to check the consistency and completeness of each process in the product development cycle, validation consists in verifying that the developed product meets the specified requirements.

Following the impossibility to guarantee program execution determinism, multicore architectures have been considered inappropriate for critical applications. Nowadays, as software complexity and hardware computational capacity grow at a strong pace, the lack of guidelines to develop critical software on multicores limits the evolution towards certifiable space systems with more advanced functionalities.

2.1.1 From single core to multicore platforms

Multicore technology was developed to achieve efficiency through parallel processing over the single core sequential execution. Parallel processing is the simultaneous use of more than one core to execute an application in order to speed up its execution time. To enhance the performance on single core processors, higher frequencies are required as the workload increases. However power dissipation constraints have limited the maximum achievable frequency and as the number of on-chip transistors has continued to grow exponentially following Moore's Law, nowadays there are very few chips that have clock speeds exceeding a few gigahertz.

Rather than putting efforts to develop faster cores, the manufacturing trend switched to multicores to avoid the power consumption problem and increase the processing efficiency. There are two types of multicore architectures, namely homogeneous and heterogeneous. In the first one, all the cores are identical and have the same features (message passing system, caches, shared memory...). Otherwise, in heterogeneous systems the cores can have independent architectures, which allows for specialized processing capabilities to handle particular tasks. At software level, one can distinguish among two different multiprocessing classes. On the one hand, symmetric multiprocessing (SMP) runs a single operating system (OS) with a shared memory for all the cores involved. This environment enables load balancing as processes can run simultaneously on different cores, as decided by the scheduler. On the other hand, asymmetric processing (AMP) assigns an OS per core, which provides dedicated scheduling control for each of the processing units. In applications that are intended to execute in real time, a Real-Time Operating System (RTOS) is deployed to ensure the compliance of the

timing constraints. Given that SMP and AMP are defined at software level, the programmer can decide which approach is more convenient and which operating systems are going to be employed to develop dedicated applications according to the required needs.
Parallelism is particularly efficient when complex problems can be split into smaller instances that can be executed at the same time. It can be subdivided in Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP). In ILP, the goal is to execute the largest number of instructions per time unit. It can be approached by software, using compiler optimizations or by hardware, employing micro-architectural techniques (pipelining, superscalar architectures, vector instructions, branch predictors...). Alternatively, TLP is intended to run multiple threads at once on different cores. This involves higher programming complexity, as it is required to write balanced applications with well defined subroutines that can fully exploit the multiple cores. A non adequate implementation might lead to significant performance drops, even to the point where a single core platform can provide better results.

In multicore processors, two types of memory systems can be distinguished. While shared memory can be accessed by all the tasks running on different cores, in distributed memory each core has a local partition allocated. It is common in multicores to have a hierarchy of private and shared cache memories. Private caches allow faster accesses for being closer to the core and reduce potential contention, whereas shared caches allow different cores to have availability of the same cache data. The main drawback of private cache memories is the necessity of having data consistency among them, which is commonly known as cache coherence. Incorrect execution can occur if various copies of a given cache block exist in different processors caches, and one of these blocks is modified. Commonly used cache coherence protocols are based on snooping, where the transaction request (read, write, or upgrade) is broadcasted to all the cores. The two main methods to implement snooping protocols are *write-invalidate* and *write-update* (see Figure 2.1). The write-invalidation cache coherence protocol ensures that as soon as a core requests to write to a cache block, that core must remove the copy of the block in any other core's cache that contains the block. When any other core attempts to read the block, it will experience a cache miss, and will have to retrieve the data from the main memory. In write-update protocols, when a write operation is observed on a location that a private cache has a copy of, the cache controller updates its own copy of the written memory block with the new data.

Finally, the interconnect architecture is another aspect to be taken into account on multicore processors. It determines both the scalability of the system and the communication performance and energy consumption. The characteristic topology defines the way switches are wired on the board and the routing algorithm specifies how is data routed around the communication network. Routing algorithms extend from deterministic and oblivious methods to adaptive techniques.

All the previously mentioned features of multicore architectures, both hardware and software, have a direct impact on the development of critical applications, their level of predictability and the way they can be validated against existing certification standards. For that reason, it is crucial to understand the underlying hardware architecture in order to safely develop and efficiently run these kind of applications.

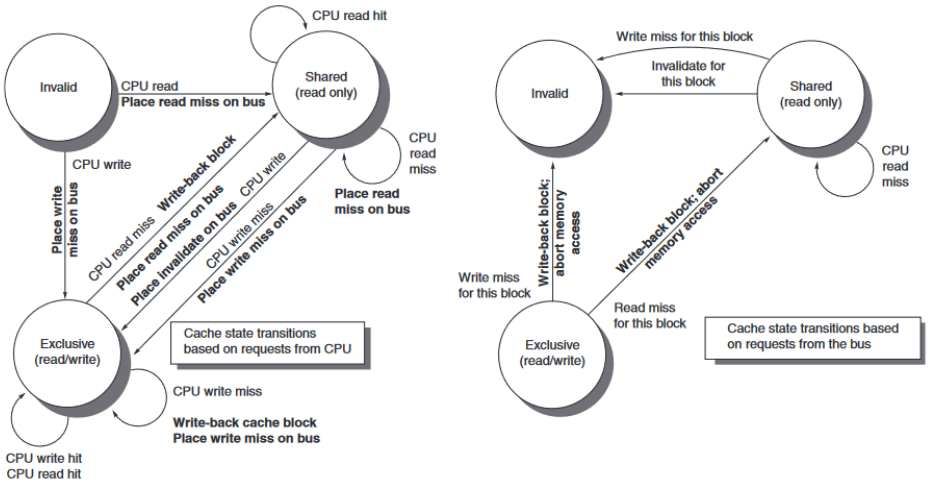

Figure 2.1: Snooping cache coherence protocols [13]

2.1.2 Software verification and validation for critical systems

Software is nowadays one of the main elements of many safety critical systems [16]. A spacecraft flight control system, the control of a nuclear power plant or a blood pressure medical analyser are some of the applications that are included in this domain. The criticality level of a software product is based on the severity of the consequences followed by a software system failure. These categories are defined according to a particular standard. In the case of the ECSS-Q-ST-80C, criticality levels range from A to D, where A indicates a catastrophic event and D implies negligible consequences. According to the level of criticality of a project, more or less strict engineering and product assurance requirements are applied.

In any software project, the development is made following a specific methodology, which is commonly known as software life cycle (see Figure 2.2). Its goal consists in guiding and organising in a structured manner the progress of software throughout its development and maintenance. Processes within the life cycle can be run sequentially, in parallel to reduce the cycle duration or iteratively to reduce project risks. Normally, for safety-critical applications, the whole system is developed to the level of the most critical function, which is a significant source of cost, especially if only a small proportion of the code is critical.

As critical embedded systems are becoming increasingly important in the industry, software verification and validation are of major importance when building high-criticality systems to assure that the specified requirements are fulfilled. Because of the strictness required on these activities, they usually consume more than half the development and assessment budget [17]. Commonly, the industry uses test-based approaches for V&V activities. However, due to the growth of software complexity in the recent years, static techniques such as model checkers or formal methods, inspection or design reviews have become attractive options in this sector. To help with V&V activities, it might be useful to apply independent software verification and validation (ISVV) to reduce risks and costs when integrating software. These complementary activities are performed by independent engineering teams who focus on the validation of non-functional requirements that can lead to software failures. Nevertheless, ISVV is only performed on software projects for criticality levels A and B. Therefore, for

robotic missions where the criticality is lower, this strategy may not be applicable. Moreover, ISVV can duplicate the cost of the software development, which might exceed the project's budget.

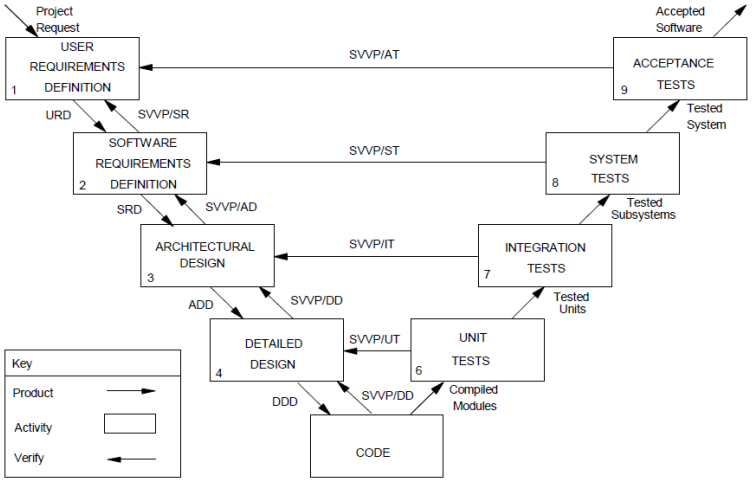

Figure 2.2: Processes of a V-shaped software development life cycle [9]

Challenges related to multicore processors

Using multicore processors for critical applications allows for SWaP optimization of the on-board processing unit to meet modern space systems demands. However, developing software is more complex than in single core processors to achieve the same level of certifiability. The presence of multiple tasks contending for access to shared resources (memory controllers, DDR memory, I/O, cache or buses) introduces unpredictable timing anomalies on the system, even when there is no explicit data or control flow between applications on different cores. These anomalies increase the number of possible execution paths and eliminate the deterministic behaviour of the application. In fact, tests have revealed that a single interfering core can increase the worst-case execution time (WCET) of an application running on another core by a factor of 8x.
Accordingly, timing analysis (see Section 2.1.3), which is key for software verification, becomes problematic. Classical static methods that are employed for single core processors, may no longer be applicable due to the complexity of modelling the execution of an application on a multicore. Alternatively, empirically-based approaches may be useful to increase the accuracy of the estimated behaviour. Because research on timing analysis for multicore processors is still a relatively new topic, timing requirements V&V on space applications calls for overly-pessimistic safety margins to ensure that pathological cases are contemplated, even if that implies losing a great amount of the available on-board performance. At the system level, federated and integrated system architectures have been the two main trends defining how to integrate different functions in avionics systems. While federated architectures map each function to an independent on-board computer, integrated schemes host multiple functions with different levels of criticality on a single platform. Nowadays, using integrated approaches is the standard way to develop avionics applications, as it reduces the SWaP requirements and reduces both maintenance and operating costs.

One way to address multicore interference in integrated architectures consists in testing and analysing the WCET for every application and their worst case utilization of shared resources. However, this becomes very challenging as a single change to any of the co-running applications will require a complete WCET re-verification. Therefore, the main concern for these architectures, such as AUTOSAR and Integrated Modular Avionics (IMA), is to have isolation between applications [22]. This allows developers to run verification and validation activities on each software component independently, which is commonly known as incremental certification.

Because the complexity of the software is proportional to the cost of the V&V activities, specific properties are sought during software design to facilitate the subsequent stages and avoid software regression [4][8]. Software composability implies that each software function deterministically meets its requirements regardless of the presence or absence of other co-running functions. Another software property that is desired to decrease the complexity of verification activities is compositionality, which indicates the possibility to decompose a particular feature of a function in individual units. This is key to determine anomalies on each component of the integration. Moreover, correctness by construction can be used to deliver software with low defect rates and resilient to changes. Its goal is to introduce sufficient precision at each development step so that the correctness of that step can be reasoned through review or using tool support.

In systems with a single core, these properties can be easily achieved because the deterministic execution character is kept when there is no contention. However, in multicore they cannot be guaranteed, thus compromising the correctness of validating each function in isolation without taking into account the other functions. If the integration is not composable, the requirements of a particular function may no longer be fulfilled when others are running in parallel. In addition, identifying the sources that alter the behaviour of the studied function becomes infeasible if the integration is not compositional. Consequently, it might not be adequate for multicore processors to follow an incremental certification approach. A strategy to have composable integrated software is partitioning, which allows to perform V&V activities on independent partitions (see Section 2.2.1) to reduce certification costs. Without partitioning, all software in the system, even low criticality applications, must be certified once a new partition is introduced in the system. Because the execution paths become more unpredictable with more complex architectures, it may be impractical to verify and validate that each function fulfills its requirements for each combination of integrated functions running in parallel.

2.1.3 Verification activities for timing requirements

When a system is said to be *real-time*, it needs not only to validate that all software tasks have a correct functional behaviour, but also to prove that they meet the specified deadlines. Based on the timing constraints, a real-time system can be classified in two classes. In hard real-time systems missing a deadline may have disastrous consequences, while in soft real-time systems, it is allowed to occasionally miss particular deadlines with a certain probability. Embedded spaceflight software is characterized as both safety-critical and real-time. It is a common practice to first analyze these aspects in parallel as two independent sets of constraints and then evaluate their relationships. In the scope of this work, we focus on robotic space vehicles

(satellites, landers, probes or launchers), where the criticality of fulfilling time deadlines on the system is considered to be higher than safety-related aspects. To analyse the timing behaviour, two main verification activities are applied during each software development life cycle process: worst-case execution time (WCET) estimations and schedulability analyses. Taking into account that the software-related technical specification and the system-related requirements baseline contain timing constraints, a test case must be performed to validate the correct behaviour of the application.

Worst-case execution time in multicore processors

Whether submitted to military or civilian authorities, a key piece of evidence for avionics and space certification is the accurate estimation of an application's worst-case execution time. In unicore platforms, the WCET is computed for each task in full isolation, with all the cache lines in dirty status, without preemptions, interruptions, or any co-runners on the other cores. In this scenario, the schedulability analysis considers the worst-case execution pattern, given by the critical instant theorem, and the Real-Time Operating System (RTOS) applies a scheduling policy ensuring that all the deadlines are respected for that case.

The critical instant theorem states that a task is schedulable if all its jobs meet their respective deadlines for the maximum preemption condition, which is given in unicore processors when all the tasks start at the same time instant.

In multicore processors, the WCET does not only depend on the isolated task but also on the scheduling overhead introduced by the RTOS plus the inter-core interference. The eviction and reloading of components like pipelines and private caches after a task is migrated to another core adds unpredictably to the overall delay. Non-preemptive scheduling policies and isolation mechanisms (cache locking and partitioning) can be a solution to substantially avoid these effects.

Another issue is priority inversion, which happens when high priority tasks get blocked by another one of lower priority that has been provided access to a shared resource. One way to eliminate this blocking effect is by setting non-preemptive critical sections, even though a long blocking time for high priority tasks may be introduced. A more common alternative is the use resource access protocols, such as the Priority Inheritance Protocol (PIP) or the Priority Ceiling Protocol (PCP), which bound the blocking time of a task due to resource sharing. With PCP the maximum blocking time is shorter and it avoids interlocking when sharing multiple nested resources (deadlock), as it happens with PIP.
The previous protocols were designed for single core processors and therefore new approaches need to be defined to manage priority inversion on multicore processors. The extension of the PCP is the multiprocessor resource sharing protocol (MrsP) and for the PIP the independence-preserving protocol (OMIP). Significant theoretical results exist for these algorithms and they are supported for most state-of-the-art RTOS.

Worst-case execution time analysis methods have been successfully employed on hardware platforms with a deterministic timing behaviour, such as unicore systems. However, the complexity to provide tight WCET bounds in multicores has led the industry to use highly pessimistic execution time margins when assessing critical applications. From a designer

perspective, this level of pessimism leads to severe under-utilization of the total processing capacity and degrades considerably the quality of schedulability analyses. The main types of worst-case execution time analysis are static, dynamic and hybrid:

- Static timing analysis (STA) tools provide an upper bound estimation for the WCET of a particular code fragment without program execution. This method relies on a good modelling of the underlying hardware and the tasks. The bounds allow safe schedulability analysis of hard real-time systems (e.g. Chronos, OTAWA, Bound-T and aiT).
- Measurement-based timing analysis (MBTA) is an empirical approach based on measuring execution times of short code segments on real hardware or on simulator for some set of inputs. Because of the huge dimensions of the execution state space in multicore processors, there is a high probability of missing longer execution paths, preventing the pathological case to be identified (e.g. Gliwa's Timing Suite T1).
- Hybrid approaches (HTA) employ on-target testing to measure the execution time of short sub-paths in the code and supports offline analyses to build up a model of the code structure. This information is used to compute worst-case execution times in a way that the time variation on individual paths due to hardware effects is captured (e.g. pWCET and RapiTime).

Modelling, and thus predicting, low composability programs executing on multicore systems is a complex task. For that reason, static timing analyses may be excluded for estimating worst-case execution times. Efforts and costs can be reduced with measurement-based analyses but the probability of missing pathological cases may be too high, providing over-optimistic bounds. A potential solution to this problem is to use micro-benchmarks or interference generators, which can be used to characterize the worst-case execution time, taking into account co-runner interference and resource contention. Hybrid analyses provide a balanced WCET estimation between the overly pessimistic result of static analysis and the optimistic values of measurement-based analyses. Probabilistic timing analysis (PTA) methods [7] [11] have recently emerged as attractive alternatives. PTA considers WCET bounds in the same manner as safety-critical systems address reliability: a joint probability distribution of hardware failures and software faults. This approach aims to obtain estimations of WCET bounds ensuring that pathological scenarios may occur with a probability below the one specified by the safety margins. Measurement-Based Probabilistic Timing Analysis (MBPTA) uses PTA and provides WCET upper bounds based on the statistical analysis of execution time measurements. Other hybrid approaches might employ artificial intelligence [12] or instrumentation point graphs [6] to compute estimates.

Scheduling analysis for symmetric multiprocessing

As mentioned previously, one of the fundamental components that has a direct impact on the execution time of a given task is the RTOS. In general, the RTOS is intended to coordinate the simultaneous execution of the tasks in order to meet the specified timing constraints and to manage the allocation of hardware resources. The execution order is determined by the chosen scheduling algorithm and when there is at least one schedule satisfying all the constraints, the system is said to be schedulable. Verifying that a task set can be properly scheduled, which is

commonly known as schedulability analysis, is a proof of evidence requested by certification standards to validate timing requirements, in addition to the estimated WCET bounds.
The entire industry of Real-Time Operating Systems (RTOS) has been built on top of the single core scheduling work of Liu & Layland [15], where they proved that the preemptive Rate Monotonic (RM) scheduling algorithm is able to *optimally* schedule a set of periodic independent tasks with implicit deadlines. The sense of optimality states that any other existing policy will be unable to schedule all the tasks before their deadlines with a higher utilization rate of the processor. In addition, they came up with a theoretical bound for the RM policy (Equation 1), proving that as the number of tasks n tends to infinity, the maximum core utilization U tends to 0.69, while being able to schedule all the tasks of WCET C_i and period T_i.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(\sqrt[n]{2} - 1) \tag{1}$$

Many other scheduling policies were proposed after the publication of Liu & Layland's breakthrough paper. The most popular are the Deadline Monotonic (DM) and the Earliest Deadline First (EDF), as they have been mathematically proven to be optimal for their scheduler categories on single core processors. The DM policy is a *fixed* priority scheduling algorithm that sets priorities in reverse order of the tasks deadlines at compile time. Its schedulability is characterized by the same theoretical utilization bound of the RM policy. The EDF scheduling algorithm is capable to achieve the core's maximum processing capacity and it assigns priorities *dynamically* so that task with the earliest deadline on the current instant of time has the highest priority.

To verify timing requirements, the selected task model needs to permit the design and implementation of a software that is statically analysable. Apart from WCET estimations, scheduling analyses introduce a blocking time parameter for each task, accounting for the system overheads. These include task context switches, mechanisms to access the shared resources, management of mutual exclusion, handling of message queues and interrupt latency. Another aspect to account for is the fact that the optimality of the previously mentioned unicore policies applies on independent periodic task models. If the application presents inter-task dependencies, fixed priority policies may be the best solution. With fixed priorities, it is easy to predict if overload conditions will cause the low-priority processes to miss deadlines, while still respecting the high priority ones. Another option to model task dependencies is through a Directed Acyclic Graph (DAG). DAGs can capture the task behaviour and produce expressive models that can be easily implemented. In addition, when the application is susceptible to aperiodic tasks, a good practice is to convert them into sporadic tasks by setting a minimum inter-arrival time. In such way, sporadic tasks can be treated as periodic and the schedulability analysis can still be accurate enough. Nevertheless, processor utilization might be lost as each sporadic task may be able to be processed before the minimal inter-arrival time. Other techniques that make a better use of the resources exist, even though they don't guarantee the schedulability of aperiodic.

In an attempt to extend unicore task models to multicore, task migrations and their associated effects on hardware resources have to be considered. A simple approach is to include these delays directly on the blocking time of each task. This allows performing the

schedulability analysis without changing the model and applying well-known analytic results. Another option is to account for these effects independently, even though existing task models will no longer apply and so will happen with the associated analytical methods to check schedulability.
The scheduling problem becomes even more complex, given that the critical instant theorem can no longer be applied [10] due to the non-deterministic behaviour of the system. Therefore the absolute worst-case pattern of job arrivals is not known and there is no simple way to find it. The scheduling algorithms proposed for multiprocessor processors can be divided in three main categories according to the management of task migrations:

- In partitioned scheduling each task is allocated offline to a unique core and migration is not permitted. The optimal number of cores to be used for a given task set and where to allocate each task are NP-hard related with partitioning scheduling. Possible sub-optimal solutions to attack these problems are bin-packing heuristics, genetic algorithms or dynamic programming. Tasks allocated to a particular core are scheduled with uniprocessor scheduling algorithms. Consequently, well-known scheduling tests can be performed analytically on each core using existing utilization bounds for specific schedulers.

- With global scheduling tasks are allocated to cores at runtime and migration is permitted. The extension of single core schedulers was originally deprecated in multicores due to the *Dhall effect*, which states that EDF and RM produce unfeasible schedules with a total utilization close to 1, independently of the number of cores. In addition, unicore scheduling algorithms lose the sense of optimality when deployed in multicores. Optimal schedulers have been developed as well for multicore processors (e.g. PFair policies), which are capable to attain the maximum processing capacity. However, they introduce high complexity in terms of implementation and timing overhead, which complicates the schedulability analysis.

- When using hybrid scheduling, different restrictions can be imposed on task migrations. Semi-partitioned and clustered scheduling are two subdivisions of the hybrid approach. In semi-partitioned scheduling some tasks are statically allocated to processors and the rest are split into subtasks, which are allocated using a global scheduling policy. Otherwise, in clustered scheduling a task can only migrate within a predefined subset of cores using affinities.

Some utilization bounds are given in Table 2, being M the number of processors and u_{max} the maximum utilization factor among all the tasks. Because EDF is an optimal uniprocessor scheduling algorithm, a higher utilization bound is not attainable with any other partitioned policy. Notice that partitioned and global scheduling algorithms can not be directly compared, as there might be task sets that are schedulable only with a one of them using the same amount of the processor's capacity.
The main advantages of partitioned scheduling are its simplicity and efficiency. If the task set is fixed, the partitioning approach is the most appropriate solution. However, if tasks can join and leave the system at runtime, it may be necessary to reallocate the tasks in order to not waste computational resources. Global scheduling algorithms allow for an automatic load balancing of the system, as dynamic loads are better managed than in partitioned schemes. In

Scheduler	Task Migration	Task Priority	Utilization Bound
RM	Partitioned	Static	$U \leq M(\sqrt{2}-1)$
EDF	Partitioned	Dynamic	$U \leq \frac{M+1}{2}$
G-RM	Global	Static	$U \leq Mu_{max}$ iff $u_{max} \leq \frac{M}{3M-2}$
G-EDF	Global	Dynamic	$U \leq M-(M-1)u_{max}$
EDF-US[ξ]	Global **(optimal $\xi = \frac{1}{2}$)**	Dynamic	$U \leq \frac{M+1}{2}$
EPDF	Global	Dynamic	$U \leq 3M + \frac{1}{4}$
PF,PD, PD^2	Global **(optimal)**	Dynamic	$U \leq M$

Table 2: Utilization bounds for different multicore scheduling algorithms

addition, the average response time and the number of preemptions is lower. However, global policies suffer from task migration costs, which lead to unpredictable runtime behaviour. Empirical tools might be more appropriate in this case, such as RapiTask, in order to launch schedulability simulations.

2.2 Functional architecture of spaceflight missions

In a typical spacecraft architecture, there are two well-differentiated parts: the avionics platform and the payload (see Figure 2.3). While the payload is mission-specific, the platform is composed by the avionics subsystems that are necessary to provide control of the spacecraft and manage its payload. The central component of the avionics platform is the on-board computer (OBC), which manages all of the spacecraft's activity. Its main elements consist in a processing module, I/O ports to connect to peripheral equipment, autonomous failure management functionalities, on-board time synchronisation and a telemetry/telecommand (TC/TM) module to process and distribute telecommands and send telemetry data to the ground station. It is common to have a data concentrator functional group (RTU), which implements discrete interfaces to control sensors and actuators from the OBC, and a mass memory to store payload or spacecraft telemetry while communication cannot be established.

The OBC executes the application software which implements the vital avionics functions of the spacecraft. These include attitude and orbit control (AOCS), telecommands dispatching, housekeeping telemetry gathering, time synchronisation, failure detection, isolation and recovery (FDIR), thermal and power control, etc.
Concerning the payload side, all the instruments are also implemented by the application software and they vary in number, type and size depending on the particular objectives of the mission. In astrophysics missions for the study of stars, galaxies or black holes, use telescopes, cameras and detectors to collect the radiation emitted by these astronomical objects. For solar system missions, the payload may include cameras, spectrometers or radars to obtain information of the studied bodies (planets, satellites, asteroids, etc...). Recent innovative missions include inertial sensors and laser technology, as it is the case of the Lisa Pathfinder mission which aims at detecting gravitational waves.
In integrated architectures, all the previous functions share the same computational resources which are managed under the supervision of a RTOS, ensuring that timing requirements are

always fulfilled.

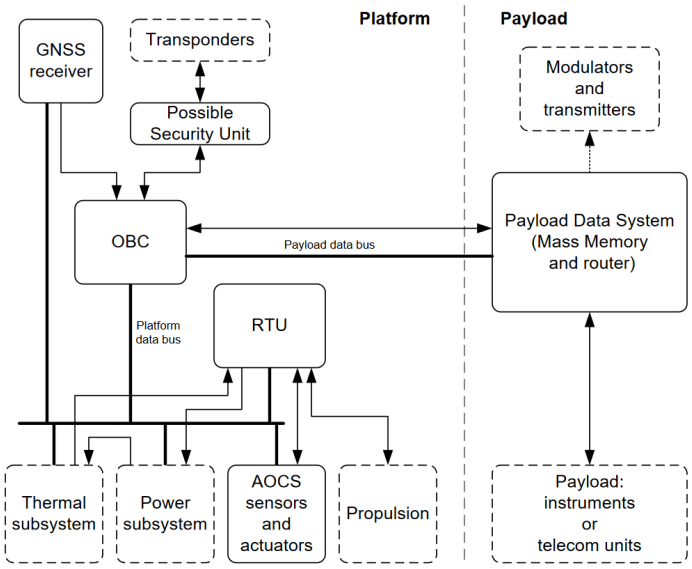

Figure 2.3: Typical spacecraft avionics implementation for ESA missions

2.2.1 Software integration of the spacecraft components

The growth in computational capacity has enabled to have a higher number of functions coexisting on the same platform, giving rise to mixed-criticality (MC) systems. To integrate functions without increasing certification costs, the common approach has been software partitioning. In a multicore system, task migrations are reduced with partitioning and previously certified applications can be reused. The industry has typically made use of software *virtualization* to implement robust partitions. Some microprocessors' instruction sets provide support for hardware virtualization. In such case, guest OSs can be run on isolated hardware, emulating an architecture with independent processing units for each function.

However, if the virtual machine does not fully simulate hardware, a *hypervisor* might be the most desirable option to enable TSP through partial virtualization (para-virtualization). Hypervisors can be classified either as bare-metal (Type 1), if it is placed between the hardware resources and the rest of the system or as hosted (Type 2), if it is run on top of another operating system. The key difference between hypervisor technology and other kind of virtualizations is the performance. Because for critical applications the lowest overhead has to be introduced while maximizing throughput, the virtual machines have to be close to the native hardware, which explains why hypervisors are an attractive solution in this field. Hypervisors apply fixed cyclic scheduling to provide robust time partitioning and avoid concurrency to guarantee determinism. Within each partition, a RTOS schedules tasks with real-time single core or multicore processor policies, depending on the number of cores allocated to that particular partition (see Figure 2.4) and it has hypervisor-specific code to make hypercalls to the layer below.

How functions are allocated into partitions has a direct effect on the overall performance of the integrated system. When having multiple cores allocated to a partition, employing

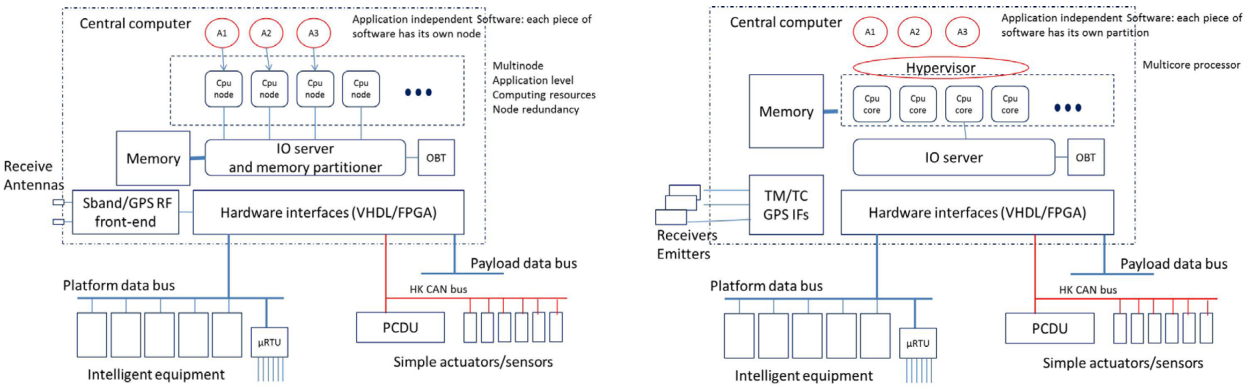

Figure 2.4: Hypervisor-based partitioning schemes [3]

AMP implies that each function is mapped to a particular core and is managed by its own hypervisor-RTOS pair. This decoupling can eventually lead to under-utilizing the total processing capacity due to the lack of load balancing. To make a more efficient use of the computational resources, SMP can be applied within each partition to balance the load across all the cores. Nevertheless, full SMP prevents the developer of knowing which threads will be running on which cores, which is a major risk for deterministic operation. Bound multiprocessing (BMP) provides a solution to this problem, as it statically binds a task to multiple cores, allowing the system architect to tightly control the concurrent execution.

To apply SMP over functions with different levels of criticality within the same partition, specific scheduling approaches have been recently studied. This problem is NP-hard in the strong sense [5] and the fact that no direct isolation mechanisms are provided increases the required pessimism on WCET estimates. Despite the problem's complexity, various global and partitioned mixed-criticality scheduling algorithms have been proposed enabling the use of less pessimistic estimates by operating the system in two modes. When the application is in normal mode, processing capacity is reserved for application tasks based on optimistic estimates, and all task deadlines are guaranteed to be met. When a critical task requires additional processing capacity, the operation mode changes to critical and tasks meet their deadlines depending on their level of criticality. Other options to be considered include applying clustered scheduling, in order to separate into global and partitioned scheduled functions according to their level of criticality. By so doing, it is possible to achieve some degree of isolation and reduce the complexity of implementing mixed-criticality policies. However, these SMP approaches will be more difficult to verify and validate w.r.t. BMP or AMP schemes, which reduce and eliminate respectively, inter-core interference within virtual partitions.

2.2.2 About XtratuM, PikeOS and RTEMS

As it was claimed previously, hypervisors and RTOSs are employed during software integration to assemble together the boot software and the different functions of the application software in a spacecraft. In the scope of missions carried out by the European Space Agency (ESA), the commonly employed hypervisors are XtratuM and PikeOS. These are currently used in several space missions, such as the OneWeb satellite constellation, the ANGELS mission or the Eye-Sat technology mission. Already scheduled missions are going to be launched

in the future carrying this technology. Regarding RTOSs, ESA uses RTEMS, which is the open source real-time executive used on their custom-made radiation-hardened processors. In the following paragraphs a brief description is provided about each one of these software items.

XtratuM is a bare-metal hypervisor provided by fentISS that is qualified for embedded real-time space systems. It uses para-virtualization to create a virtual environment that provides time and space partitioning, enabling applications to share the same hardware platform without interfering with one another. Different operating systems can be used by partitions running over XtratuM, including the RTEMS real-time kernel. The LEON3 and LEON4 processors (SPARCv8) are supported, which is the hardware that is used in the experimental part of this work. Three layers can be distinguished within the architecture of XtratuM. A hardware-dependent layer implements the set of drivers to manage the required hardware. This layer is isolated from the rest by the Hardware Abstraction Layer (HAL), which hides the complexity of the underlying hardware. An internal service layer provides the services that are not available to the partitions. It includes a minimal C library which provides the needed standard C functions and data structures. Lastly, a virtualization service layer supplies the services required to support the service virtualization for partitions. XtratuM comes together with a tool called Xoncrete, which captures the timing behaviour of complex partitioned systems and offers the possibility to run scheduling analyses.

PikeOS is a RTOS developed by SysGo that offers a separation kernel-based Type 1 hypervisor with multiple logical partition types and it is supported by many other OSs including RTEMS. It is available for the SPARCv8 processor architecture and it provides multicore processor support. The PikeOS technology is certifiable by various certification standards including the DO-178C for avionics or the ISO 26262 for automotive. It combines a modular and highly flexible architecture with a variety of certification standards. PikeOS incorporates a scheduler combining time and priority driven scheduling, which offer compliance with hard real-time requirements while still providing best effort scheduling for non-critical tasks. It is possible to switch between multiple pre-configured time partition scheduling. schemes to optimize CPU usage based on the platform operating mode. Concerning the use of multicore processors. PikeOS is certified for the CAST-32A and for the highest level of criticality in DO-178C, among other standards from different domains. Inter-core interference mitigation is provided by shared cache partitioning, fine grained locking and Bandwidth Access Monitoring (BAM) for applications. The CODEO development and configuration tool is based on the Eclipse IDE and it offers a complete environment for embedded systems covering the whole development cycle.

The Real-Time Executive for Multiprocessor Systems or RTEMS is a multi-threaded, single address-space RTOS with no kernel and user space separation. It is capable to operate in an SMP configuration and it provides support for public Application Programming Interfaces (API), such as POSIX. It is used in many embedded devices of independent domains, including space, and it currently supports 18 processor architectures and around 200 Board Support Packages. The SMP configuration of RTEMS has been certified by ESA for ECSS criticality category C and D on the Cobham-Gaisler LEON processors GR740 and GR712RC. The pre-qualified RTEMS version contains specific features of the space profile and it is statically linked to the application code without protection (no distinction between application or user

and kernel memory space). Because RTEMS can get influenced by the application software or the other way around, it is crucial to qualify the RTOS when it is combined with the software application running on top of it.

2.2.3 Space-qualified LEON processors

Outer space is characterized by a harsh operational environment due to the high amount of ionizing radiation, coming from galactic cosmic rays, solar particle events or Van Allen radiation belts. In order for spacecrafts to safely operate in this environment, it is required to design and manufacture platforms that can reduce the damage introduced by radiation. The main damage mechanisms provoked on transistors by radiation are classified into lattice displacement and ionization effects. While the first one induces lasting damage, the second one is usually associated to transient effects, including single-event mechanisms (electrostatic discharges, bit flips, latchups, etc...). Despite having a transient behaviour, these mechanisms can can trigger others that can induce lasting damage on the die. To reduce the impact of ionized particles, there exist two main ways to fabricate radiation-hardened hardware. The physical approach is radiation hardening by process, which involves the use of insulating substrates instead of semiconductor wafers. The logical approach is based on radiation hardening by design, which involves element redundancy, hardened latches and error correcting codes (ECC) to check and potentially correct data that has been corrupted.

As the European space industry consensus was to continue developing on SPARC architecture for the Next Generation Microprocessor (NGMP), ESA followed Cobham Gaisler's initiative to develop and commercialize the radiation-hardened space-qualified GR740 bnboard. This microprocessor has four SPARCv8E LEON4-FT cores, built around five AMBA AHB buses (see Figure 2.5): one 128-bit processor AHB bus, one 128-bit memory AHB bus, two 32-bit I/O AHB buses and one 32-bit debug AHB bus. The four LEON4-FT cores are connected through the processor AHB bus to a shared L2 cache of 2 MiB. The memory AHB bus is placed between the L2 cache and the main external memory interface (SDRAM) and attaches a memory scrubber. The two separate I/O buses connect peripherals such as PCI master/target, PROM/IO memory controller, timers, interrupt controllers, UARTs, GPIO ports, SPI controller, MIL-STD-1553B interface, Ethernet MACs, CAN controllers, and a SpaceWire router.

The GR740 provides high-performance general-purpose processing, support for symmetric and asymmetric multiprocessing and shared resources can be monitored. The GR740 software ecosystem supports the RTEMS SMP qualification package provided by ESA, even though it does not offer memory protection (single address-space). As it is not possible to implement full hardware virtualization, hypervisor services are provided by third parties, including fentISS (XtratuM) and SysGo (PikeOS), to implement time-space-partitioned software. The TSIM3 is the simulator provided by Cobham Gaisler for this microprocessor as it introduces new functionalities w.r.t. previous versions for the GR740 multicore models and a Tcl frontend for easier automation.

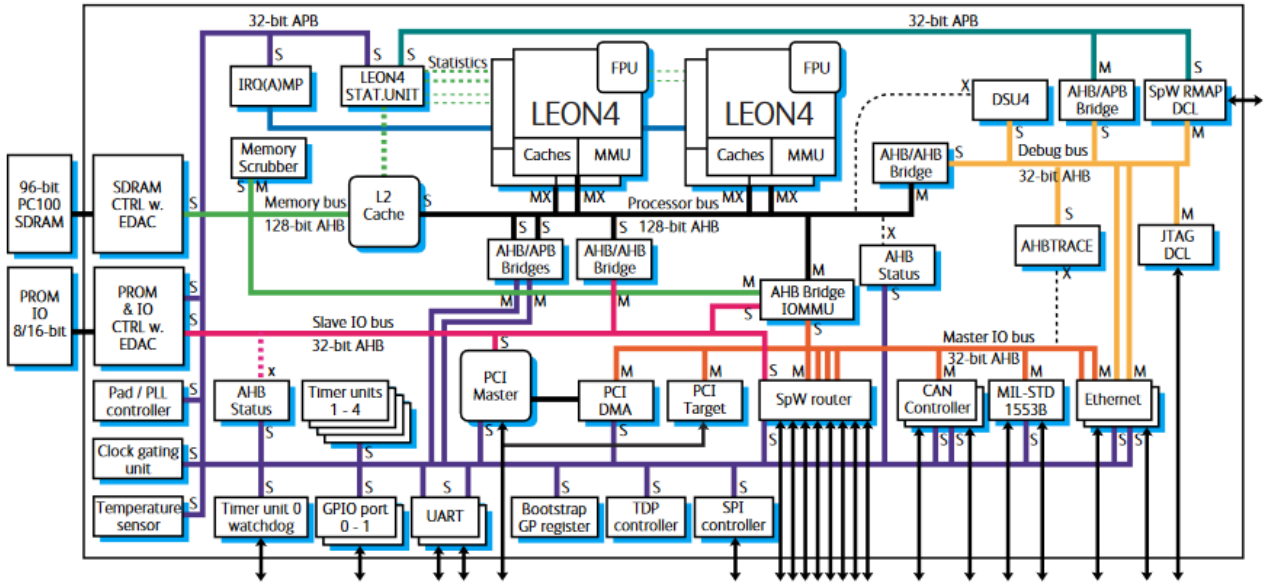

Figure 2.5: Architecture of the GR740 LEON4-FT quad-core [20]

2.3 Standards for critical software engineering

Certification is a procedure by which a third party confirms that a product, process or service is in conformity with a particular standard. For critical systems, certification is obtained through a combination of strict V&V activities and a RAMS programme (Reliability, Availability, Maintainability and Safety), defined to prove the compliance of the standard requirements. During the programme, the life cycle evidence is audited by an external regulatory organisation that is legally empowered to approve the permission for the system's deployment.

Certification and qualification services vary depending on the application domain. In avionics, the Federal Aviation Administration (FAA), the European Union Aviation Safety Agency (EASA) or the Civil Aviation Authority (CAA) are some of the entities in charge of civilian flight certification. The requirements for assuring avionics designs are extensive, especially for high Design Assurance Levels (DAL), which describe the safety level according to the DO-178B standard. In parallel, the Radio Technical Commission for Aeronautics (RTCA) and European Organisation for Civil Aviation Equipment (EUROCAE) deal with aviation standardisation. Several airborne systems' standards have been developed along the years, among which stand out the DO-178B and the DO-178C for software requirements and the DO-254 for electronic hardware design assurance. In the space domain, the main entities working on standardisation for space systems are the International Standards Organization (ISO), the American Institute of Aeronautics and Astronautics (AIAA) and the European Cooperation for Space Standardization (ECSS). In particular, the ECSS is a collaboration ESA, several national space agencies and European industry associations to develop and maintain a single set of coherent standards for European space activities. Its standards are divided in three branches which are intended to be applied together for management, engineering and product assurance. The most significant standard within the scope of our work is the ECSS-E-ST-40C, which is related to the area of software engineering.

2.3.1 Related work on standards for multicore processing

In existing certified critical applications running on multicore systems, the software has always been statically allocated to execute on a unique core within designated memory space boundaries. For instance, the DO-254 certification standard relies on core deactivation [21] when using multicore processors for airborne electronic hardware. Also, existing guidance in the DO-178C standard from the RTCA only covers the verification process for software running on a single core processor and it does not address multicore verification issues.

In 2012, the EASA conducted a research study called MULCORS, which identified flight certification issues related to multicore platforms. Two years later, the Certification Authorities Software Team (CAST) released the CAST-32A position paper about flight certification for a dual-core processor. The paper was later extended to a higher number of cores in 2016. to identify the potential topics compromising safety, performance, and integrity on a multicore architecture. To supplement the CAST-32A position paper, the FAA and EASA developed the A(M)C 20-193. In this advisory circular, several requirements must be demonstrated to certify the software product. These include the following statements, related to the identification of shared resources and the required mitigation techniques:

- **MCP_Resource_Usage_3**: The applicant has identified the interference channels that could permit interference to affect the software applications hosted on the MCP cores, and has verified the applicant's chosen means of mitigation of the interference.
- **MCP_Resource_Usage_4**: The applicant has identified the available resources of the MCP and of its interconnect in the intended final configuration, has allocated the resources of the MCP to the software applications hosted on the MCP and has verified that the demands for the resources of the MCP and of the interconnect do not exceed the available resources when all the hosted software is executing on the target processor.

Rockwell Collins and Wind River proposed a multi-faceted approach to achieve DO-178C certification evidence with a multi-core processor [18], using TSP with the VxWorks 653 virtualization platform (validated on the latest ARM, Intel, and PowerPC architectures). The configuration The activities were based on analysing interference channels, shared memory mechanisms and the robustness of the virtual partitions. The DO-297 standard provides guidance for integrating robust applications on an IMA architecture, which may be interesting to identify and mitigate inter-partition interference for software certification on multicore. Mercury Systems developed the CIOE-1390 module for helicopters and urban air mobility vehicles, which is considered the first commercially-available module with Intel Atom multicore processors for DO-254 and DO-178 DAL-C certification. The engineering team of Mercury Systems uses test tools including Understand, PCLint, LDRA, and Test RealTime for static and dynamic software coverage analysis.
Moreover, the Multi-Core for Avionics (MCFA) working group, founded in 2008 by NXP, has been key in advancing the use of multicores for DAL A-certified avionics. MCFA includes avionics systems' designers and developers working to migrate avionics from federated architectures built on single core processors to integrated modular architectures using multiple cores.

2.3.2 ECSS standards for software engineering

The ECSS-E-ST-40C is the main standard that defines the principles and requirements applicable to space software engineering within the ECSS structure. Since the very first use of ECSS software engineering standards, several space projects, ranging from full-size satellites to on-board software or ground equipment specific activities have been deployed in compliance with these standards.
The concept of customer–supplier is the fundamental principle of this standard, in which a chain of customer–supplier relationships are extended downwards to the prime contractor and and its subcontractors. Instead of defining how to perform the necessary work, ECSS provides information about what shall be accomplished to certify a space project. Therefore, the idea is to permit suppliers to use their own standards, provided that they comply with the requirements of the ECSS or some customer-defined tailored version. The general requirements for tailoring are defined in the ECSS-S-ST-00. Tailoring for software development constraints takes into account the special characteristics of the software being developed and the development environment. In parallel with the E-ST-40C, the ECSS-Q-ST-80C defines the principles and requirements applicable to space software product assurance. Management standards concerning the software life cycle are also employed for a having successful development campaign.
The ECSS-E-ST-40C standard is applicable to all the elements of a space project, including the space segment, the launch service segment and the ground segment. It covers all the aspects of space software engineering including requirements definition, design, implementation, verification and validation, operation and maintenance. In the standard, these are defined with the specific methods to use in during the development cycle and the expected outputs for each software engineering process.

In the ECSS-E-ST-40C, both the user-defined requirements baseline and the technical specifications (software requirements) have to undergo validation activities to prove that the final product fulfills them. The main documents that have to be delivered concerning the validation of the software product are the validation plan (SValP) and the validation specification (SVS). The objective of the SValP is to describe the approach to the implementation of the validation process against against the requirements baseline and the technical specification. The SVS w.r.t. the technical specification and w.r.t. the requirements baseline specify the test cases, the test procedures and the items that have been tested through analysis, inspection and design of review. The validation results are provided in the software validation report w.r.t. each type of requirements. All these documents belong to the Design Justification File (DJF) and they are part of the milestones that have to reached for specific software development activities.

Verification activities are performed over the requirements baseline, the technical specification, the architectural design, the software detailed design and the code implementation. Test-based validation activities have to be verified as well, including unit and integration testing, validation w.r.t. the technical specification and w.r.t. the requirements baseline. In addition, the verification process requires schedulability analyses for real-time software and technical budget estimations, including the memory consumption, the utilization of the cores and the margins to meet timing deadlines. The previous verification activities are not only done at the integration level but a refinement at lower levels is required. The first document

to be delivered is the software verification plan (SVerP), which describes the approach and the organization aspects to implement the software verification activities. The second and last document concerning this activity is the verification report (SVR) where the gathered results are presented. While the SVerP is a deliverable for the first review, the SVR has to be delivered at each project review along the software development life cycle. As it happens for the validation process documents, the verification documents belong to the DJF.

Looking back to the existing work concerning standards for software development on multicores, it can be claimed that the ECSS-E-ST-40C currently supports activities to only validate software on single core system. A clear example can be found during the description of real-time verification through scheduling analyses. The described computational models in the software engineering handbook, which define the temporal behaviour of the application, only concern task scheduling on single core processors. This handbook does not address partitioned systems, even though a brief explanation is provided about employing IMA and logical partitions, as it is standardized (for a single core) in the ARINC-653. Within ESA's ISVV guide, annex F lists the methods considered for design analysis. These include detailed information about the use of formal methods, inspection, schedulability analysis, WCET computation or static coding analysis, among others. Such methods may result useful to complement the information that is provided in the actual standard but despite that, multicore processing is never considered. A tailored version of the ECSS-E-ST-40C could be useful to provide specific V&V guidelines for multicores, specially in terms of interference mitigation and software integration tips to ease the development process and to have the best possible trade-off between CPU utilization and validation efforts.

3 Design of a space application testbench for multicore

Validating software on multicore systems is starting to become a recurring research topic, as new functionalities are needed to launch more complex missions. To study the multicore validation problem, a software application has been designed with specifically chosen algorithms in order to emulate a real space use-case.
The **Prospect** instrument has been used as reference for the functional and timing requirements. This module is being developed by ESA and it is supposed to be part of the payload of a future lander from NASA. The application software of Prospect is deployed over a distributed system that consists in two independent unicore Leon 2 connected through a SpaceWire. More precisely, each platform runs two different modules: **proSEED** and **proSPA**. While the former is based on a robotic drill that allows for sample extraction, the latter consists in an on-site scientific laboratory to analyse the composition of the soil and store the results. The instrument counts as well with an imaging system to extract high resolution frames.

In general lines, specific functionalities of the Prospect module have been migrated from two single core boards to a unique multicore processor. To that end, a quad-core Leon 4 GR740 has been used as hardware platform and the application has been deployed over RTEMS 6 to run in SMP configuration. Such modification of the mission's architecture would directly impact the total weight and power consumption, allowing for a more integrated design and a higher performance, at the expense of a more exhaustive validation process.

3.1 Benchmarks for embedded space applications

To design the software application, specific algorithms were selected from off-the-shelf benchmarks (see Table 3) to match certain Prospect's functionalities. The functions executed by proSEED and proSPA include the reception and processing of a telecommand, the execution of the control actions associated to the received telecommand, the monitoring of the spacecraft's state and the transmission of telemetry packets.

As described in the documentation of Prospect, the time required to process TC and TM packets is negligible with respect to the processing time of the state machines to control the instruments. For that reason, it was considered that only implementing the control algorithms would provide sufficient insights of the studied problem. Controllers are typically implemented as nested loops, each one executing at a certain frequency. For instance, the proSEED module manages the rotational motion of the drill by two PID feedback controllers: a low frequency loop provides position commands to a higher frequency one that transforms them into input signals for the actuators of the drill.

Two main benchmarks were considered for embedded control applications: **CoreMark** and **AutoBench**. On the one hand, AutoBench is a set of algorithms that allow users to predict the performance of microprocessors in automotive, industrial, and general-purpose applications. It is composed of generic workload tests, basic automotive functions and signal processing algorithms. On the other hand, CoreMark provides functions based on matrix arithmetic and finite state machine operations. Taking into account the implementation structure of control loops (arithmetic, compare and branch operations), the selected algorithms were

Benchmark	Algorithm	Functions	Description
AutoBench	Tooth to spark	Control flow Table look-ups	Real-time processing of air/fuel mixture and ignition timing
AutoBench	Road speed calculation	Control flow Integer arithmetic	Estimation of the road speed based on differences between timer values
AutoBench	Floating point matrix operations	LU decomposition Determinant calculation Matrix cross product	Embedded automotive application which performs a lot of matrix arithmetic
OBPMark	Image calibration and correction	Offset correction Bad pixel correction Radiation scrubbing Gain correction Space-time binning	Pre-processing pipeline for panchromatic imaging in remote sensing applications
OBPMark	Image compression with CCSDS 122.0	Discrete wavelet transform Bitplane encoder	Three-level 2D discrete wavelet transform for lossy image compression

Table 3: Integrated algorithms to reproduce real spacecraft functionalities

Tooth to Spark and Road Speed Calculation from AutoBench [1], which appeared to be the most complete for the searched purpose. These are based on finite state machines that perform common control functions in the automotive domain, specifically by the car's Electronics Control Unit (ECU).

To emulate the operations carried out by the imaging system of the instrument, two algorithms were selected from **OBPMark**, which is a set of benchmarks developed by the Barcelona Supercomputing Center (BSC) in collaboration with ESA [2]. The algorithms implement spacecraft on-board data processing applications, such as image and radar processing, data and image compression, signal processing and machine learning. The selected algorithms were an image processing pipeline and the CCSDS 122.0 recommended standard for image compression, both with configurable frame size.

In addition to the previously mentioned algorithms, others were studied to specifically generate load on particular channels of the board. To stress the floating point unit, a matrix arithmetic benchmark was selected from AutoBench, in order to reproduce the operations that are performed during orbit propagation. Even though they were finally not implemented, two other benchmarks were considered. The first one consisted in RapiDaemons, provided by Rapita Systems, which required a license and a specific trace generator. The second studied approach was based on a set of microbenchmarks developed by the BSC. In this case, they were implemented specifically to stress different levels of the cache memory system. However, this aspect was already covered by the image processing and compression algorithms as they operate on high resolution images that can't be entirely loaded neither in the L1 data cache nor in the L2 cache.

3.2 Real-time implementation on RTEMS SMP

Nowadays, most applications that are embedded on multicore platforms run time-space partitioned software, in order to avoid timing anomalies and have an application as much predictable as possible. In this project, the application is implemented on top of RTEMS 6, to study the effect of inter-core interference with a SMP architecture. As mentioned in the previous section, four tasks are running in parallel: the ones emulating proSEED and proSPA which will be identified as **critical**, and the imaging system and orbital propagation tasks, that will operate as **loads** to add interference to the system.

3.2.1 Task set description and timing requirements

The software application has been designed to run until four frames are processed by the imaging system. To compromise as much as possible the compliance of real-time requirements, several aspects related to the mission have been taken into account to reproduce the heaviest workload that the instrument may need to process:

- The worst case for proSEED is given by the execution of three motion commands: drill rotation, drill translation and sampling operation. For each motion command, a control loop based on a PID controller is executed. The three motion commands need to be executed every 50 ms and the estimated processing time when ran in isolation is 27 ms.
- On its side, the worst case execution time of proSPA occurs when both the actuation of the mechanical systems and the composition analysis of a sample are required. This loop is executed every 100 ms and each one of the operations has to be executed in less than 10 ms. When measured in isolation, the proSPA loop takes 34 ms to be processed.

In table 4 the set of tasks that has been implemented is described, together with the associated benchmark algorithms, the worst case execution time of the task measured in isolation, the period and the deadline.

Task	Functionalities	Benchmark/s	WCET	Period	Deadline
ProSEED	Drill rotation Drill translation Sampling tool	Tooth to spark 3 sequential calls 180 iterations per call	27 ms	50 ms	50 ms
ProSPA	Manifold system Mechanical system MS spectrometer ITM spectrometer	Road speed 4 sequential calls 2000 iterations per call	34 ms	100 ms	50 ms
ImagSYS	Frame processing and compression	Image calibration Image correction CCSDS 122.0	1467 ms	2000 ms	-
OrbPROP	LU decomposition Determinant Cross product	FP arithmetic 10^8 iterations per call	10000 ms	20000 ms	-

Table 4: Implemented set of tasks and their associated real-time requirements

Concerning the loads, the imaging system acquires the frame data from memory, it applies calibration and correction operations and compresses the processed image following the CCSDS 122.0 recommended standard. Each frame consists of 1024 by 1024 pixels, each one of them represented by a 32-bit integer. In addition, the orbit propagation system performs matrix decomposition, determinant calculations and matrix cross-products with floating point variables, after having loaded the data of the input models. These tasks are executed with a period of 2000 ms and 20000 ms respectively and since they are soft real-time functions, no deadline is assigned to them.

3.2.2 Configuration of the RTOS

To properly configure an RTEMS application, it is required to specify certain macros to determine the configuration of the RTOS and how it is going to interact with the Board Support Package. For the pre-qualified version of RTEMS in particular, it is essential to disable the file system and the *newlib* library, as they are not part of the items that have been included in the qualified version. Otherwise, an error is given by the compiler if these settings are not deactivated. The following code snippet shows the macros to configure the application.

```
#include <rtems.h>
#include <rtems/bspIo.h>

/* Definition of attributes and storage size for each RTEMS task */
#define MAX_TLS_SIZE RTEMS_ALIGN_UP( 64, RTEMS_TASK_STORAGE_ALIGNMENT )
#define TASK_ATTRIBUTES RTEMS_DEFAULT_ATTRIBUTES
#define TASK_STORAGE_SIZE RTEMS_TASK_STORAGE_SIZE( MAX_TLS_SIZE +
    RTEMS_MINIMUM_STACK_SIZE, TASK_ATTRIBUTES )

/* Definition of maximum RTEMS objects in the application */
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_MAXIMUM_PROCESSORS 4
#define CONFIGURE_MAXIMUM_TASKS 5
#define CONFIGURE_MAXIMUM_FILE_DESCRIPTORS 0
#define CONFIGURE_MAXIMUM_THREAD_LOCAL_STORAGE_SIZE MAX_TLS_SIZE
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_INIT_TASK_ATTRIBUTES TASK_ATTRIBUTES
#define CONFIGURE_INIT_TASK_INITIAL_MODES RTEMS_DEFAULT_MODES
#define CONFIGURE_INIT_TASK_CONSTRUCT_STORAGE_SIZE TASK_STORAGE_SIZE

/* Unavailable attributes in the pre-qualified version of RTEMS SMP */
#define CONFIGURE_DISABLE_NEWLIB_REENTRANCY
#define CONFIGURE_APPLICATION_DISABLE_FILESYSTEM
#define CONFIGURE_MICROSECONDS_PER_TICK 1000

/* Always defined after all the configuration macros have been set */
#define CONFIGURE_INIT
#include <rtems/confdefs.h>
```

Any application that is implemented on top of RTEMS needs to have an entry-point, which consists in the *Init* function by default, and it is as well configured through specific macros as it can be observed above. This function defines the settings of each task, initializes the data structures and starts the tasks that have been configured.

Before launching the tasks, they need to be configured using the *rtems_task_contruct()* routine. This function takes as input the task identifier and a *rtems_task_config* object, whose attributes will define the specific configuration that the task is going to present. An example is given below to define the configuration structure of a generic task on RTEMS 6.

```
RTEMS_ALIGNED( RTEMS_TASK_STORAGE_ALIGNMENT )
static char task_storage[ TASK_STORAGE_SIZE ];

static const rtems_task_config task_config = {
  .name = rtems_build_name( 'R', 'U', 'N', '1' ),
  .initial_priority = 2,
  .storage_area = task_storage,
  .storage_size = sizeof( task_storage ),
  .maximum_thread_local_storage_size = MAX_TLS_SIZE,
  .initial_modes = RTEMS_DEFAULT_MODES,
  .attributes = TASK_ATTRIBUTES
};
```

Each task is implemented as an infinite loop where the associated functions are called. Before entering the loop, the task calls *rtems_task_set_scheduler()* in order to be assigned to a specific scheduler. To control the cyclic execution of the tasks, the rate monotonic manager of RTEMS has been used. After creating a rate monotonic object within the body of the task, the function *rtems_rate_monotonic_period()* is called at the end of the infinite loop to block the execution until the remaining time of the period is elapsed.

To implement SMP, it is required to define scheduler objects and assign them to a specific core before launching the application. This can be easily done by adding certain configuration macros after having defined the maximum cores of the system. Three different scheduling scenarios have been implemented:

- **Partitioned scheduling**: Each task is dispatched on a unique core controlled by a single scheduler. As this approach does not allow for task migrations, the inter-core interference is purely generated by shared resource contention.

```
#define CONFIGURE_SCHEDULER_EDF_SMP
#include <rtems/scheduler.h>

RTEMS_SCHEDULER_EDF_SMP( sch_0 );
RTEMS_SCHEDULER_EDF_SMP( sch_1 );
RTEMS_SCHEDULER_EDF_SMP( sch_2 );
RTEMS_SCHEDULER_EDF_SMP( sch_3 );

#define CONFIGURE_SCHEDULER_TABLE_ENTRIES \
RTEMS_SCHEDULER_TABLE_EDF_SMP(sch_0,rtems_build_name('s','c','h','0')),
RTEMS_SCHEDULER_TABLE_EDF_SMP(sch_1,rtems_build_name('s','c','h','1')),
RTEMS_SCHEDULER_TABLE_EDF_SMP(sch_2,rtems_build_name('s','c','h','2')),
RTEMS_SCHEDULER_TABLE_EDF_SMP(sch_3,rtems_build_name('s','c','h','3'))

#define CONFIGURE_SCHEDULER_ASSIGNMENTS \
RTEMS_SCHEDULER_ASSIGN(0, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY),
RTEMS_SCHEDULER_ASSIGN(1, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY),
RTEMS_SCHEDULER_ASSIGN(2, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY),
RTEMS_SCHEDULER_ASSIGN(3, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY),
```

- **Global scheduling with 4 cores**: proSEED, proSPA and imagSYS are dispatched with a single priority-based scheduler on cores 1, 2 and 3, while the OrbPROP task is allocated on core 0.

```
#define CONFIGURE_SCHEDULER_EDF_SMP
#include <rtems/scheduler.h>

RTEMS_SCHEDULER_EDF_SMP( sch_0 );
RTEMS_SCHEDULER_EDF_SMP( sch_1 );

#define CONFIGURE_SCHEDULER_TABLE_ENTRIES \
RTEMS_SCHEDULER_TABLE_EDF_SMP(sch_0,rtems_build_name('s','c','h','0')),
RTEMS_SCHEDULER_TABLE_EDF_SMP(sch_1,rtems_build_name('s','c','h','1'))

#define CONFIGURE_SCHEDULER_ASSIGNMENTS \
RTEMS_SCHEDULER_ASSIGN(0, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY),
RTEMS_SCHEDULER_ASSIGN(1, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY),
RTEMS_SCHEDULER_ASSIGN(1, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY),
RTEMS_SCHEDULER_ASSIGN(1, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY)
```

- **Global scheduling with 3 cores**: proSEED, proSPA and imagSYS are dispatched with a single priority-based scheduler on cores 1 and 2, the OrbPROP task is allocated on core 0, and core 3 is left in IDLE state.

```
#define CONFIGURE_SCHEDULER_EDF_SMP
#include <rtems/scheduler.h>

RTEMS_SCHEDULER_EDF_SMP( sch_0 );
RTEMS_SCHEDULER_EDF_SMP( sch_1 );

#define CONFIGURE_SCHEDULER_TABLE_ENTRIES \
RTEMS_SCHEDULER_TABLE_EDF_SMP(sch_0,rtems_build_name('s','c','h','0')),
RTEMS_SCHEDULER_TABLE_EDF_SMP(sch_1,rtems_build_name('s','c','h','1'))

#define CONFIGURE_SCHEDULER_ASSIGNMENTS \
RTEMS_SCHEDULER_ASSIGN(0, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY),
RTEMS_SCHEDULER_ASSIGN(1, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY),
RTEMS_SCHEDULER_ASSIGN(1, RTEMS_SCHEDULER_ASSIGN_PROCESSOR_MANDATORY),
RTEMS_SCHEDULER_ASSIGN_NO_SCHEDULER
```

For the global scheduling configurations, the priorities have been assigned following a rate monotonic approach, meaning that tasks with shorter periods will preempt the execution of other ones having longer periods: proSEED > proSPA > imagSYS > orbPROP.
As it will be shown in section 4, each scheduling configuration will have a particular impact on the amount of interference introduced in the system. Choosing the right scheme will be key to facilitate validation and ensure that all deadlines are respected.

3.3 Launching the software testbench

To extract the execution patterns of the implemented experiments, several steps had to be previously accomplished. Firstly, it was necessary to identify which metrics would provide relevant data and oppositely, which would be irrelevant. Secondly, the measurement mechanisms had to be selected, either hardware or software, to generate the minimum amount of

overhead. Finally, a processing support had to be used in order to generate traces and pull out the useful information. The following points describe the path followed in this project to cover the aforementioned aspects.

3.3.1 Metrics and code instrumentation

To characterize the performance achieved by the software application, different metrics were considered, including the **execution time** of each task and **core-specific events**. Measuring the execution time of each task is achieved with the RTEMS free-running counter, that can be sampled with *rtems_counter_read()*. This counter was read before and after the call to the task's main function to compute the total number of ticks elapsed, which was later transformed into milliseconds, using the the *rtems_counter_frequency()* routine.

To sample specific events on different cores, the L4 statistics unit was used. Since only 16 counters are provided, it was necessary to select them strategically. Given that no I/O devices were involved in the experiments, the events of interest were only related to the memory system and the associated AHB buses. The set up of the counters can be done directly on GRMON or through GDB and they can be configured to reset its value once they are sampled. At the end, the measured events included L1 cache misses for both data and instructions, L2 cache misses, AHB bus read operations and write operations, on cores 1, 2 and 3.

The following code snippet shows how a generic task is instrumented to sample the RTEMS free-running counter and the L4 statistics unit events, every time the loop is re-executed. The global variable *processed_frames* indicates the number of images analysed by the imaging system, which is employed to delete each task once the objective of the application has been achieved. The function *read_L4stat_counters()*, is used to obtain the values stored in the 16 addresses corresponding to each counter's register. As it was explained in section 4, each task has a specific execution period, controlled by the RTEMS rate monotonic manager. Since the counters are task independent and only depend on what is being executed, the function to poll the counters is called within the loop of the fastest task, every 50 ms.

```
uint32_t counter_freq = rtems_counter_frequency() / 1000 ;

while ( processed_frames < 4 ) {

    ticks_start_T1 = rtems_counter_read() ;
    /* Calls to the task's functions */
    ticks_stop_T1 = rtems_counter_read() ;

    read_L4STAT_counters() ;
    ticks_diff_T1 = (ticks_stop_T1 - ticks_start_T1) / counter_freq ;

    status = rtems_rate_monotonic_period( rm_id , 50 ) ;
}
```

3.3.2 Execution of the application

The toolkit that has been used to compile and link the application's source code with RTEMS 6 corresponds to the pre-qualified SMP version. This set of tools allows end-users to qualify

their applications on space-qualified hardware. The target application area is payload data processing and it uses the for the GCC-based cross-compiler provided by the RTEMS Source Builder for the GR740. The RTOS is statically linked to the application using a flat memory model without protection, which means that they share the same memory space.
Therefore, the application software behavior may influence RTEMS and the other way around, which implies that validating each one separately may be prohibitory.

Once the executable file has been generated, GRMON 2 is used to interface with the hardware. A batch configuration file is employed to enable the GNU debugger/GDB, activate profiling and configure the counters of the L4 statistics unit all at once. There are some features from the GR740 board that won't work if clock gating is deactivated, such as the statistics unit. To activate it, it is required to pass the *-c* flag to the command line when launching GRMON.

The execution of the application is controlled through GDB. Several library-compatibility problems were encountered to use the debugger of the pre-qualified toolkit directly on the server where the GR740 was connected. To proceed with the experiments, GDB was ran inside a Docker container with the required dependencies and the ports forwarded from the server. The process of loading the executable on the board and running the application was automatized through a .gdbinit file. To extract both the execution time and the counter values, a breakpoint was set at the end of each task's loop and specific global variables (value_counters and ticks_diff) were printed out.

The printed data was redirected into an output trace file and formatted as shown below, to minimize the parsing complexity and speed up the post-processing of the information.

```
1 Thread 2 hit Breakpoint 4, proSEED_task(arg=0) at src/EDF-partitioned.c:137
2 $1 = 27
3 $2 = {577, 1642, 778, 4454, 302122, 43, 545, 260, 1176, 224232, 0, 209, 0,
     418, 1100, 0}
4
5 Thread 3 hit Breakpoint 5, proSPA_task(arg=1) at src/EDF-partitioned.c:168
6 $3 = 35
7 $4 = {580, 21547, 813, 44270, 249163, 131, 12700, 765, 25662, 361732, 0,
     183, 0, 366, 900, 0}
```

3.3.3 Post-processing

A series of Python scripts have been developed to process the GDB traces, once the execution terminates. The script parses these traces accordingly, sorts the metrics by task names and arranges all the information in a CSV file. After the file has been generated, different functions can be computed on the CSV data, from statistical moments to graphical representations in the form of probability density functions, histograms or boxplots.
As it will be seen in section 4, these will become extremely useful to determine the reduction in inter-core interference provided by specific hardware mechanisms and to claim that a partial software validation is possible following the CAST-32A guidelines.

4 Interference mitigation for partial CAST-32A validation

Aviation standards such as the DO-178C or the ED-12C were written when single core processors were the only solution available for civil aircrafts. The publication of the CAST-32A paper had a notable repercussion on the avionics field, as no guidelines had never been provided before for multicore validation. Regarding the space sector, no institution has yet standardized validation activities to simultaneously use more than one CPU on a multicore.
In this section, a practical study is proposed concerning specific certification objectives of the CAST-32A to determine if a space application in SMP configuration can be partially validated on the GR740. Four main activities have been covered: identification of interfering channels, description of the configuration that will mitigate the interference, verification of the mitigation mechanisms and guaranteeing that each software component has sufficient time to complete its execution.

4.1 Identification of interfering channels

The goal of this activity is to verify the presence of interference on the GR740's shared memory, interconnect, or any other shared resources, as part of the MCP_Resource_Usage_3 objective of the CAST-32A. In first instance, the counters of the L4STAT unit have been used to obtain the execution patterns corresponding to the standalone execution of proSEED and proSPA, which have been later compared with the application running the complete task set to identify the interfering channels.

4.1.1 Standalone execution of the critical tasks

The first tests have consisted in executing in a single core first proSEED and proSPA after with the rest of the cores in IDLE state. In figure 4.1, it can be observed the number of L2 cache memory misses, AHB bus read and write accesses for 200 iterations.

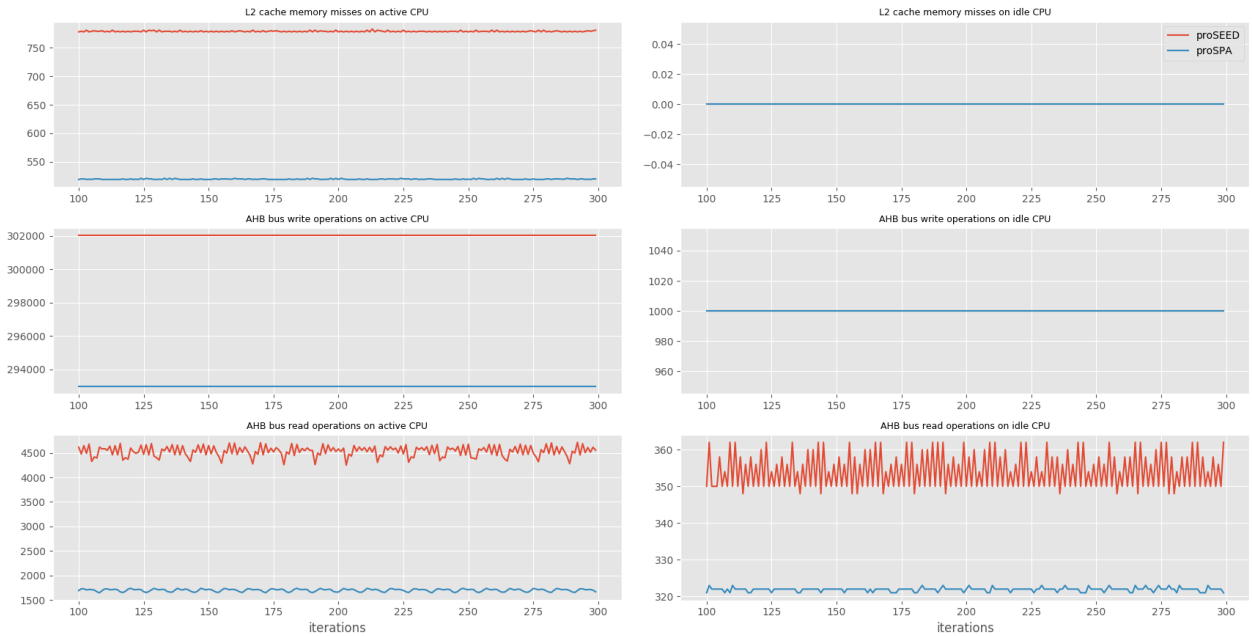

Figure 4.1: Performance metrics for standalone proSEED and proSPA tasks

As it can be spotted, despite not hosting any task on them, the IDLE cores access the AHB processor bus in terms of both read and write operations. This adds a certain amount of interference to the system and needs to be taken into account as a potential interference source. To avoid the interfering activity of IDLE cores, there is a power-down feature available to minimize power consumption during this state. The IDLE thread in the SPARC/LEON BSP typically runs a load operation to make sure it works, which could explain this effect.

4.1.2 Interfering channels

Once the critical tasks have been characterized in isolation, the application has been configured to execute the tasks described in table 4 for the three scheduling scenarios proposed. As it can be seen in figure 4.2, the partitioned scheduling increases the number of L2 cache misses by three orders of magnitude with respect to the isolated case. The impact of the imaging system task can be clearly identified with the four intervals where the number of misses significantly raises over the mean. Concerning the global scheduling configurations, the effect is much softer, as the order of magnitude does not change, but oppositely to the isolated case, the pattern becomes more erratic and unpredictable.

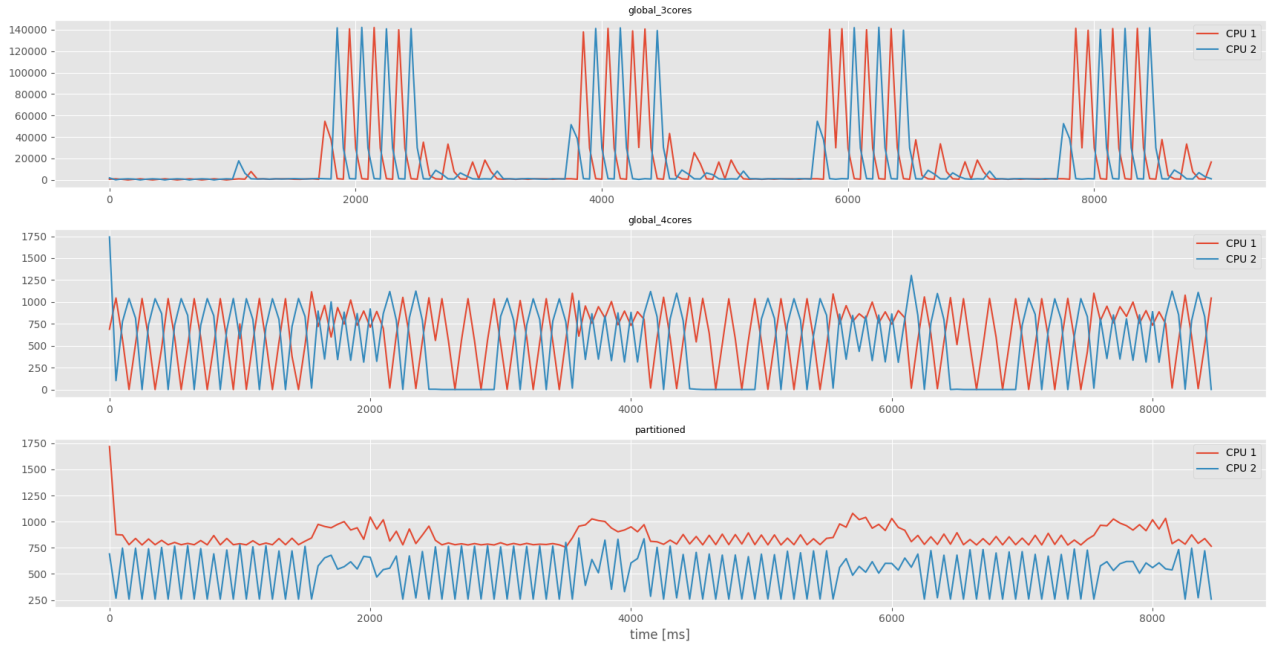

Figure 4.2: L2 cache misses of the application running the full task set

The biggest impact however is suffered by the bus connecting the cores to the L2 cache memory. The number of write operations, which are displayed on figure 4.3, increases by three orders of magnitude for both the global and partitioned schemes with four cores. With three cores, the attained values are higher, given that the image task is continuously preempted by either proSEED or proSPA, which force the cache memory to flush the contents and reload them continuously. With the partitioned scheduler, the variability is slightly reduced with respect to the four-core global scheme.
Concerning the read accesses to the bus, with four-core global and partitioned scheduling approaches, the number increases by two orders of magnitude while with three cores the growth is again by three orders of magnitude, as it can be seen in figure 4.4.

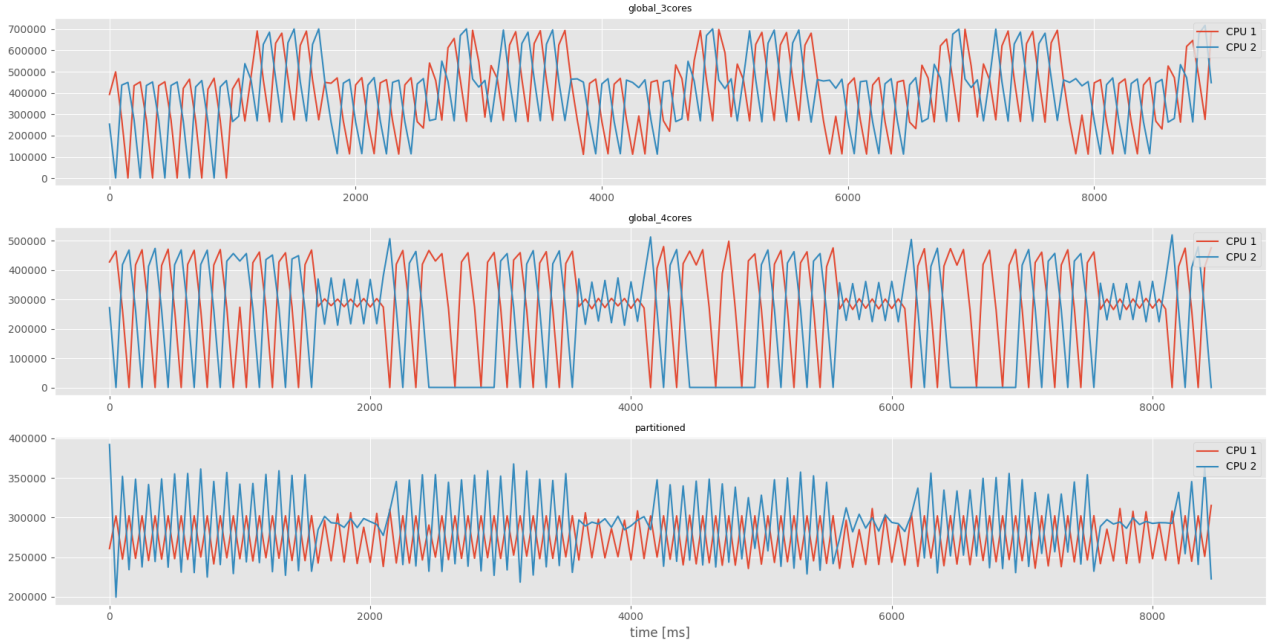

Figure 4.3: AHB bus write operations of the application running the full task set

The structure of each function and in particular the data access frequency is a major contributor to give shape to the patterns. For instance, the core executing the imagSYS task may unable to load all the frame information which implies having to search it on L2. Contrarily, the data set used by the orbPROP task is much smaller and therefore it doesn't need to depend on the L2 cache storing most part of its data. For that reason, this task will add much less interference to the system than imagSYS, and it will definitely be less affected by the other co-running tasks.

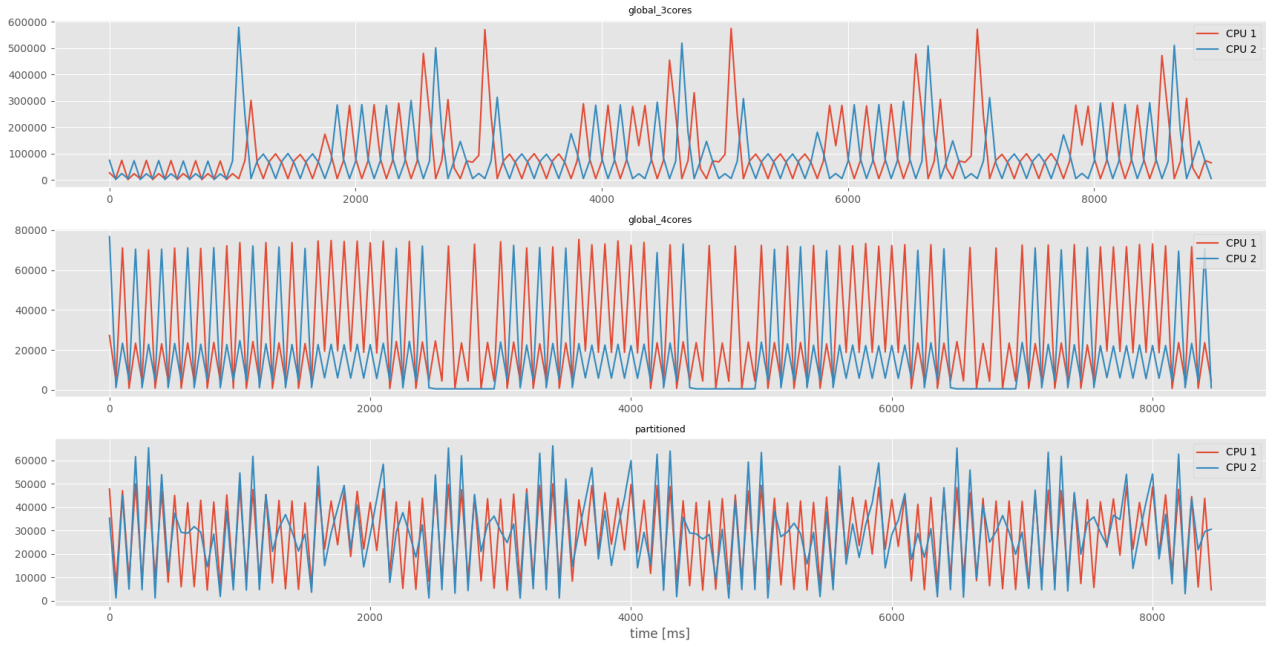

Figure 4.4: AHB bus read operations of the application running the full task set

In views of the obtained results for the application running proSEED, proSPA, imagSYS and orbPROP, it can be claimed that the main interference channels for the designed testbench are the L2 cache memory, the processor bus, the main memory and the memory bus. The maximum execution times observed for the critical tasks are 41 ms and 61 ms, depending on the scheduler choice. This proves that the execution time can be increased in certain occasions by almost a factor of 2, seriously compromising the fulfillment of the timing requirements.

4.1.3 Characterization of the scheduling impact

Another responsible agent for the shape of the execution patterns is the scheduling configuration, as it determines which tasks are executed, in which core they are dispatched and where are they migrated to when a preemption occurs. In figure 4.5, the execution times for both proSEED and proSPA are plotted for each scheduling configuration.

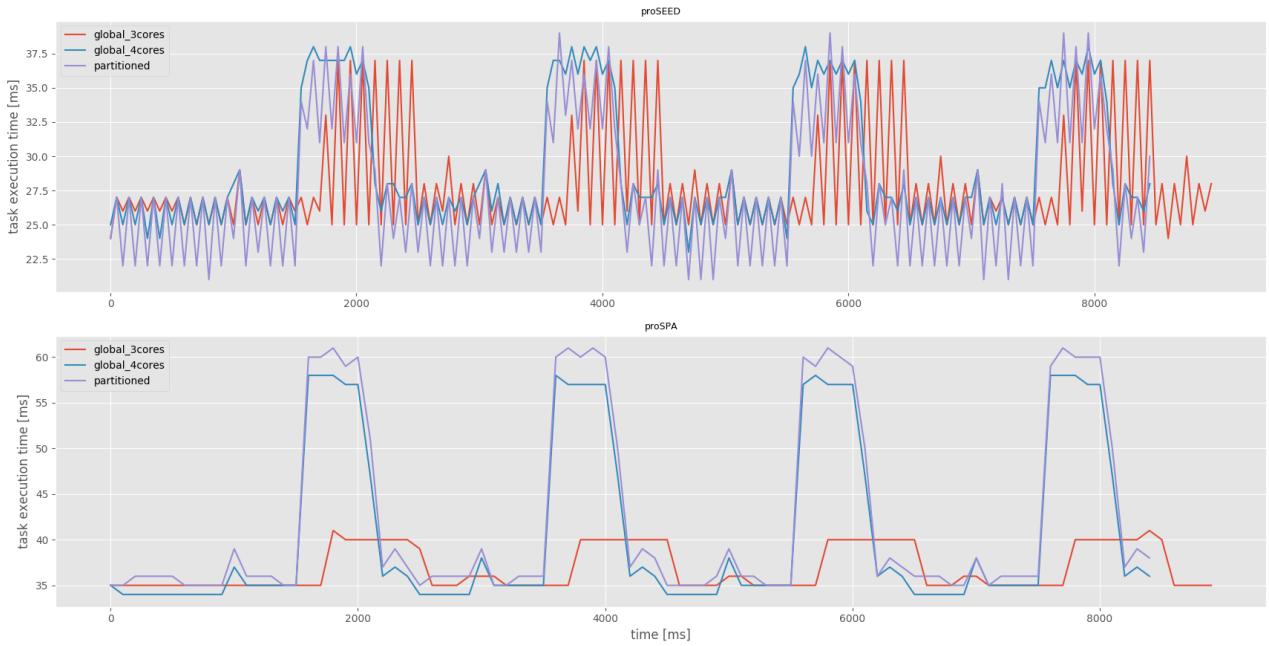

Figure 4.5: Execution time patterns for different scheduling schemes

It can be claimed that when the application uses only three cores, the amount of interference is minimized because a smaller number of tasks is running simultaneously. However, that provokes a delay of 500 ms to process the four frames as the imaging system which has the lower priority is preempted most of the time. Between the global and the partitioned schemes with 4 cores, the response times are exactly the same but it can be seen that the patterns slightly differ among themselves. This effect can be attributed to task migrations caused by the global scheduler, which introduces a certain amount of overhead and additionally, the data stored in the core's L1 cache needs to be flushed and reloaded on the CPU where the task is migrated to.

With the partitioned scheduling configuration, all the tasks are always running which maximizes the amount of interference in the system. To compare the partitioned and the global configurations with four active cores, figure 4.6 shows the number of bus read and write operations on a specific core, where the imagSYS task is being executed. This results

in a good way to show how the performance varies due to a higher amount of interference (partitioned configuration) or due to task preemption (global configuration) independently. The observed information is basically the data access pattern of the imagSYS task, as it has to continuously load data from L2 or memory to L1 given that it is unable to load the entire frame (4 MBytes).

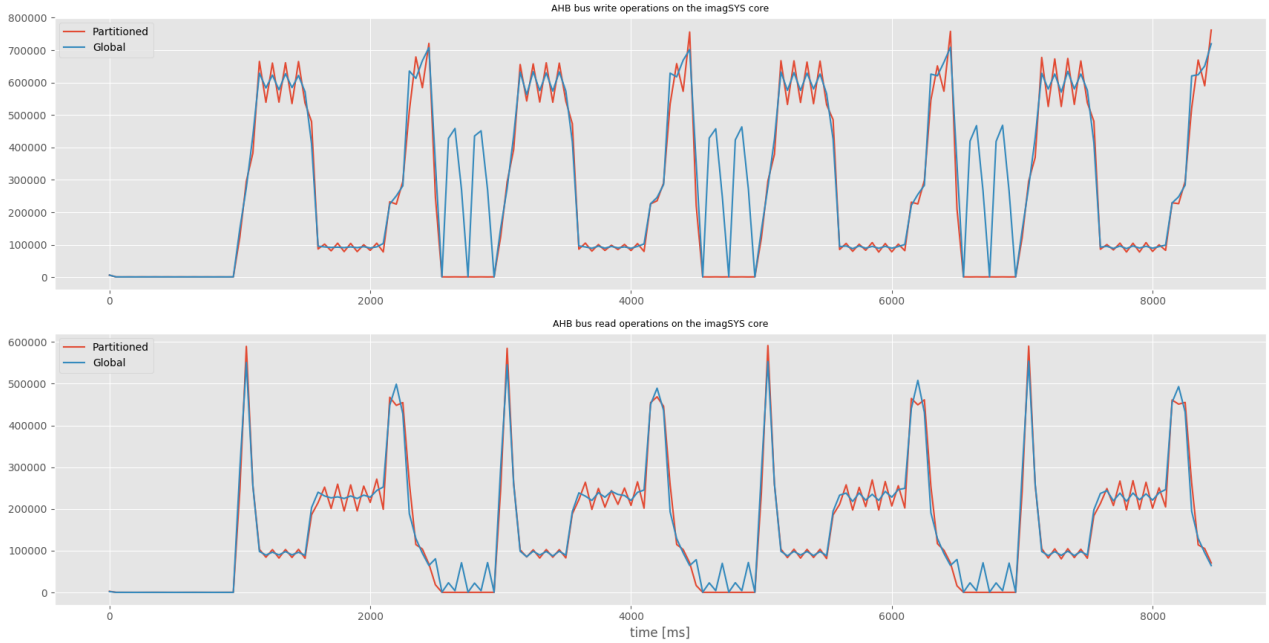

Figure 4.6: AHB bus read and write operations on the core running imagSYS

While the partitioned scheme allows the task to avoid accessing the bus around 2500 ms, 4500 ms and 6500 ms, global scheduler doesn't. Nevertheless, during the intermediate phase of the task, the global scheduler has a smoother pattern than the one offered by the partitioned approach, which will have a direct impact on the variability of the execution times. Thus, it can be concluded that it is significantly complex to predict the effect of the scheduler, as it might be counter-intuitive in some occasions. For that reason, it may be interesting to characterize the effects that it provokes and ultimately understand which approach is better to have the lowest amount of interference.

4.1.4 Non-interfering channels

Due to the structure of the designed application, no IO devices have been used, which reduced the shared resources to the list of channels identified in the previous section. The CAST-32A states that if the applicant identifies interference channels that cannot affect each software component, those channels do not need to be mitigated and no verification is needed.

In case of using IO devices, which would be the case for a real implementation of a space application, for instance having to control the drivers of sensors and actuators through FPGAs, other items should be tested to characterize the interference introduced by them. Specific events can be measured on the IO MMU device of the GR740, which connects the slave and master IO buses (SpW, MIL-STD 1553B, Ethernet...) to the AHB processor and memory buses, making it a potential interference channel. Another shared channel that may generate

interference to the system is the PCI master, linking resources such as GPIO, UART or SPI to the IO MMU bridge.

In the end, all IO devices would be indirectly connected to the same buses that connect the cores to the L2 cache memory and the main memory. With an increasing number of IO devices, it would become more complex to identify the peripherals adding interference. For that reason, this task would not be as straightforward as it has been with the implemented application. Nonetheless, it should be noted that having more cores on the board could promote the reduction of IO peripherals, using a specific CPU if the interference from other cores could be mitigated, leading to a shorter list of potential interfering channels.

4.2 Configuration of mitigation mechanisms

As required by the MCP_Resource_Usage_1 objective of the CAST-32A, the hardware and software configuration that has been employed to reduce the interference on the system needs to be described. The following points cover the different hardware mechanisms that have been used on the GR740 in order to mitigate the timing anomalies and ensure that the deadlines of the critical tasks are always respected.

4.2.1 Cache replacement policy

The first proposed mechanism was based on changing the L2 cache **replacement policy**. By default, a least-recently used (LRU) policy is used. Using a pseudo-random replacement (RR) randomly selects the cache line to replace when data is loaded from memory.

Figure 4.7: Execution time patterns for different scheduling schemes with a cache RR policy

Having changed the replacement policy of the L2 cache memory, the execution times of proSEED and proSPA are shown in figure 4.7 when the pseudo-random replacement policy is used. With this configuration, the pattern described by the number of L2 cache misses is

kept constant throughout the execution, as opposed to the case where the LRU policy is used, which clearly favors the tasks that access the memory more frequently. Even if the number of cache misses increases with the RR policy on the core executing the highest priority task, the others have a bigger chance of finding their data in L2.
This reduces the amount of bus access requests, specially for imagSYS, decreasing the execution time of every task. In general lines, it can be stated that despite adding randomness by changing the replacement policy, this configuration balances the number of L2 cache misses among all the tasks, which allows lower priority tasks to have shorter execution times. For that same reason, with a set of tasks that presents the same characteristics in terms of memory accesses, this feature might have a smaller impact, in comparison to the case where the tasks are more heterogeneous in that aspect.

The following code shows how to configure the REPL field of the L2 cache control register to select the desired replacement policy.

```
/* L2 cache memory replacement policy configuration */
  volatile unsigned int* add_L2_control_reg = (unsigned int*)0xf0000000;
  unsigned int reg_val = *add_L2_control_reg;
  *add_L2_control_reg = reg_val | (unsigned int) 0x10000000; // RR
  *add_L2_control_reg = reg_val & (unsigned int) 0xc0000000; // LRU
```

4.2.2 AHB bus with split requests

The second experiment consisted in using **split transactions** on the AHB buses. When an application running on a specific core requests data to L2, the cache memory inserts wait-states until it is determined whether the read access is a hit or a miss. If the outcome is positive, the data is delivered accordingly. However, if it's a miss, the cache can either insert wait-states during the access to memory or issue an AHB SPLIT operation.

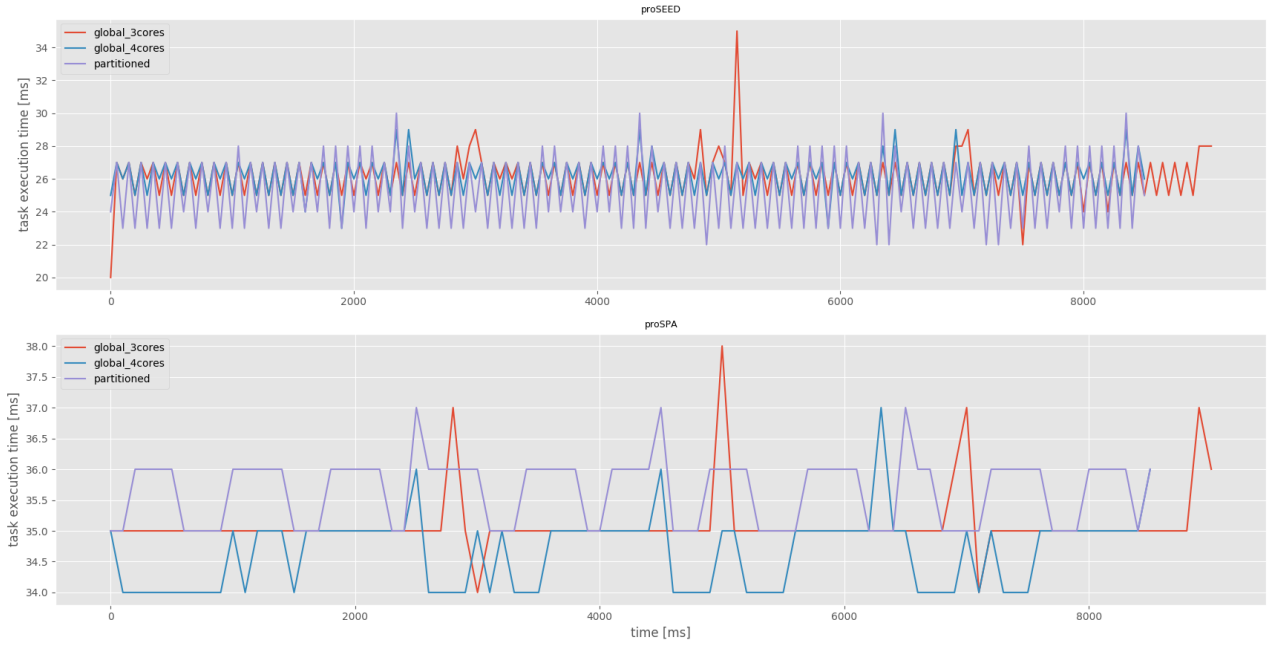

Figure 4.8: Execution time patterns for different scheduling schemes with AHB SPLIT

With this feature, concurrent read requests coming from different cores can be served simultaneously, which reduces the data access latency. According to the measurements obtained, the number of L2 cache misses and bus accesses when split transactions are enabled remains almost the same as with the default configuration. In figure 4.8 the execution time patterns of both proSEED and proSPA are displayed, which can be compared with figure 4.5 to the case when this setting is not activated.

Despite having similar performance metrics, the execution time patterns with and without split transactions are very different. For both critical tasks, the maximum and minimum values lay now within a much smaller range, even though some peaks can still be spotted when using three cores. While the maximum execution times observed with default configuration are 41 ms and 61 ms for proSEED and proSPA respectively, with split transactions these can be reduced to 35 ms and 38 ms. The reason of this improvement is given by the different approach to access the bus, as now tasks accessing data with a lower frequency (proSPA and orbPROP) would still be able to request the data and will avoid getting blocked by other tasks having a much higher access frequency to the cache (proSEED and imagSYS).

The following code shows how to enable split transactions on the L2 access control register.

```
/* Enable/disable AMBA SPLIT responses */
volatile unsigned int* add_L2_access_reg = (unsigned int*)0xf000003c;
unsigned int reg_val_2 = *add_L2_access_reg;
*add_L2_access_reg = reg_val_2 | (unsigned int) 0x2; // enable
*add_L2_access_reg = reg_val_2 & (unsigned int) 0xd; // disable
```

4.2.3 Additional methods

Other mechanisms to remove interference from the system have been considered but not implemented on the final configuration. For instance, cache way locking allows to block a set of lines of the L2 cache memory to prevent a specific task of having cache misses. This could be particularly useful in order to keep frame data in L2, so that the images do not need to be loaded from the main memory, every time the task is preempted. Therefore, it may be a feature that could be employed in order to optimize the application, taking into account the memory access patterns of each task. In case of using IO devices, one interesting feature would be to work with the settings of Direct Memory Access (DMA), which allows peripherals to access the memory without going through the CPUs. It would be a good practice to study the effect of having DMA directly into the main memory, instead of passing through L2 to avoid overloading the resource. Other changes that could be done on the L2 could be switching the write policy of the cache or even disabling it in specific scenarios of the application.

4.3 Verification of interference reduction methods

This section covers part of the MCP_Resource_Usage_3 objective of the CAST-32A, in order to verify that the proposed mitigation mechanisms are able to sufficiently reduce the presence of inter-core interference in the system. Additionally, in compliance with the MCP_Software_1 point, proof of evidence is provided to show that each hard real-time software component has enough time to execute before compromising the integrity of the application.

A measurement-based approach has been used to obtain the WCET for both proSEED and proSPA, based on the **Chronbach's alpha** index. This coefficient ranges from 0 to 1 and provides a reliability measurement of the values present in the data set. It is computed as a function of the number of tests and the average inter-correlation among the items. A coefficient value above 0.9 indicates a high consistency on the obtained results, while a low value might indicate that more tests are required to expand the observed range.

For each hardware configuration and scheduling scheme, a specific number of iterations has been launched to attain a 0.9 alpha coefficient. Even though this indicator is usually used to characterize the consistency of a data set, here it has been employed to quantify the randomness of a specific configuration and to achieve the necessary amount of confidence on the results. To determine how much predictable the application becomes with a particular configuration, the probability density function of the execution times has been calculated and it has been plotted on figure 4.9.

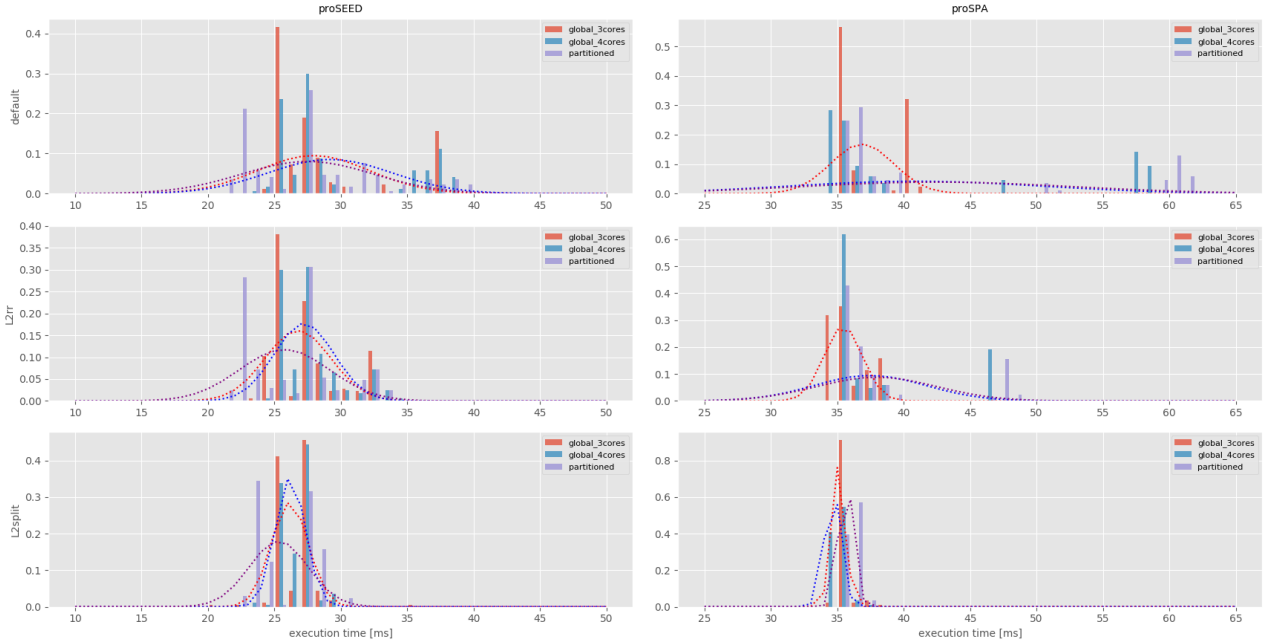

Figure 4.9: Probability density functions of the execution times in every implemented scenario

As it can be observed, the biggest impact is sensed by the proSPA which has a lower priority than proSEED and executes at twice its period. For this task, the variance is reduced by an 80.01% with the RR policy and 99.60% with SPLIT transactions, when using a four-core global scheduler. When the partitioned scheduler is employed, the reduction becomes 80.57% and 99.71%. Otherwise, if three cores are used the differences are less significant (63.34% and 95.37%), since the variance is already small due to the fact that only three tasks execute in parallel instead of four.
In the case of proSEED, the reduction in execution time variance with respect to the default configuration is of 65.27% and 88.99% with three-core global scheduling, 76.72% and 94.28% with four-core global scheduling and finally 53.22% and 80.03% with four-core partitioned scheduling, each pair representing the reduction achieved by RR and SPLIT respectively.

Considering the execution times as samples of a Gaussian random variable, the probability to have each possible execution time between the minimum observed and the deadline of the task has been computed. To do that, the Q distribution (2) has been employed, which depends on the error function (3) and represents the tail distribution function of a standard normal random variable. The x variable is expressed as the difference between the studied execution time and the average of the distribution, divided by the standard deviation.

$$P(Y > y) = P(X > x) = Q(x) = \frac{1}{2} - \frac{1}{2} erf(\frac{x}{\sqrt{2}}) \tag{2}$$

$$erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt \tag{3}$$

The obtained probabilities have been displayed on figure 4.10, for proSEED and proSPA using different combinations of hardware mitigation mechanisms and scheduling settings. In view of the results, it can be claimed that despite minimising the probability of missing the deadline of proSEED (< 0.01), using the L2 cache RR policy does not sufficiently reduce it for proSPA, when all the cores are used. More specifically the probability of having an execution time above 50 ms is 0.24 and 0.28, using the global and partitioned schedulers, which is too high to be considered a sufficiently valid mechanism to mitigate interference.

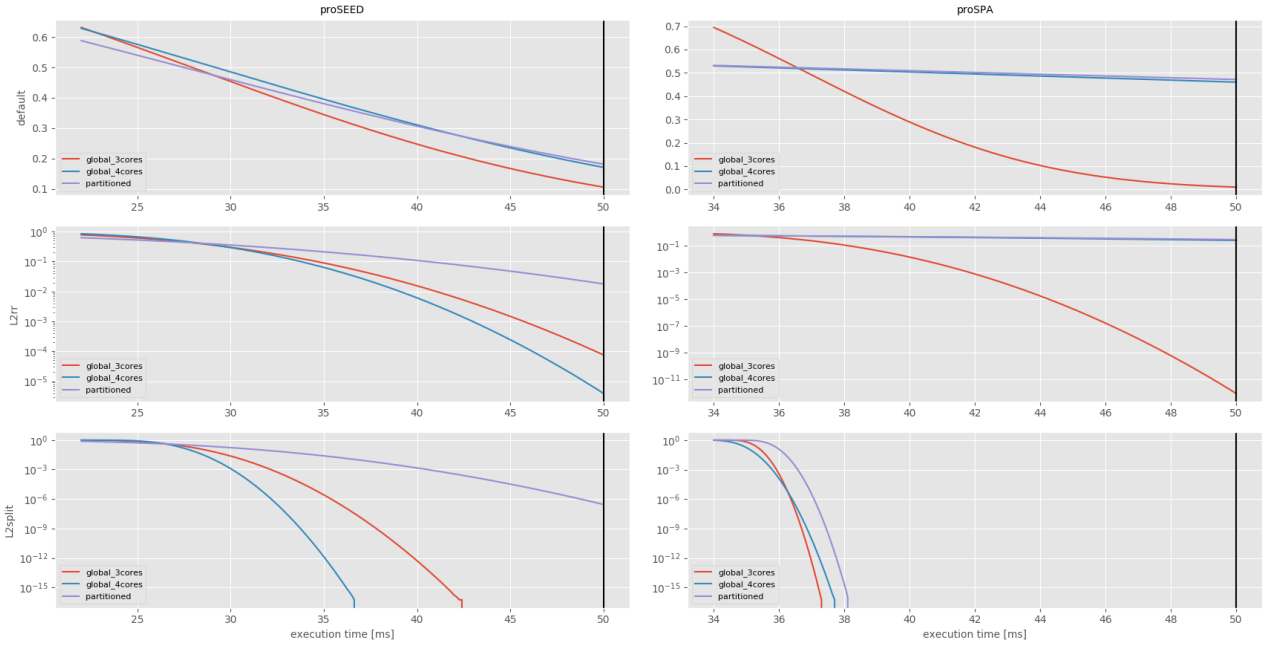

Figure 4.10: Execution time occurrence probabilities for all the available configurations

When split transactions are enabled, the results are much positive, as the probability of surpassing the deadlines is reduced below the the order of 10^{-9}. In addition, it is the global scheduler with all active cores that works better in this configuration, as the probability curves decay faster and earlier than with the other schedulers. The obtained probabilities are sufficient to prove that using the described configuration of the application, the inter-core interference is mitigated, which is directly reflected on the fact that executing each critical takes almost the same time as in the standalone case.

In addition to the probabilistic study of the application's worst-case execution time, in table 5 are provided the performance results using all the possible configurations that have been implemented.

	Global scheduling with 3 CPUs					
Task	**proSEED**			**proSPA**		
Configuration	Default	L2 RR	SPLIT	Default	L2 RR	SPLIT
Average execution time	27.94 ms	26.77 ms	26.15 ms	36.86 ms	35.44 ms	35.09 ms
Variance execution time	17.65 ms	6.13 ms	1.94 ms	5.62 ms	2.06 ms	0.26 ms
WCET	41 ms	32 ms	35 ms	58 ms	38 ms	38 ms
	Global scheduling with 4 CPUs					
Task	**proSEED**			**proSPA**		
Configuration	Default	L2 RR	SPLIT	Default	L2 RR	SPLIT
Average execution time	29.19 ms	27.24 ms	26.19 ms	40.87 ms	37.45 ms	34.65 ms
Variance execution time	21.87 ms	5.09 ms	1.25 ms	91.35 ms	17.84 ms	0.36 ms
WCET	38 ms	34 ms	29 ms	58 ms	46 ms	37 ms
	Partitioned scheduling					
Task	**proSEED**			**proSPA**		
Configuration	Default	L2 RR	SPLIT	Default	L2 RR	SPLIT
Average execution time	27.50 ms	25.75 ms	25.32 ms	42.44 ms	37.96 ms	35.64 ms
Variance execution time	24.69 ms	11.55 ms	4.93 ms	105.3 ms	20.46 ms	0.30 ms
WCET	39 ms	33 ms	30 ms	61 ms	48 ms	37 ms

Table 5: Timing statistics for proSEED and proSPA for each implemented configuration

Apart from the information that is given below, the response time of the application must be as well taken into account. As mentioned in the previous section, it takes the same amount of time to process four frames when all the cores are used for both global and partitioned schemes. Oppositely, when only three cores are employed, the application needs 500 extra milliseconds to accomplish the same objective.

5 Conclusions

Throughout the course of this project, the challenge of hosting space applications on a multicore system has been approached by following the guidelines proposed in the CAST-32A paper. To analyse the performance of a real space application, a simplified version of the Prospect's software architecture has been implemented.
In attempt to extend the same functionalities to a multicore system, a quad-core Leon 4 GR740 has been used to run a SMP application deployed on top of RTEMS 6, reducing weight and power consumption and increasing the efficiency with respect to the original design of the studied instrument.

The steps that have been followed to achieve the objectives set during the initial phase of the project are summarized in the following points:

1. A use case based on a realistic application has been designed and implemented to take on the validation of space flight software running on a multicore processor:

   (a) An analysis of the most significant benchmarks for embedded systems in the areas of aerospace and automotive has been performed and discussed. Specific algorithms have been selected from AutoBench and OBPMark, in order to emulate the functionalities that are implemented on the Prospect instrument module (3.1).

   (b) A tunable application has been implemented in order to evaluate the impact of different system's parameters on the overall performance, including the type of scheduling scheme, the number of active cores and specific hardware settings related to the cache memory system of the board (3.2).

2. The guidelines provided in the CAST-32A paper have been followed to perform a partial validation of the implemented application on a Leon 4 GR740 board:

   (a) The shared channels generating timing interference on the timing behavior of the software application have been identified on the GR740 multicore processor. Subsequently, mitigation mechanisms have been proposed and later verified (4.1).

   (b) The correct behavior of each software component hosted by the multicore processor has been validated following a measurement-based approach, showing that the probability of not respecting real-time constraints is negligible (4.2 and 4.3).

The interfering channels that have been identified on the GR740 for the designed application included the L2 cache memory, the processor AHB bus, the SDRAM memory and the memory AHB bus. Nonetheless, it has also been noted that the scheduling configuration has a direct impact on the amount of interference present in the system. Additionally it has been concluded that disabling cores instead of leaving them in IDLE state when nothing is being executed can increase the level of predictability of the execution times.

More advanced applications may use additional resources such as SpaceWire, MIL-STD-1533 or Ethernet buses. Due to the limited number of counters in the board's statistics unit, gathering information from all the interfering channels at all the cores may become more complex and expensive when the number CPUs increases. This might be a limiting factor for

the next generation of SPARC processors, such as the octa-core GR765 Leon 5, in order to study the interference generated on the board. Furthermore, it may be difficult to identify which specific channel is interfering the system when the number of used resources increases, as they are connected either directly or indirectly to the same communication network.

In view of the obtained results, it can be claimed that a space application can be partially validated by identifying the interference generators and mitigating the associated timing anomalies. For an application that stresses the memory system, it has been proven that the way the bus is accessed becomes key to mitigate inter-core interference.
The probability of the critical tasks not respecting their deadlines below a soil of the order of 10^{-9}, using all the available cores and allowing tasks to migrate. Consequently, the execution times can be well bounded around a specific value with a maximum error of one millisecond. Following the outcome of the experiments, the safety bounds that are pessimistically set to account for possible pathological cases can be made much tighter, substantially improving the accuracy of the scheduling analysis and concluding the activities for timing analysis.

Despite the positive results achieved of this project, some further work considerations should be taken into account to approach full validation on multicore platforms:

- Migrating multiple single core boards to a unique multicore processor implies having inter-core communication instead of external buses to route the information. That would require additional testing to ensure that the mechanisms to lock shared variables work properly, so that data and control coupling are not compromised.

- Automatizing the process to instrument the code by tracing the routines provided by RTEMS's classic API. In this project, the code has been manually instrumented and the traces have been generated and processed through a combination of GRMON, GDB and Python scripts. Applying this procedure to more advanced applications would become infeasible due to the increase in time and complexity.

- It can occur that no hardware mechanisms are available to mitigate timing anomalies. Software techniques may be employed in this case to provide robust partitioning, such as a lightweight hypervisor. Additionally, message passing with proper locking mechanisms can be implemented among tasks that need to access the same resource, so that just a single task can be dedicated to acquire the data and distribute it to the others.

References

[1] AutoBench Performance Benchmark Suite. EEMBC.

[2] OBPMark – Open Source Computational Performance Benchmarks for Space Applications. Zenodo, June 2021.

[3] European Space Agency. Space avionics open interface architecture, 06 2019.

[4] F. Barbier. Composability for software components: an approach based on the whole-part theory. In *Eighth IEEE International Conference on Engineering of Complex Computer Systems, 2002. Proceedings.*, pages 101–106, 2002.

[5] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Nicole Megow, and Leen Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 61(8):1140–1152, 2012.

[6] Adam Betts. Hybrid measurement-based wcet analysis using instrumentation point graphs. 01 2010.

[7] Francisco J. Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu-Grosjean, Benoît Triquet, Guillem Bernat, E. Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Code Lo, and Dorin Maxim. Proartis: Probabilistically analysable real-time systems. 2012.

[8] Martin Croxford. Correctness by construction : A manifesto for high-integrity software. 2006.

[9] ESA Board for Software Standardisation and Control (BSSC). *Software Verification and Validation.* ESA PSS-05-10 Issue 1 Revision 1, 1995.

[10] Joël Goossens, Raymond Devillers, and Shelby Funk. Tie-breaking for edf on multiprocessor platforms. 01 2003.

[11] Carles Hernández, Jaume Abella, Francisco J. Cazorla, Jan Andersson, and Andrea Gianarro. Towards making a leon3 multicore compatible with probabilistic timing analysis. 2015.

[12] Thomas Huybrechts, Siegfried Mercelis, and Peter Hellinckx. A new hybrid approach on wcet analysis for real-time systems using machine learning. 09 2018.

[13] David A. Patterson John L. Hennessy. *Computer Architecture: A quantitative approach (5th edition).* 2011.

[14] Larry Kinnan. Use of multicore processors in avionics systems and its potential impact on implementation and certification. pages 1.E.4–1, 11 2009.

[15] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, jan 1973.

[16] John McDermid and Tim Kelly. Software in safety critical systems: Achievement and prediction. *Nuclear Energy-journal of The British Nuclear Energy Society - NUCL ENERG-J BRIT NUCL ENERG*, 2:140–146, 05 2006.

[17] John A McDermid and David J Pumfrey. Software safety: Why is there no consensus?

[18] David Radack, Harold Jr, and Paul Parkinson. Civil certification of multi-core processing systems in commercial avionics. 06 2018.

[19] FAA Software Team CAST-32A. "multi-core processors", position paper, certification authorities. 2016.

[20] CAES (Cobham Advanced Electronic Solutions). *GR740 Data Sheet*. 2021.

[21] Wind River Systems. *Certification of Avionics Applications on Multi-core Processors: Opportunities and Challenges*. Wind River Systems, Inc, 2018.

[22] Alex Wilson and Thierry Preyssler. Incremental certification and integrated modular avionics. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pages 1.E.3–1–1.E.3–8, 2008.