Applying Hypervisor-Based Fault Tolerance Techniques to Safety-Critical Embedded Systems

by

Santiago Lozano Terol

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in

Computer Science and Technology

Universidad Carlos III de Madrid

Advisor(s):

Jesús Carretero Pérez

Tutor:

Jesús Carretero Pérez

March 2023

ACKNOWLEDGEMENTS

El éxito de esta tesis es el resultado de la aportación, de una forma u otra, de muchas personas diferentes. Por tanto, y aunque intentaré ser breve, es de justicia escribir unas líneas de agradecimiento hacia ellas:

A mi tutor en la universidad, Jesús. He comprobado que un doctorado con mención industrial es muy difícil de dirigir: las necesidades de la empresa son cambiantes y siempre urgentes y no es fácil mantener el foco en la investigación. En los momentos críticos, tú me ayudaste a encauzar la tesis.

A los miembros del tribunal de la prelectura de la tesis: Félix, Javier y Katzalin. Os agradezco la exhaustiva y respetuosa revisión de nuestro trabajo. Vuestros comentarios fueron muy valiosos para terminar de darle forma al libro.

A Cristina Tato, que me ofreció la posibilidad de realizar este doctorado y me animó a ello. En los momentos en los que se puso más difícil compatibilizar mi trabajo en la empresa y en la universidad me repetías que, en el futuro, veríamos toda esta aventura como un acierto. Creo que tenías razón. Gracias por realizar el delicado ejercicio de equilibrio que supone guiarme y, a la vez, darme la autonomía y la responsabilidad suficientes para crecer como ingeniero.

A mi madre. Gracias a ti empecé a estudiar una ingeniería hace ya diez años. Si ahora alcanzo el techo académico es, en parte, porque siempre me he esforzado para que estés orgullosa de mí.

A mi padre. Quizá por esta estúpida vergüenza masculina no te he dicho nunca lo que siento y creo que sabes bien: que estos años he coincidido con grandísimos profesionales de muchos países diferentes y aún no he conocido a ninguno al que admire tanto como te admiro a ti. Ahora que he finalizado mis estudios, mi próximo objetivo es convertirme en un profesional tan bueno y distinguido como tú.

A mis hermanos, a mis tíos y primos, a mis abuelos (y a mis abuelas). A mis amigos, a mis compañeros de trabajo. Tengo la suerte de que sois tantos que empezar a escribir nombres no tiene sentido, pero vosotros sabéis quiénes sois. Si en estos últimos tres años hemos reído, hemos comido, cenado, hemos salido de fiesta, hemos viajado, hemos hecho deporte o, en definitiva, hemos pasado un buen rato juntos, estás incluido en este párrafo.

Sobre todo a ti, Tere. Gracias por el cariño, por la paciencia, por asumir en muchas ocasiones tus responsabilidades y las mías para regalarme un poco de tiempo. Por hacerlo todo con una sonrisa. Sin ti no hubiera sido posible y sin ti no tendría sentido.

Por último, a la personita que me ha obligado a darle el empujón definitivo a la tesis. Pequeña Lola: todas las horas diarias que se ha llevado esta investigación las pienso dedicar, de ahora en adelante, a pasar tiempo con tu mamá y contigo.

PUBLISHED AND SUBMITTED CONTENT

Publications and contributions have been made and are included as part of this thesis:

- Santiago Lozano, Tamara Lugo and Jesús Carretero. A Comprehensive Survey on the Use of Hypervisors in Safety-Critical Systems. *IEEE Access*, 2023.
  - This article is currently under review. The material contained in this article has been fully included in Chapter 2 of this thesis. The content from this item has not been singled out with typographical means and references throughout this thesis.
- Santiago Lozano, Juan Fombellida, Carlos Rodríguez, Cristina Tato, Jesús Carretero. MFOC Project: MPSoC-Based Multi-Purpose Execution Platform. In *III Congreso de Ingeniería Espacial : El espacio, la última frontera*, 27-29 Octubre 2020, Madrid, España. ISBN: 978-84-09-31948-0. pp. 120-122. Extended abstract and oral presentation.
  - The material contained in this publication has been partially included in chapters 1 and 2 of this thesis. The content from this item has not been singled out with typographical means and references throughout this thesis.

OTHER RESEARCH MERITS

Other articles, publications or lectures conducted during the doctoral thesis:

- Tamara Lugo, Santiago Lozano, Javier Fernández, and Jesús Carretero. A Survey of Techniques for Reducing Interference in Real-Time Applications on Multicore Platforms. *IEEE Access*, 10:21853–21882, 2022. doi: 10.1109/ACCESS.2022.3151891.
- Santiago Lozano. MIA: Multi-Purpose Space Platform using cFS and TSP. Flight Software Workshop 2022. Oral presentation.
- Santiago Lozano. Plataforma Multipropósito para los Sistemas Espaciales 2023. I Semana Interdisciplinar del Espacio y IV Congreso de Ingeniería Espacial, 2021. Oral presentation.
- Santiago Lozano. Revisiting the design of small satellites for speeding up the products and improving flexibility and performance. ESA Joint Board on Communication Satellite Programmes 5G Advisory Committee, 2022. Oral presentation.
- Santiago Lozano. Design and deployment of space systems using MPSoC technology. Go2Space Hackaton, 2021. Oral presentation.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

This chapter is an introduction to the work done in this PhD thesis. The chapter is divided into five main sections:

- Section 1.1 contains the technical motivation of the thesis.
- Section 1.2 explains the particular conditions under which this doctoral thesis originated as a result of the collaboration between academia and industry.
- Section 1.3 sets out the main objective of the research and the various specific objectives to achieve it.
- Section 1.4 lists the main contributions that have been obtained as research outputs, which can be disseminated in journals and conferences or exploited for projects in the aviation and space industry.
- Section 1.5 sets out the structure of the rest of the document and briefly describes its contents.

1.1. Motivation

The success of many safety-critical systems, such as flight systems or some automotive subsystems, depends to a large extent on their ability to maintain deterministic behavior over long periods of time and in an autonomous or semi-autonomous manner (with very limited external real-time support). This task is hindered by the hostile conditions in which these systems have to operate in many cases: vacuum, radiation, extreme temperatures, impacts.... All these constraints make the processes and products related to safety-critical industries complex and costly, which is a major barrier to entry for small and medium-sized companies. For example, the most common fault tolerance mechanisms to protect flight systems against radiation-induced soft errors are usually the use of electronic components that are specially hardened against ionizing particles or hardware redundancy. In both cases, the fault tolerance mechanism entails a cost overrun and a considerable increase in the weight and power consumption of the final product.

In recent years, both in flight systems and automotive systems, there has been a trend towards the use of COTS components with high processing capacity to meet increasingly demanding system requirements. In the space sector, the term "new space" has emerged, which is often used to refer to a new non-institutional market based on small platforms with associated lighter development and manufacturing processes and less stringent reliability requirements. Obviously, in this type of platform, fault tolerance mechanisms must evolve accordingly so that they do not entail such a high cost overrun. This thesis

addresses this problem, using virtualization to propose an alternative to hardware redundancy that maintains high levels of reliability while drastically reducing the redundancy footprint.

1.2. Context

This industrial PhD was born from the collaboration between SENER Aerospacial and the Universidad Carlos III de Madrid. After a long and fruitful collaboration in research and teaching tasks between both institutions, in 2020 a consortium was formed, coordinated by SENER Aerospacial, to carry out a project called Madrid Flight on Chip (MFOC). In addition to SENER Aerospacial, this consortium is made up of the Universidad Carlos III de Madrid, IMDEA Software and a number of small and medium-sized companies from the Community of Madrid: GENERA, CENTUM, REUSE and MARM.

The objective of the MFOC project is to develop and mature new techniques for the development of small satellites and next generation space systems. This includes exploring software and hardware techniques that can revolutionize the space design and development process to obtain more agile and cost-efficient missions than traditional space missions. The project uses commercial electronic components such as embedded multicore processors and FPGAs, which implies the challenge of making these components robust to cosmic radiation, since they have not been specially designed for operation in such a hostile environment. Studying the existing scientific literature on hardware redundancy, it became apparent that there was an option not widely explored that potentially involves much less system footprint: using a hypervisor to deploy virtual machines on top of the embedded system, so that redundant applications run on these virtual machines in isolation from each other.

Given this research opportunity and given that there were very good conditions within the framework of the MFOC project, since SENER Aeroespacial and the Universidad Carlos III de Madrid had to carry out several activities of the project in close collaboration, the possibility of carrying out an industrial doctoral thesis was raised. In addition, in 2019 the Community of Madrid launched a call offering funding for the realization of industrial PhDs (order 1724/2019 of May 30, 2019). In November (order 52/2019) the aid was granted to SENER Aeroespacial and the Universidad Carlos III de Madrid, so that in March 2020 the doctoral thesis officially began.

1.3. Objectives

The initial objective of the thesis is **to explore novel techniques to improve the reliability and efficiency of software on MPSoC platforms**. During the research, looking at the different possibilities, we have decided to implement a fault tolerance mechanism based on the use of a hypervisor to redundant software applications in independent virtual

machines, isolated from each other, and examine how this affects the reliability and safety of the system. This general objective can be subdivided into different specific objectives:

O1 Study and selection of the appropriate tools and environments for the research:

(a) Study and selection of the hardware environment, which should include at least a multicore processor and an embedded FPGA.

(b) Study and selection of the hypervisor (and the operating system, if applicable) that will constitute the software base of the mechanism.

O2 Co-design and implementation of a hypervisor-based space system that will enable the deployment of mixed-criticality applications.

O3 Definition and implementation of a space use case for applying the Hypervisor-Based Fault Tolerance (HBFT) mechanism.

O4 Investigation of the effect of the HBFT mechanism on system safety and reliability.

In addition, given the mixed nature of this PhD (industry-university collaboration), the following non-technical objectives are proposed:

O5 Dissemination of results in specialized congresses related to the space industry and in research journals.

O6 Strengthening of the links between the university (Universidad Carlos III de Madrid) and the company (SENER Aeroespacial) to allow future collaborations in space projects.

1.4. Contributions

The main contributions of this thesis are:

C1 A methodology that allows implementing a Hypervisor-Based Fault Tolerance (HBFT) mechanism. This methodology involves the use of a hypervisor and the replication of the functionality to be redundant in different virtual machines, following (at least) a Triple Modular Redundancy scheme. In addition, an additional virtual machine called Health Monitoring is used, which monitors the correct operation of the virtual machines and takes the necessary measures to restore them in case of failure.

C2 A prototype implementing the HBFT mechanism exposed, as well as an evaluation of the prototype by simulating Single Event Upsets (SEUs) that cause failures in the computations of the virtual machines following different statistical distributions. For the functionality to be redundant, a space benchmark currently in use in several real missions is used.

C3 A comprehensive state-of-the-art review of the use of hypervisors in safety-critical embedded systems, especially focused on the space, aviation, and automotive industries. In addition to the review of related works, a comparison is made between all hypervisors according to different parameters. Beyond helping to choose the most suitable hypervisor for this study, the review will allow equivalent decisions to be made for other future studies.

C4 A trade-off that evaluates the advantages of moving the voter used to implement the HBFT mechanism to the FPGA, rather than running it as just another virtual machine. This approach of bringing all the voting logic to digital hardware has not been raised or executed in other studies dealing with safety-critical systems, to the best of our knowledge.

1.5. Structure and Contents

This document details the work conducted through the development of this thesis, and it is structured as follows:

- Chapter 1, *Introduction*, has briefly presented the motivation, objectives, and contributions of this thesis.
- Chapter 2, *Fundamentals*, exposes a series of concepts that are necessary to correctly understand the information presented in the rest of the thesis, such as the concepts of virtualization, hypervisors, or software-based fault tolerance. In addition, this chapter includes an exhaustive review and comparison between the different hypervisors used in scientific studies dealing with safety-critical systems, and a brief review of some works that try to improve fault tolerance in the hypervisor itself, an area of research that is outside the scope of this work, but that complements the mechanism presented and could be established as a line of future work.
- Chapter 3, *Problem Statement and Related Work*, explains the main reasons why the concept of Hypervisor-Based Fault Tolerance was born and reviews the main articles and research papers on the subject. This review includes both papers related to safety-critical embedded systems (such as the research carried out in this thesis) and papers related to cloud servers and cluster computing that, although not directly applicable to embedded systems, may raise useful concepts that make our solution more complete or allow us to establish future lines of work.
- Chapter 4, *Proposed Solution*, begins with a brief comparison of the work presented in Chapter 3 to establish the requirements that our solution must meet in order to be as complete and innovative as possible. It then sets out the architecture of the proposed solution and explains in detail the two main elements of the solution: the Voter and the Health Monitoring partition.

- Chapter 5, *Prototype*, explains in detail the prototyping of the proposed solution, including the choice of the hypervisor, the processing board, and the critical functionality to be redundant. With respect to the voter, it includes prototypes for both the software version (the voter is implemented in a virtual machine) and the hardware version (the voter is implemented as IP cores on the FPGA).

- Chapter 6, *Evaluation*, includes the evaluation of the prototype developed in Chapter 5. As a preliminary step and given that there is no evidence in this regard, an exercise is carried out to measure the overhead involved in using the XtratuM hypervisor versus not using it. Subsequently, qualitative tests are carried out to check that Health Monitoring is working as expected and a fault injection campaign is carried out to check the error detection and correction rate of our solution. Finally, a comparison is made between the performance of the hardware and software versions of Voter.

- Chapter 7, *Conclusions and Future Work*, is dedicated to collect the conclusions obtained and the contributions made during the research (in the form of articles in journals, conferences and contributions to projects and proposals in the industry). In addition, it establishes some lines of future work that could complete and extend the research carried out during this doctoral thesis.

2. FUNDAMENTALS

The central idea around which this thesis revolves is the application of virtualization techniques to implement fault tolerance in safety-critical embedded systems. There are several works related to this topic, especially in the last decade, which will be reviewed in chapter 3. Before that, however, it is important to expose certain concepts that will help to understand the research interest of this thesis and to understand some of the decisions that have been taken to design and develop the prototype, such as the choice of the hypervisor. Likewise, this chapter (especially the section related to embedded safety-critical hypervisors) may help other researchers to replicate or extend the results obtained in this work.

The chapter is divided into six main sections:

- Section 2.1 is devoted to virtualization. It briefly reviews the history of virtualization and explains key concepts such as hypervisors and how they are usually classified. It also discusses how virtualization is applied to industries where embedded processors and electronics are used and safety is a critical requirement. In particular, the space, aviation and automotive sectors are analysed. Finally, the scientific literature on hypervisors applied in these industries is reviewed in detail, and a comparison is made between them. The aim of this is twofold: on the one hand, to justify the choice of the hypervisor chosen to develop the prototype presented in chapter 4; on the other hand, to summarise in a few pages the main characteristics of the hypervisors used in different research carried out during the last two decades, so that other researchers have easier access to this information and can choose the hypervisor that best suits their needs, in case they wish to replicate or extend the work done in this thesis.

- Section 2.2 is devoted to briefly explain the main standards by which certification authorities measure and approve software safety in the space, aviation and automotive industries. This thesis has a dual nature, since it is carried out through a university-industry collaboration, so it is interesting to explain, even if only briefly, the standards that the hypervisor and the rest of the software developed in this or other research works would have to comply with, if it were to be put into practice in a real commercial system.

- Section 2.3 is dedicated to briefly explaining what redundancy in safety-critical systems consists of, why it is necessary and what types of redundancy exist. In particular, some software-based redundancy techniques will be analysed, which will help to understand why it is interesting to apply virtualization to implement this type of redundancy.

- Section 2.4 is devoted to analysing some of the main works that focus on protecting hypervisors against potential failures. Although this is not the subject of this thesis, since our solution proposes to **use the hypervisor** to **protect the rest** of the system, it is a complementary area of research that will need to be explored by anyone who wants to put into practice what is going to be presented in this thesis.
- Section 2.5 briefly exposes the concept of Single Event Upset (SEU), which are the main errors that are intended to be detected and corrected by the mechanism presented in this thesis.
- Section 2.6 briefly summarises the chapter and introduces the contents of the following chapter.

2.1. Virtualization

2.1.1. Brief History of Virtualization

Despite being a topic that has been booming in recent years, the roots of virtualization date back to the 1960s. Christopher Strachey is considered one of the pioneers on the subject, being the first person [1] to publish an article dealing with the concept of time-sharing at the International Conference on Information Processing at UNESCO, Paris, in June, 1959 [2]. The time-sharing technique is based on the fact that a computer can be used by several users at the same time. This technique would imply over time revolutionary changes in the industry, including the appearance of the concept of virtualization.

The first experimental time-shared system was accomplished in 1961 by a MIT group, led by Professor Fernando J. Corbató [3]. In 1963, the same group developed an operational version for the IBM 7094 mainframe, called CTSS (Compatible Time Sharing System). This system would be the basis of the famous MIT's Project MAC (*Mathematics and Computation*, later renamed to *Multiple-Access Computer*) [4]. One of the main objectives of the project was the creation of a large, multiple-access computer system, available to meet the needs of a large number of users individually. In this context, MIT contacted several computer vendors, including GE and IBM. At the time, IBM did not consider the demand for a time-sharing computer to be large enough to invest in. GE, on the other hand, committed to developing a time-sharing computer, so MIT chose GE as its supplier. In May 1964, a GE computer was used for a demonstration of a time-sharing system at Dartmouth College [5]. This probably became a wake-up call for IBM, especially when Bell Labs announced that it needed a similar system [6]. In response, IBM designed the CP-40 OS, which provided an environment for up to 14 simultaneous virtual machines [7]. Although this OS was never commercially distributed (it was only used in laboratories), it is important for being the first OS to use complete virtualization and being the forerunner of the CP-67 and the famous VM/370 [8], which was used with one of IBM's best-known mainframes, the System/370.

Each user of these systems had a terminal, a keyboard and a mouse, the whole set being an input/output device that was connected to the mainframe, the central computer where all the calculations were performed. Due to economic constraints, and because virtualization allowed it, companies used to use a single mainframe and provide a terminal to each employee, instead of providing a computer to each employee [9].

In 1974, Gerald J. Popek and Robert P. Goldberg put on paper the characteristics that a system had to fulfill to support virtualization [10]. Their article described the properties and functions of virtual machines and virtual machine monitors that we still use today. According to its definition, a virtual machine (VM) can virtualize any hardware resource, including processors, memory and network connectivity. A virtual machine monitor (VMM), more commonly known as a hypervisor, is the layer of software that provides the execution environment for the virtual machine. In their article they also described the three properties that a VMM had to satisfy:

- **Equivalence**: The environment that it provides to the virtual machines must be identical to the hardware that is being virtualized. Thus, a program running on the VMM must behave in the same way as if it was running directly on the physical machine.
- **Resource control**: The VMM must have full control over system resources.
- **Efficiency**: If possible, there should be no difference between a virtual machine and a physical equivalent.

These properties are still valid today, although the term Virtual Machine Monitor is no longer so common, being gradually replaced by the term *hypervisor*.

Over time, throughout the 1980s and 1990s, as Moore's prediction continued to hold true [11], computers became increasingly powerful, cheap, and small. Personal computers appeared to replace mainframes and terminals [12] [13]. The emergence and rise of personal computers changed the direction of computer development and meant that, for some time, the need for virtualization was no longer so urgent and its development was somewhat slowed down. However, in recent years we have experienced another boom in virtualization technologies due to different reasons, some of which are:

- **Resource optimization**: The great power of today's computers means that, in many cases, they are idle most of the time, since the use required of them does not consume all their resources. Virtualization allows several applications, each one even with different operating systems or execution environments, to run on the same hardware in isolation and without interaction between them. This has allowed to optimize hardware resources in several types of applications [14] [15].
- **Isolation as a security measure**: One of the great advantages of hypervisors is the isolation among virtual machines so that, ideally, malicious activity on one virtual

machine does not affect the rest [16]. Also, virtualization provides a way to implement application redundancy without having to purchase additional hardware. If one application fails, another application (running on a different virtual machine) can take over.

- **Physical space reduction**: The fact that virtualization makes it possible to use fewer hardware resources allows to save physical space.
- **Less power consumption**: Like the previous point, this is a direct consequence of the resource optimization. The fact of being able to use fewer hardware resources for the same functionality can lead to less power consumption, and eventually to a smaller carbon footprint [17] [18].
- **Easy migration and legacy protection**: Hypervisors, by definition, decouple the OSs and applications of the host hardware, thus benefiting the migration of virtual machines from one host to another without disruption. This is a great advantage when it comes to efficient work-balancing or when designing plug-and-play systems [19].

2.1.2. Hypervisors

As explained above, VMMs have their origin in the 70s, and their birth responds to very specific needs. Today, VMMs allow us to take full advantage of the capabilities of new processors, which are becoming more and more powerful. The term Virtual Machine Monitor has progressively stopped being used, and today it is much more common to refer to them as *hypervisors*.

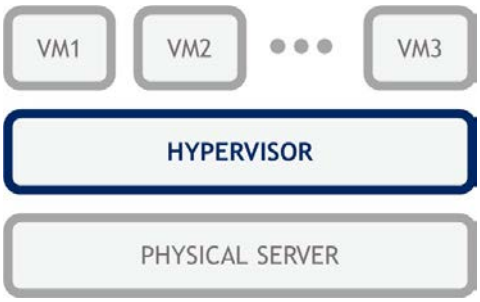

Figure 2.1: Graphical depiction of a hypervisor

Ankita Desai et al. [20] define a hypervisor as a thin software layer that provides abstraction of hardware to one or several operating systems, by allowing them to run on the same host hardware. Indeed, as shown in figure 2.1, a hypervisor is a layer of software that creates and manages virtual machines or partitions, to which it provides abstraction of the hardware. It is more debatable whether an operating system that uses the hardware virtualized by the hypervisor is necessary: as we will see later, there are hypervisors that make the virtualized hardware directly available to users, allowing them to manage

it as if it were real hardware. Programming an application on these hypervisors would be equivalent to programming what is commonly known as a "bare-metal application", except that the hardware on which it runs is virtual, not real.

Hypervisors can be classified into two types depending on the environment in which they run:

- **Type I hypervisors** or bare-metal hypervisors, in which the hypervisor runs directly on top of the host hardware. In this, the hypervisor has the responsibility of scheduling and allocating system resources to each of the virtual machines, and no operating system runs below it.
- **Type II hypervisors**, in which the hypervisor runs as an application on top of a host operating system. The host OS does not necessarily have to know about the hypervisor, it treats it as any other process.

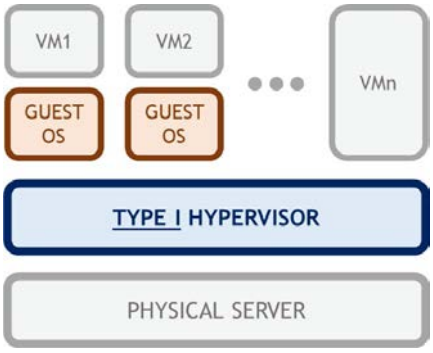

Figure 2.2: Type I hypervisor

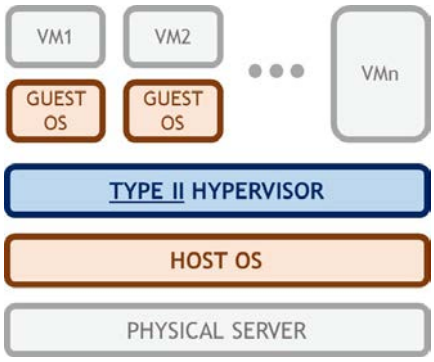

Figure 2.3: Type II hypervisor

Hypervisors can also be divided into two types based on the type of virtualization they offer:

- **Full virtualization** hypervisors allow running unmodified guests operating systems. The hypervisor completely emulates the physical platform it runs on, so the operating systems running on it don't even know they are running on a virtualized platform [21]. The great advantage of this approach is the flexibility it offers, by allowing any guest OS to run. However, it tends to carry a significant overhead, with up to 30% more latency when compared to running directly on physical hardware [22].
- **Paravirtualization** hypervisors cannot run unmodified OSs, but the guest OS must be aware that it has been virtualized and provides special hooks to directly take advantage of the services offered by the hypervisor [22]. In other words, the hypervisor does not have to translate the instructions of each VM, but receives direct instructions from it, usually called *hypercalls*. Of course, this alternative is much less flexible, since each guest OS must be modified to work with the hypervisor on

which it runs, but it has the advantage of offering higher performance in terms of access time to hardware, as theorized and demonstrated by Dordevic et al. in an article comparing the two virtualization techniques [23].

Some modern processors offer hardware tools to achieve (ideally) full virtualization, reducing the overhead of classic full virtualization. This particular case goes by different names: Xen calls it hardware virtual machine (HVM) [24], but it is generally referred to as hardware-assisted virtualization or accelerated virtualization [25]. Sometimes, this type of virtualization continues to have a too large overhead and is combined with some paravirtualized drivers, which is why it is also referred to as hybrid virtualization [26].

2.1.3. Virtualization in Safety-Critical Embedded Systems

As explained above, virtualization is one of the most powerful tools in the present and near future for the efficient use of modern multicore platforms. However, although it is generally accepted that the future of electronic systems is multicore technology, sectors with critical security requirements (such as the space, aviation, or automotive sectors) have traditionally been reluctant to adopt this technology. For example, despite the fact that multicore systems began to come into existence in 2005, 2008 is the first year in which the subject is directly addressed in two articles by the American Institute of Aeronautics and Astronautics (AIAA): one on Multiple Levels of Independent Security (MILS) [27] and another dealing with the future of jet fighter mission computers [28]. Most probably, the reason behind this slow permeation of multicore technology in critical sectors is related to interference problems between cores that need to consume the same resources and how this affects the Worst Case Execution Time (WCET). Years ago, works such as those of Kinnan et al. [29] and Wilhelm et al. [30] already pointed out some of the problems of shared resources, how they cause variability in execution times and how this un-predictability impacts the implementation and certification of these systems. A decade later, bounding the WCET to obtain deterministic behavior remains one of the main challenges in, for example, avionics platforms, as reflected in the work of Annighoefer et al. [31] These problems can be mitigated, in many cases, by intelligently planning architectures to improve predictability, as the works of Cullman et al. [32] and Kliem and Voigt [33] point out.

However, it is essential to address the problems that multicore processors poses, as they are becoming impossible to avoid in the present and future of critical real-time systems. In their article [34], in which they review the challenges of the future in terms of avionics architectures, Bieber et al. explain how multicore architectures have been replacing monocore architectures since the mid-2000s, so that monocore processors are progressively less common and more expensive. In addition to economic reasons, it is clear that monocore processors have a ceiling when it comes to computing power. The power consumption and the heat dissipated by the processing units as they become more powerful is specially problematic. This means that the future of avionics and other sectors

of critical needs inevitably pass-through multicore processors. For this reason, software tools are needed to help take advantage of their processing capacity, maintaining high levels of safety and security. Among these tools, virtualization has emerged as one of the most powerful, gaining popularity even in technologically conservative sectors such as automotive or avionics.

Virtualization in the Aviation Industry

From Federated Architectures to Integrated Modular Avionics

Traditionally, flight systems have had a *federated* architecture, in which each function consists of a black box with dedicated computational resources [35].

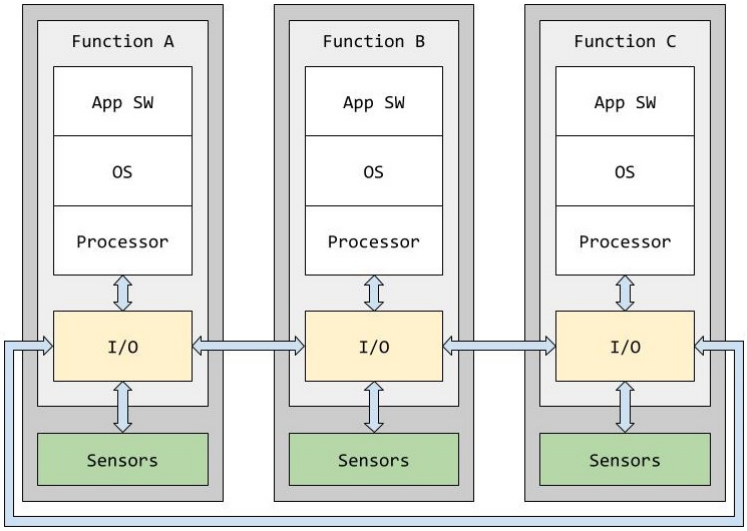

Figure 2.4: Federated architecture example

Each of the black boxes can contain a completely different configuration (hardware or software) inside, and they are physically isolated from each other. Naturally, this architecture is excellent in terms of fault isolation and fault tolerance, but it presents a series of serious problems: duplication of resources, lack of flexibility (adding a functionality requires adding a new box) and high cost, not only economic, but also regarding power consumption and weight. It is in this context that the concept of Integrated Modular Avionics (IMA) was born, in the early 1990s [36]. In an IMA architecture, the coexistence of different avionics functions on the same platform is pursued, without interference between them. For this, the different functions share a series of hardware resources (CPU, communications, I/O devices...) and are separated by robust partitioning mechanisms inherent to the architecture itself [37]. Today, the advantages of using IMA are not discussed in the industry and virtually every airplane model that enters service uses this philosophy [38]. One of the key enablers for this paradigm shift was the release of the ARINC 653 software standard in 1996.

ARINC 653

ARINC 653 is a software specification for time and space partitioning in safety-critical real-time systems. The first draft describing ARINC 653 was published in 1996 [39], and two supplements have since been published, the most recent being in 2007. Originally, it defined the general structure that the operating system of an IMA architecture should follow, but ARINC 653 can also be applied to a hypervisor, since some of its characteristics fit even better with it.

The objective of ARINC is to specify the characteristics of a software execution environment in which several applications can run, separated from each other in virtual containers called partitions. Ideally, these containers should be perfectly isolated from each other, so that the execution or failure of one partition does not affect any other partition. The way to achieve this isolation is to separate the hardware resources spatially and temporally. The similarity between ARINC 653, in this regard, with the hypervisor concept is remarkable, and this is reflected in the equivalence between figures 2.1 and 2.5.

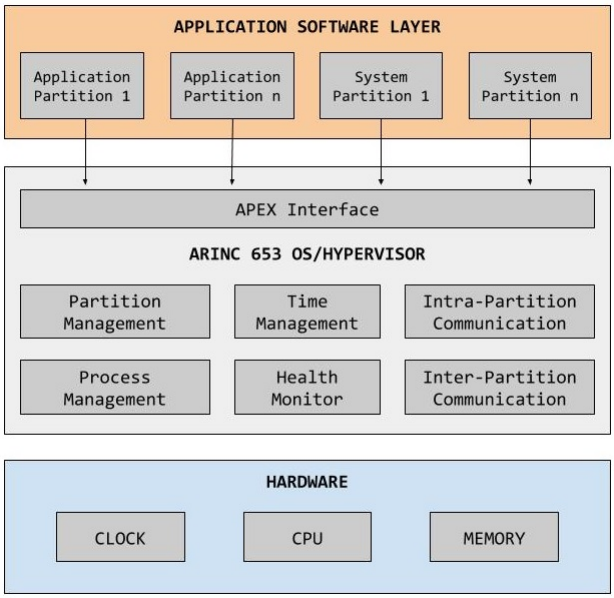

Figure 2.5: Graphical depiction of the ARINC 653 standard

In addition to defining the services to be offered, one of the most important features of ARINC 653 is that it standardizes the interface between the hypervisor/OS and the application layer. This interface is called APEX (Application/Executive), and it offers several advantages in line with the IMA philosophy: portability, reusability, modularity and easy integration of software blocks [40].

The services offered by ARINC 653 can be divided into several modules:

- **Partition Management**: This module provides means to modify the operating mode of partitions, and is in charge of scheduling the partitions.

- **Process Management**: Each partition can have multiple periodic or aperiodic processes. This module provides means for modifying the operational mode of processes, and it includes process scheduling.
- **Time Management**: This module ensures that hard real-time requirements are met and provides time-related services, such as reading time or wait/timeout services for processes. There must be a single time source for all partitions, regardless of their execution.
- **Inter-Partition Communication**: Communication among the different partitions is carried out through ports and channels. Conceptually, a port could be seen as a gate at the borders of a partition, while channels link two or more ports. The standard also defines two modes of operation: sampling mode (oriented to fixed-size synchronous messages) and queuing mode (oriented to asynchronous messages of variable size).
- **Intra-Partition Communication**: This module is in charge of providing communication among processes inside a partition, which is realized through buffers and blackboards. Events and semaphores are also used, to provide synchronization among processes.
- **Health Monitor**: This module is responsible for defining, detecting and reacting to different errors at the process, partition or system level.

In their article VanderLeest et al. [41] make an interesting reflection on how the ARINC 653 standard seems to prohibit interruptions, since they can undermine the determinism of a system, allowing one partition to steal time from others. Indeed, most implementations of ARINC 653 do assume that interrupts are not allowed when following the IMA philosophy, but they reason that interrupts do not have to involve non-determinism, and they design interrupts that offer predictability via a decreasing time budget, so that the system is predictable at the scale of major time frame. They also developed a prototype to demonstrate how these interrupts work using the Xen hypervisor, although the hypervisors used in safety-critical use cases will be reviewed in more depth in section 2.1.4.

The hypervisor concept fits really well with the IMA philosophy and the ARINC 653 standard. Virtualization provides, by definition, time and space partitioning. The rest of the functionalities that ARINC 653 describes (for example, health monitoring or the APEX interface) must be implemented, either in the hypervisor itself or at other levels of the software architecture (typically, the application interfaces, such as APEX, POSIX or OSEK are covered at OS level). In a 2015 article [42], VanderLeest et al. conclude that hypervisors are the tool that will allow the aviation industry to firmly adapt to multicore systems, which are the only way to increase processor performance. Moreover, ARINC 653 is also one of the most powerful candidates to standardize the space industry as well,

as the requirements of the civil aviation world that prompted the definition of the standard are also applicable to the space industry [43]. The adoption of ARINC 653 would bring benefits in terms of cost reduction, modular certification, and less integration effort.

The DO-297 standard (IMA Development Guidance and Certification Considerations) is the document used by certification authorities such as the FAA and EASA to approve aviation IMA systems [44]. In turn, this document recommends ARINC 653 to define the interfaces and specify the behavior of the system. Later, in section 2.2, we will briefly explain some of the software standards used not only in aviation, but in other safety-critical sectors such as space and automotive.

Virtualization in the Space Industry

The space industry has always been closely related to the aviation industry, so it is reasonable that there are numerous articles that discuss the possibility of applying aviation standards and practices to the aerospace industry. An example is the article by Windsor and Hjortnaes, in which they analyze the advantages of incorporating TSP techniques into the spacecraft avionics architectures based on the IMA aeronautical concept and the ARINC 653 standard [45]. They also consider the areas where these techniques could have the most impact and give different examples of use cases.

Actually, in some respects the operation of space systems is even more delicate than that of military and commercial aircraft. The success of space missions depends on being able to obtain deterministic behavior over long periods of time and, in many cases, with very limited real-time support from the ground operating teams. In addition, the environment in which a space system operates can be significantly more hostile than that of other critical real-time systems, due to factors such as vacuum, radiation or extreme temperatures. All of this makes the processes to design, develop and test space equipment time-consuming and costly. Likewise, the avionics in such equipment often consist of old and robust components, often created specifically for use in the space industry (such as LEON processors [46]), which have proven their reliability over years of successful missions.

For the same reasons as in aviation (increased complexity of software applications, increased processing capacity of hardware platforms, promotion of interoperability and reusability...), in recent years there is also a trend in the space sector towards COTS components, in order to take advantage of the powerful resources they offer, while reducing development costs, power consumption and physical space on the spacecraft. This has led to the emergence of a new space market, commonly referred to by that name: *New Space* [47]. This new market is based on the development of small space platforms that have a significantly lower associated cost than classic space missions and, therefore, less stringent reliability requirements. Both the reduction in cost and the openness to commercial components have led New Space to welcome private companies [48], something not common in the space industry, which has traditionally been driven by public organi-

zations. The entry of private companies has considerably increased the number of players in the industry, which has boosted its competitiveness and the emergence of new low-cost, high-performance applications such as space debris removal, Earth observation missions or satellite-based global communication networks.

It is clear that there is a problem in combining all or most of the functionality of a space system on a single high-capacity hardware platform: there are safety-critical functionalities whose failure would have a catastrophic impact on the mission objective, while other functionalities are not so critical and can therefore be subject to less demanding and less costly verification and validation processes. In this context, virtualization is a key technology since, as explained in section 2.1.1, it guarantees by its inherent characteristics the temporal and spatial separation of different software modules, so that there is no interference between them and the failure of one does not affect the others. When virtualization is used to deploy functionalities of different criticality levels on the same platform, the resultant systems are usually referred to as mixed-criticality systems [49] [50].

Virtualization in the Automotive Industry

Since the appearance of the first electronic ignition systems in the 1970s, the number of electronic components in cars has constantly increased. Whereas previously mechanical systems accounted for most of the complexity of cars, electrical and electronic systems have become more sophisticated over time [51]. In fact, it is currently estimated that 35% of the cost of a vehicle corresponds to electronics, greatly exceeding the 20% that it supposed in 2000 or the 30% that it supposed in 2010, and it is estimated that this figure will continue to grow up to 50% by 2030, according to a report by PriceWaterhouseCoopers [52]. Today, most vehicles are based on a powerful electrical and electronic (E/E) architecture on which the engine, braking, steering and other comfort and safety features depend. Embedded software has begun to gain paramount importance in automotive design and development, and it will continue to gain importance as companies continue to develop the technology, especially in areas such as autonomous driving. This means that the number of ECUs (Electronic Control Units) is increasing in modern vehicles: a current car has more than 150 ECUs for different purposes [53]. Faced with this challenge, different solutions have been proposed in recent years. Among these, virtualization is one of the most studied [54], since it would allow combining several ECUs on the same hardware platform, maintaining a temporal and spatial separation between them. This separation is essential, since critical systems such as driving assistance, which are usually managed by safety-certified RTOSs, coexist in the same vehicle with information and entertainment systems, which have significantly more lax safety constraints [55]. Using virtualization, the functionality previously distributed among many ECUs can be collected in a smaller number of DCUs (Domain Control Units), each system adhering to its own safety requirements [56].

Although not as direct as between the space and aviation sectors, there are many simi-

larities between these and the automotive sector: Gaska and Chen [57] make an interesting analogy between the architecture of an automotive system and an IMA avionics architecture. They discuss how multicore processing, along with hypervisor technology, are key enablers of the future of these types of systems, and propose various candidates for both multicore platforms (Intel Xeon FPGA SoC, NVidia Tesla SoC or Xilinx Ultrascale+) and hypervisors (VxWorks, Lynx or Green Hills). However, the article does not show any type of test or deep analysis that allows us to opt for any of the alternatives.

2.1.4. Hypervisors for Safety-Critical Real-Time Systems

Having explained the basis for understanding the hypervisor concept and some of the reasons why virtualization not only has a place, but may play a fundamental role in the future of several safety-critical industries, the following section includes an exhaustive survey in which the existing scientific literature is reviewed in search of evidence of the use of hypervisors in safety-critical embedded systems, in order to be able to qualitatively compare the different hypervisors used. The search criteria used to select these scientific articles are described below.

Paper Selection Criteria

The interest of the research topic for the community of real-time systems is undoubted. The term hypervisor is very popular in Google trends being around 75% in the last 10 years. As we want to restrict the overview on hypervisors to safety-critical systems, we have searched for a term consisting of the word *"hypervisor"* together with different words related to industries in which safety-critical embedded systems are developed in the most popular and relevant databases of scientific and engineering research articles: *IEEE Xplore* and *Science Direct*. Table 2.1 shows the results obtained.

Table 2.1: Number of papers resulting from the search of the word "hypervisor" and words referring to safety-critical sectors

	Automotive	**Aerospace**	**Aviation**	**Avionics**	**Safety-Critical**	**Spacecraft**	**Aircraft**	**Vehicle**
IEEE Xplore	44	29	6	22	37	4	5	24
ScienceDirect	144	68	39	71	117	22	64	322
Total	188	97	45	93	154	26	69	346

Note that not all the articles and books resulting from the search are of interest to this survey, as some only mention the hypervisor concept in a context not directly related to this technology. However, sorting the search results in chronological order clearly shows the upward trend in the importance and depth of hypervisors in safety-critical embedded systems. After eliminating duplicate results, Table 2.2 presents the distribution of articles and books by year and Figure 2.6 shows them graphically.

These results are supplemented with searches in Google Scholar, to complement the

Table 2.2: Distribution of the results of the previous search by year

2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022
2	1	2	4	3	1	8	10	22	18	27	40	36	41	48	55	66	74

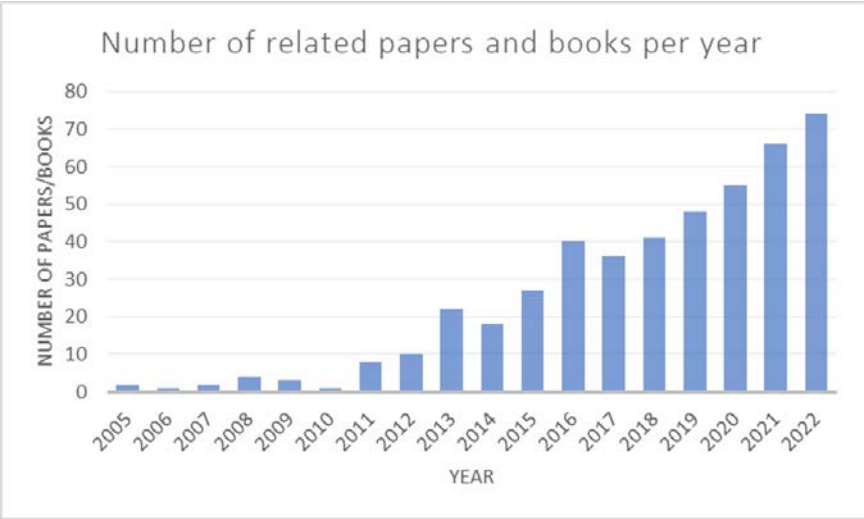

Figure 2.6: Number of papers published each year according to the previously defined search

articles found in these two large databases with the most relevant articles found in smaller databases. From all these results, only journal articles and conference proceedings are selected, discarding books, which usually collect valuable technical information but do not usually present new information that has not been previously presented in articles. Finally, among the remaining articles, a review is carried out to select only those articles that meet one of the following criteria:

- The article deals with the use of an existing hypervisor (open-source or proprietary licensed) in the context of a safety-critical embedded system.
- The article presents modifications made to an open-source hypervisor for use in the context of a safety-critical embedded system.
- The article presents the development of a new hypervisor for use in safety-critical embedded systems.

The criteria for selecting the articles to be reviewed in this survey are reflected graphically in Figure 2.7.

The selected jobs and the hypervisors used in them are discussed in detail below. Because of their importance and the greater existence of evidence in this regard (in scientific literature or through dissemination by private companies), work using Xen (2.1.4), KVM (2.1.4) and XtratuM (2.1.4) is discussed in dedicated sections. The rest of the hypervisors are grouped together in section 2.1.4. Previously, however, section 2.1.4 introduces

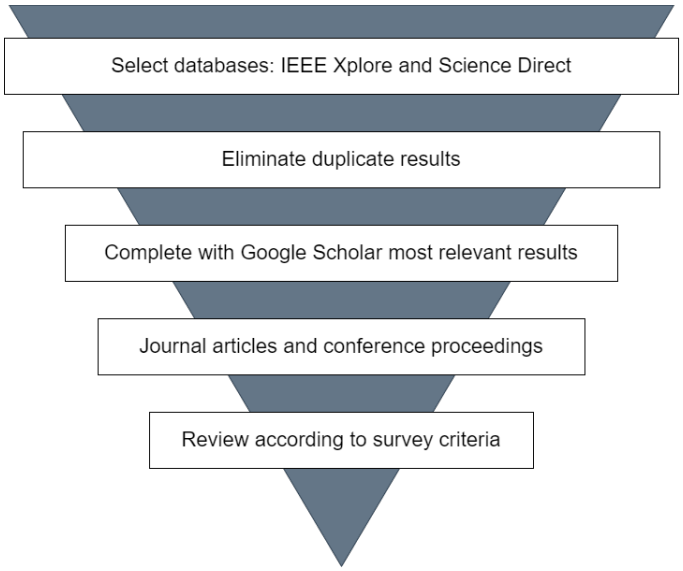

Figure 2.7: Articles selection criteria

some hypervisors that have great importance and presence in the industry but, due to their proprietary licensing and high price, are hardly studied in academic works.

Proprietary Hypervisors

Note that although a hypervisor inherently offers TSP features, it is not the only option to implement them. An RTOS can offer TSP features if it is able to fully isolate software modules, so each of them can run different criticality partitions. For example, there are a number of proprietary RTOS, oriented to safety-critical real-time industrial applications, that offer TSP features and have already been certified according to standards such as DO-178C or ISO 26262 (some details on these and other standards are given in section 2.2, so that the importance of complying with them is understood when the developed software is actually to be used in real use cases). It is the case of WindRiver VxWorks, Green Hills Integrity, LynxOs, SYSGO PikeOS, JetOS or DDCI Deos. In some cases, given the increasing complexity of these types of applications and the increase in processing power in current hardware platforms, they have ended up including virtualization capacity, or resulting in the appearance of new virtualization products from the companies that develop them. These products are:

- WindRiver Helix
- Green Hills Integrity Multivisor
- LynxSecure Separation Kernel Hypervisor
- SYSGO PikeOS
- DDCI Deos

- RTS Hypervisor

These hypervisors have a proprietary license and are aimed at critical real-time systems such as aviation or automotive, so they are quite expensive. This makes their use confined almost exclusively to commercial purposes, so there is little published literature on them. Therefore, the information compiled in the following table, which compares each of these options, has been compiled mainly from sources provided by the developers themselves. In addition, it must be taken into account that these hypervisors are all developed by large companies and are direct competitors, which is why they offer similar functionality in features such as inter-partition communication, real-time support or safety and security services. Therefore, a comparison between the hypervisors listed above is made below, based on criteria by which they differ:

- **Hypervisor Type**: As explained in section 2.1.2, hypervisors can run directly on the hardware (type 1 or bare-metal) or on an OS host (type II).
- **Supported HW Architecture**: Although it is not an exhaustive analysis (the compatibility case with each different processor or SoC could be studied), the processor architectures supported by each of the hypervisors are listed.
- **Virtualization Type**: As explained in section 2.1.2, hypervisors use different virtualization techniques that can be roughly grouped into two types: paravirtualization and full-virtualization.
- **Supported guest OSs**: The different operating systems that can be run as guests on each of the hypervisors. Note that, in the cases of hypervisors that offer full-virtualization, any operating system can be run unmodified, although it is usually more efficient to run a paravirtualized operating system if possible.
- **Nationality of the developer**: This is an important aspect to take into account, since there may be cases in which an original equipment manufacturer faces restrictive regulation when it uses the product of a company from another country, in the event that there is competition between both countries (as sometimes happens with the USA, Russia and many European countries).

The comparison is reflected in Table 2.3.

Xen

Xen is a type-1 hypervisor, originally developed by the University of Cambridge Computer Laboratory, that is now being developed by the Linux Foundation, with support from Intel. Xen is free and is one of the few type-1 hypervisors that is available as open-source. It is also, by far, the hypervisor on which most literature and research are based,

Table 2.3: Proprietary hypervisors comparison

	Hyp. Type	Supported HW Architectures	Virtualization Type	Supported Guest OS	Developer Nationality
WindRiver Helix	1	ARM x86	Full Virtualization	Any unmodified OS	American
Green Hills Integrity Multivisor	2	ARM x86	Full Virtualization	Any unmodified OS	American
LynxSecure Separation Kernel	1	ARM PowerPC x86	Paravirt. Full Virtualization	Any unmodified OS (fully virtualized) LynxOS and Linux (paravirtualized)	American
SYSGO PikeOS[a]	1	ARM PowerPC x86 SPARC	Paravirt. Full Virtualization	Any unmodified OS (fully virtualizated) PikeOS and Linux (paravirtualized)	German
DDCI Deos[a]	2	ARM PowerPC x86	Paravirt.	RTEMS	American
RTS Hypervisor	1	x86	Full Virtualization	Any unmodified OS	German
OpenSynergy's COQOS	1	ARM	Full Virtualization	Any unmodified OS	German

[a]Note that in the case of PikeOS and Deos, the same product offers typical RTOS functionality and virtualization capabilities.

so this section is dedicated to briefly describing its characteristics and analyzing the results of the main research works with respect to Xen, especially those related to real-time safety-critical systems.

As other hypervisors do, Xen allows to run many instances of an OS (or different OS) in parallel on a single machine. A running instance of a virtual machine is called a domain or guest. However, although the hypervisor is the first program running after exiting the bootloader and runs directly on the hardware, it needs a special first domain (called *domain 0* or *dom0*) that has specific privileges and is responsible for controlling the hypervisor and starting other guest domains. These other domains are called *domUs* (because they are *unprivileged* domains, in the sense they cannot control the hypervisor or manage other domains). *Dom0* has direct access to the hardware, so the hypervisor does not contain device drivers. Instead, the devices are attached to *dom0* and use standard Linux

drivers. *Dom0* can then share these resources with the rest of the domains. Although Xen is not as vulnerable to single-point failures as a type II hypervisor (in which the host OS crash automatically causes the guest OS to crash), the fact that most physical resources reside on *dom0* means that, if there is a failure in *dom0*, the rest of the system will lose communication capabilities, both externally and between domains. Even if their functionality is degraded, the *domUs* could remain operational even after the failure of *dom0*, but any additional failure in the *domU* would be unrecoverable, since *dom0* would not be able to reboot it. You could return the system to its initial configuration by rebooting *dom0* but obviously doing so would interrupt system availability for mission-critical applications. There are methods to mitigate this problem by enhancing the autonomy of *domUs*. For example, a *domU* can be given direct hardware access through Xen's pass-through virtualization feature.

Xen was originally developed for x86 processors in 2003, and for that architecture it provides both paravirtualization and full virtualization. The porting of Xen to ARM involved major changes in its architecture and, as a result, its code for ARM architectures is considerably smaller than that of x86 architectures but, as a main limitation, it stands out that Xen for ARM only supports paravirtualization [58].

After this brief explanation of Xen, the most important works that use this hypervisor, especially those that apply it to safety-critical embedded systems, are listed below.

VanderLeest et al. [42] use Xen over the Zynq UltraScale+ MPSoC as a case study. Although the Zynq Ultrascale+ contains a quad-core A-53 processor, as a possible solution to the problem of shared resources, they propose a simplification in which different partitions do not run simultaneously on different cores. This prevents interference in, for example, access to the L2 cache or memory bus bandwidth. However, this simplification does not allow the efficient use of the Zynq Ultrascale+ large processing capacity. In the event that more than one partition needs to access the same I/O resource, the article proposes two ways to do it: via software (although this can be a bottleneck, limiting bandwidth or latency) or via hardware, implementing an arbitrator in the FPGA of the MPSoC. It must be taken into account that the certification of arbitration logic must be done at a level equivalent to the highest level of criticality of any of the serviced guests (if implemented in software, in accordance with the DO-178C standard; in case of being implemented in hardware, in accordance with the DO-254 standard). The possibility of using the LynxSecure Separation Kernel Hypervisor and the Mentor Graphics Multicore Framework is also discussed in the article, although neither of these options is analysed in depth.

In their article [59], Daniel Sabogal and Alan D. George explain the development of a framework called Virtualized Space Applications (ViSA). ViSA leverages the capabilities of the Xen hypervisor to provide a safe environment (software-based fault tolerant) on the Zynq Ultrascale+ and improve the dependability and availability of a flight system. The framework manages to improve the system in these aspects but presents problems when

the APU of the Zynq Ultrascale+ is irradiated. In addition, there is work to be done to solve one of the main problems of Xen-based systems, which is the dependency on dom0. Currently, no article has been published that continues this work.

In 2010, DornerWorks introduced a prototype implementation of the ARINC 653 standard, extending the Xen hypervisor [60]. In a 2013 article, VanderLeest and other DornerWorks researchers explain these ARINC 653 extensions made on Xen a little more in depth and name the resulting hypervisor ARLX (ARINC 653 Real-time Linux on Xen) [61]. A few years later, ARLX was renamed Virtuosity, and VanderLeest announced that they were adapting the hypervisor to FACE conformance [62]. Today, Virtuosity remains an open-source hypervisor and DornerWorks benefits from it by offering maintenance and support. Virtuosity presents the limitations of any Xen-based hypervisor, especially in terms of certification: m the control partition (dom0) runs Linux as a guest OS. Despite its widespread use, it is very difficult to certify Linux according to the most demanding aviation or automotive standards, due to the little documentation on some of its main components, as well as the great effort involved in certifying an operating system so large.

Kistijantoro and Gilbran extended the ARLX partition scheduler to use the primary-backup scheme, so that the scheduler can guarantee certain services even in the event of partition failure, using backup partitions [63]. Although they demonstrated that this method improved the overall reliability of the system, it still presented several significant problems: it resulted in an unacceptably high maximum latency, the scheduler was not able to autonomously detect the failure of a partition (it depended on the partition being capable of reporting it) and did not consider the deadline of each process within a partition.

Bijlsma et al. [64] propose a safety mechanism for autonomous vehicles that, among other tools, uses Xen to isolate software modules in the same SoC. During their experiments, using Xen's Null scheduler, they measured the time it took for the hypervisor to shut down a faulty VM, and found that it was much longer than the time it took to pause it, so they opted for this option. Even so, although the mean time in 2000 experiments was 55us (acceptable to avoid a collision and prevent the fault from propagating), they obtained outliers of up to 1.5ms. This large variation indicates that a more deterministic scheduling is needed in the hypervisor, to be able to use it in such critical functionality.

Karthik et al. used Xen to offer an integrated cockpit solution, in which four different automotive systems ran on the same heterogeneous SoC [55]. However, it should be noted that Xen was used (on an ARM Cortex-A15 processor) only to run three of those systems, which used Linux and Android and were not safety-critical. A microcontroller (ARM Cortex-M4) and an RTOS were used exclusively to run the other system, which was safety-critical, without any kind of virtualization.

Xen on ARM has also been used to integrate Linux with a real-time operating system such as ERIKA OS, which obtained the OSEK-VDX certification for automotive applications, on the same platform [65]. However, this option still has many limitations in

terms of certification: the deployment architecture is very specific (a dual-core platform in which each OS runs on a different core) and ERIKA OS can run only as a guest domU, so it depends entirely on the privileged domain dom0, which runs a general-purpose Linux. As in previous examples, there is also the challenge of certifying the Xen hypervisor itself, which does not seem approachable in its current form for meeting DAL A/B/C according to the DO-178C standard or an automotive-grade standard such as ASIL.

Recently, Schulz and Annighöfer conducted an empirical study to test the suitability of Xen to operate on safety-critical real-time systems [66]. In their experiments they obtained some promising results, but in some scenarios, they observed unpredictable behaviour in terms of latencies and execution times. Although further research is needed, they conclude that Xen is not realistically feasible for such systems in its current state.

KVM

KVM is an open-source hypervisor originally released in 2007 for the x86 architecture and, since 2012, ported to the ARM architecture. KVM is integrated into the Linux kernel (since its version 2.7.20 for x86 and its version 3.9 for ARM), so it takes advantage of a large part of its functionality, such as memory management or CPU scheduling. In fact, as Dall and Nieh explain [67], although in x86 KVM resides entirely in the kernel, in ARM it is divided into two parts: one (called Highvisor) that resides in kernel space and corresponds to most of the hypervisor's functionality, and another (Lowvisor) that resides in Hyp mode and is in charge of enforcing isolation and performing the context execution switches between VMs and host. This is because trying to implement KVM entirely in the host kernel would have involved a series of modifications on the kernel that would have been negative in terms of performance and portability. A big advantage of this approach is that porting KVM from one ARM platform to another is easier than with a bare-metal hypervisor like Xen. Since ARM platforms are not very standardized and it is very common for them to support a version of Linux higher than 3.9, this advantage is key when compared to Xen [58].

KVM does not offer virtualization of hardware devices but relies on external tools that run in user space, such as QEMU. Together, KVM and QEMU allow running unmodified guest OSs [68]. However, to avoid the overhead that full-virtualization implies, there is also the possibility of running paravirtualized OSs using Virtio [69].

There are numerous papers analyzing Xen and KVM performance overhead on x86 architectures [70] [71] [72], but only a few that do so on ARM architectures and using embedded systems. Among them, perhaps the most complete is that of Raho et al. [58], who make a comparison in which they also include container technology (Docker, in particular). The conclusion they reach is that the overhead performance of any of the solutions is very small, with slight differences depending on the test run. They also analyze how KVM is more easily portable than Xen and how Docker, although fast and easy to deploy, is a less secure alternative to hypervisors, because hypervisors use hardware

extensions to offer greater isolation (VMs do not share kernel space, while containers do). However, in a recent paper Müller et al. measured the overhead of KVM on a self-driving car-oriented Nvidia Drive AGX SoC and concluded that KVM produces too high an overhead in this particular case, which makes it unusable in real-world use cases [73].

XtratuM

XtratuM [74] is a type-1 hypervisor originally developed by researchers at the Universidad Politécnica de Valencia and currently maintained by the Spanish company fentISS. XtratuM is targeted to real-time safety-critical systems, especially in the space sector [75], being designed based on the ARINC 653 standard. Currently, it supports Linux, RTEMS and LithOS (an operating system from the same developers) as paravirtualized guest OSs and allows to run bare-metal partitions using XRE, a minimal execution environment offered by the hypervisor itself. Unlike Xen, it uses no control partition (dom0): the hypervisor manages the partitions and its communications, IRQs and HW I/O access. The latest versions of XtratuM support SPARCv8, ARMv7 and RISC-V architectures. Although they are not up to date and their support is not continuous, there are also older versions of XtratuM for PowerPC [76] and x86 [75] architectures.

XtratuM can be downloaded under the GNU General Public License, although fentISS also offers a commercial version called XtratuM Next Generation (XNG). This version is currently the most widely maintained by developers and, although it offers similar functionality to the GPL version, its internal structure is significantly different.

ESA and CNES are two of the main promoters of XtratuM, financing projects and research in which its development is continued and the possibility of using it in space missions is being evaluated [77] [78]. However, the ARINC 653 oriented nature of XtratuM makes it a good candidate for aviation applications as well, although as of today the costly process to certify it according to the corresponding safety standards (such as DO-178C) has not started. Efforts have also been made to use XtratuM in automotive systems, such as the work carried out in the OVERSEE project, in which FreeOSEK (an RTOS OSEK/VDX-compliant) was ported as a guest OS on top of XtratuM [79].

Although it is a hypervisor widely used in private industry and, for competitive reasons, companies are sometimes not interested in disseminating knowledge about it, there are a few academic papers that mention or use XtratuM on safety-critical embedded systems. These are described below:

Larrucea et al. define a series of characteristics that a safety-critical hypervisor should meet to comply with the IEC 61508 standard, defined by the International Electrotechnical Commission, which covers the functional safety of electrical, electronic, and programmable electronic equipment [80]. In the same article, they map the defined features to the XtratuM capabilities, demonstrating how it successfully covers them.

Muttillo et al. carried out a series of tests, on different hardware platforms that used

both the LEON3 and LEON4 processors, in which they demonstrated that the performance of XtratuM competes with that of a highly proven hypervisor like PikeOS, improving it in some aspects (such as timing and memory access), although it is somewhat less predictable in the overhead introduced [81].

Researchers from the Korea Aerospace Research Institute made the effort to port a version of RTEMS that would support Symmetric Multi-Processing (SMP) on top of XtratuM [82] [83]. Currently, fentISS has released versions of XtratuM with which bare-metal partitions can be run on the hypervisor in SMP, both on LEON4-based boards and on boards based on the Zynq-7000 SoC. In addition, it is developing a BSP that will allow to run a Linux SMP guest OS on Zynq-7000.

Campagna et al. [84] presented a prototype of an architecture in which the XtratuM hypervisor runs on a LEON3 processor. As they describe in their article, they run three partitions on top of XtratuM: two of them running a number-crunching application and a checker partition that checks the outputs produced by the other two. The purpose of the paper is to study the solvency of their solution to the injection of errors. The failure model they assume is Single Event Upset (SEU), which models the impact of ionizing radiation on the processor as a result of a memory bit flip. They show that using a hypervisor is an effective method for task segregation and scheduling, as well as error detection. The use of the hypervisor has an overhead close to the minimum possible overhead and it is capable of detecting 96.2% of SEU faults.

During a study on the robustness of the separation kernels, XtratuM was used on a LEON3 as a use case and 9 notable vulnerabilities were discovered that had not been detected during the validation campaigns of the hypervisor development [85]. This not only demonstrates the effectiveness of the error injection method used, but it was a considerable help to further strengthen XtratuM, as part of the hypervisor developer team was involved in the experiment.

XtratuM is also one of the foundations of XANDAR, a project that aims to provide a toolchain for developing safety-critical embedded systems [86]. The toolchain, developed by a large group of partners from industry and academia, has been tested in both avionics and automotive use cases.

Onaindia et al. propose an architecture oriented to real-time systems that monitors and it is able to reduce system power consumption [87]. The lowest-level component of this architecture is a hypervisor. For the prototype, because of its features and its support for Xilinx Zynq-7000 SoCs, XtratuM is used (and extended) as hypervisor, and the prototype is tested on two use cases of avionics and railway systems. XtratuM has also been used, on a representative computer system used in avionics, as the basis for building a feedback control mechanism implemented at the hypervisor level [88].

Other Hypervisors

Some research papers using or developing hypervisors other than Xen, KVM or XtratuM are listed and briefly described below:

Missimer et al.'s **Quest-V** [89] uses hardware virtualization for safe and secure resource partitioning, offering partitions (called sandboxes) that can run their own operating system, called Quest, or Linux. However, the work is currently discontinued and does not support ARM multicore platforms with hardware virtualization.

Rodosvisor is a type 1 hypervisor developed by Tavares et al. It supports paravirtualization and full-virtualization, and was tested on a Xilinx FPGA with a built-in PowerPC core [90]. It is inspired by the ARINC 653 standard, but it is not fully compliant with it, since it implements the services defined by the standard, but not strictly following the corresponding API. In the experiments in the article, two bare-metal (OS-less) partitions are deployed, and a third partition runs RODOS, an RTOS for embedded systems. However, apart from one article enhancing the hypervisor for integration into the POK operating system in 2016 [91], there are no other known articles describing a continuation of the work, porting the hypervisor to other HW platforms, or supporting new guest OSs.

Pinto et al. presented in 2016 a hypervisor called **RTZVisor** (Real Time TrustZone-assisted Hypervisor) oriented to space applications, which used the ARM TrustZone to provide virtualization on a Xilinx Zynq platform [92]. A few months later, they introduced two extended versions of the hypervisor, called μRTZVisor [93] and SecSSy [94], designed to increase the safety and security of its predecessor. These hypervisors have some interesting features, such as the ability to run different almost unmodified guest OSs, but they have other important limitations: they disable the caches and MMUs of the guest OSs, they are limited to ARM processors offering ARM TrustZone technology and, despite being tested on a multicore platform (Zynq ZC702), they do not support multicore processing, so they use only one of the processor cores.

Although RTZVisor is probably the most advanced of its kind, it is neither the only nor the first hypervisor to be based on ARM's TrustZone technology. Winter proposed in 2008 a method that, using TrustZone, provided a virtualization framework and implemented a prototype in which he deployed a non-secure guest in a secure Linux environment [95]. Cereia and Cibrario carried out a similar exercise, implementing a virtualization layer that allowed to deploy an RTOS and a guest OS, pointing out some of the limitations that ARM TrustZone imposed on them at the time: it only allowed the execution of two OSs and, while the guest OS did not it could access or interfere with the RTOS, it did not work in reverse, so it would not support two secure RTOS [96]. These limitations are shared by ARM TrustZone-based hypervisors proposed in later work, such as the Secure Automotive Software Platform by Kim et al. [97] or the open-source Xvisor presented by Cicero et al. [98] **VOSYSmonitor**, from Virtual Open Systems, also allows parallel execution of a secure partition (running an RTOS) and a partition without real-time guarantees (GPOS), but has the particularity that it allows the non-critical partition

(GPOS) to use another hypervisor (such as Xen or KVM), so it could be argued that it also supports multi-guest OS [99]. VOSYSmonitor gives full priority to the RTOS, allowing the GPOS(s) to run when there are no active tasks on the RTOS.

Dasari et al. conducted a series of experiments with the ETAS Lightweight Hypervisor (**LWHVR**), a commercially viable solution in the automotive industry, which they extend by implementing Reservation Based Scheduling (RBS) [100]. ETAS LWHVR is a hypervisor oriented to multicore microcontrollers, with a low overhead and memory footprint. One of the cores works as a master, and in it runs all the SW that has direct access to the HW. Different VMs can run in the rest of the application cores. This architecture makes the use of this hypervisor not viable in monocore systems and, probably, inefficient in the case of processors with few cores (such as dual-core processors).

Jailhouse is a simplicity-oriented hypervisor based on Linux. The hypervisor is implemented as a Linux kernel module, just like Xen or KVM. As Ramsauer et al. [101] explain, it does not perform any kind of scheduling, and simply provides static partitioning, directly allocating hardware resources to each partition. This has the advantage that legacy applications can be run with no active hypervisor overhead and simplifies certification efforts. For this reason, among others, Jailhouse is the hypervisor chosen to cement a computing platform called SELENE, which aims to serve as a basis for developing different safety-critical applications, from flight applications to autonomous robotics [102]. However, the fact of this hypervisor being based on Linux implies other complications in terms of safety and certification, such as the fact that it requires other software elements (UEFI Firmware code or bootloader, for example) that must be considered in the certification process. In addition, it still has limitations in essential aspects such as communication between VMs, for which it does not offer end-to-end timing guarantees. Some of these issues may be faced during the development of SELENE, which is scheduled to end in December 2022. Boomerang [103] is another proposal that leverages the features of a hypervisor to develop a system in which to run a critical partition along with a non-critical guest OS.

Bao [104] is also a proposal based on this concept of static partitioning, in which the hypervisor is freed from resource management, once the CPU cores, memory or I/O devices have been assigned to each guest OS. One of the limitations of this simplistic approach is that the number of guest OSs is limited by the number of physical CPUs, unless other virtualization technology runs on top of the static partitioning hypervisor.

OKL4 is a popular Type I hypervisor developed by Open Kernel Labs (the company was acquired by General Dynamic Mission Systems in 2012, and the hypervisor is no longer open-source), intended to be deployed in embedded systems. It is especially popular in the mobile phone industry (estimated to have been deployed in hundreds of millions of them [105]) and is capable of paravirtualizing various high-end operating systems, including Linux, Windows and VxWorks. In addition, it can offer a simple POSIX interface by itself, so it is able to function as a minimal OS for the implementation of safety

or security critical applications [106]. Although it does not support full-virtualization, OKL4 is able to take advantage of the virtualization extensions of some ARM processors to reduce the effort required to paravirtualize an OS.

seL4 is also a microkernel of the L4 family available, at different levels of maturity, for ARM, x86 and RISC-V architecture processors. Its development started in 2006, with the intention of providing a basis for secure, reliable, and safe systems. The kernel is open source, available under the GNU GPL v2 license, and most of the libraries and tools are under the BSD 2 clause. seL4 can run standalone as an OS with TSP capabilities, but it can also be configured as a bare-metal hypervisor on which, in addition to running native applications, Linux virtual machines can be deployed [107].

Another open-source hypervisor geared towards having a small footprint for its use in embedded systems is **NOVA** [108]. According to its developers, the hypervisor is the base of every other component of a system that uses it, so it should be as small and trusty as possible. However, unlike OKL4, NOVA offers full virtualization, thus, as explained above, it results in a slightly more complex and less efficient, yet more flexible hypervisor [105].

ACRN [109] is a lightweight hypervisor oriented to IoT and embedded systems. Although its architecture makes it quite flexible and allows multiple non-safety-critical VMs to be deployed, the number of safety-critical VMs is limited to two at best. Another important limitation is that it is based on Intel virtualization technology, so its supported HW is limited to some processors from this manufacturer. Among the guest OSs that it supports we can find Ubuntu, Android, Windows, as well as others more interesting in terms of safety, such as VxWorks and Zephyr.

Elektrobit also offers its hypervisor implementation, called Corbos, of which there is not much published information. It is known to be a microkernel-based hypervisor that allows at least Linux partitions to be deployed alongside other safety-critical partitions. Corbos is mentioned in an article by Lampka and Lackorzynski, to exemplify an automotive architecture that uses virtualization to harness the computing power of an ECU, without sacrificing the safety and security of the most critical software [110].

In 2014, Kim et al. introduced a hypervisor geared towards critical real-time systems, called QPlus-Hyper [111]. This hypervisor allowed the execution of an RTOS and a GPOS on the same platform, using the virtualization extensions present in some ARMv7 cores. However, there is no evidence that the development of this hypervisor has been continued since 2015, when this hypervisor was used to carry out a proof of concept that investigated how a GPU that is shared between several guest OSs could be virtualized [112]. Other than that, it is only briefly mentioned in a 2019 article that discusses cache-interference issues on clustered multicore platforms [113]. For this reason, and due to the lack of public information on the internal structure of the hypervisor, it has been decided not to take it into account in the comparison.

Reinhardt and Morgan evaluate a type I hypervisor called RTA-HV, developed by

ETAS Ltd and aimed at efficient use of resources in multicore systems. In their research, they paravirtualized guest OSs on the Infineon AURIX TC27X platform [114]. As highlighted in the article, a non-intrusive hypervisor has the added advantage that it acts as an abstraction layer, which encourages software reuse, facilitating porting to another platform. However, in the work presented in their article there are certain limitations, mainly due to the low virtualization support on the part of the HW used. For example, they did not achieve complete temporal isolation between partitions and could only do a one-to-one mapping between partitions and CPU cores. Despite this, they analyze and reason that hypervisors are a good solution to the problem of consolidating different systems in the same ECU.

Manic et al. [53] used Blackberry's QNX hypervisor to deploy two virtual machines, with different OSs, on the same ECU. The objective of the experiment was, in addition to being able to carry out independent processing in each of these virtual machines, to demonstrate that they can share a single graphic display without affecting either the performance or the safety of the vehicle. QNX is a Type I hypervisor certified to ISO 26262 ASIL D (in addition to the industry standard IEC 61508 SIL 3). It supports both safety (QNX Neutrino RTOS and QNX OS for Safety) and non-safety (Linux or Android) guest OSs, and can be deployed on the latest ARMv8 and x86-64 SoCs.

Lemerre et al., from the Atomic Energy and Alternative Energies Commission, extended the RTOS **PharOS**, adding a paravirtualization layer that allowed it to run Trampoline (an OSEK/VDX-compliant RTOS) partitions, which run as time-triggered tasks within PharOS [115]. Configured in this way, PharOS can be considered a type II hypervisor.

HTTM is a relatively recent type 2 hypervisor that offers full virtualization on MIPS architectures, so there is no need to use any hardware-specific extensions [116]. Its main weakness in terms of its application in safety-critical systems is that it must run on a Linux host, which limits its ability to deliver real-time performance, and that it currently only allows the creation of a guest VM. Even so, the work is still evolving, and there are recent papers evaluating and trying to improve the efficiency of HTTM [117]. Although it is theoretically open-source, we have not been able to find the source code in any repository and we do not know the license under which it is distributed.

Minos is an open source type 1 hypervisor that allows multiple VMs to be deployed on SoCs based on the ARMv8-A architecture. Through paravirtualization, it can host several Linux, Android and Zephyr guest VMs, and is oriented to IoT and embedded devices. There are hardly any research papers mentioning Minos, but details about its internal structure and source code can be consulted through its Github repository [118].

There are a couple of initiatives that propose hypervisors targeted at SoCs that cannot be considered Type 1 or Type 2, since they are implemented as another hardware module (implemented in the SoC's FPGA), rather than as a software layer. Developers of these hypervisors are often referred to as type 0 hypervisors. Janssen et al. are the first to

coin this term [119], although they did not go so far as to define a complete hypervisor, but rather a prototype running on a Microblaze core that allows bare-metal applications to be deployed isolated from each other. Jiang et al. developed BlueVisor, which does achieve this "type 0 hypervisor" in which all its components run in hardware, and allows paravirtualized guest OSs (FreeRTOS, uCOS-II and XilKernel) to be deployed on softcore processors at the highest privileged level [120]. However, due to the current immaturity of these initiatives and because their nature is different from that of the hypervisors reviewed in this survey, we leave these type 0 hypervisors out of the comparison.

2.1.5. Safety-Critical Real-Time Hypervisors Comparison

Having analysed the main hypervisors that have been applied to safety-critical embedded systems, and according to the information gathered from the cited sources (mainly academic research papers), this section compiles their characteristics and compares them.

There are different characteristics according to which hypervisors can be compared and ranked. Based on user requirements and their knowledge of embedded systems, Hamelin et al. propose a set of practical criteria by which any potential user could select the most suitable one for their application or system [121]. Since in section 5 virtually all hypervisors for use in a safety-critical real-time embedded system have been reviewed, it is interesting and potentially useful for future research to classify these hypervisors according to some of the parameters established by Hamelin et al:

- **Hypervisor Type** (type 1 or 2).
- **Supported HW architectures** (ARM, x86/64...).
- **Supported Guest OS** (Linux, RTEMS, FreeRTOS...).
- **Communication Services** (inter-partition communication).
- **API** (POSIX, OSEK, ARINC 653).
- **License**.

To these criteria we can add a few more, having reviewed the state of the art and seen where many of the hypervisors studied differ:

- **Virtualization Type** (full virtualization or paravirtualization).
- **Scheduling** (ARINC 653, no scheduling...).
- **Real-time partitions support** (and limitations, if any).
- **Developers' nationality/country of origin** (in addition to being of interest for analyzing the investment of each country in this type of technology, it could have an impact on its use in certain countries).

- **Ongoing maintenance** (evidence of maintenance in the last 3 years).
- **Multi-Guest OS** (supports more than two operating systems running simultaneously on different partitions).
- **Multicore** (runs on target hardware using more than one CPU core).

Note that this comparison will take into account the most prominent hypervisors in sections 2.1.4, 2.1.4, 2.1.4 and 2.1.4. Proprietary hypervisors that, due to their high price, are oriented towards commercial exploitation and are not interesting for future research, have their own comparison table according to other criteria in section 2.1.4.

Table 2.4: Hypervisors Comparison (Part I)

	Xen	**XtratuM***	**KVM**	**Virtuosity**	**Quest-V**
Hyp. Type	1/2[a]	1	1/2[a]	1/2[a]	2
Virtualization Type	Paravirtualization[b] Full Virtualization	Paravirtualization	Paravirtualization Full Virtualization	Paravirtualization	Full Virtualization
Supported HW Architectures	x86 ARM	ARM SPARC RISC-V	x86 ARM PowerPC	x86 ARM	x86
Supported Guest OS	Any unmodified OS that runs on the supported HW architectures	LithOS Linux RTEMS	Any unmodified OS that runs on the supported HW architectures	Linux FreeRTOS	Any unmodified OS that runs on the supported HW architectures
Inter-Partition Communication	Yes	Yes	Yes	Yes	Yes
API	Own API	APEX	Own API	APEX	POSIX
License	GNU GPL v2	GNU GPL v2 Professional version also available	GNU GPL	GNU GPL v2	GNU GPL v3
Scheduling	Borrowed Virtual Time Simple Earliest Deadline First Credit ARINC 653	ARINC 653	Completely Fair	ARINC 653	No (static partitioning)
Multi-Guest OS	Yes	Yes	Yes	Yes	Yes
Real-Time Support	No[c]	Yes	No[c]	No[c]	Yes
Developers Nationality	Worldwide	Spanish	Worldwide	American	American
Ongoing Maintenance	Yes	Yes	Yes	Yes	No
Multicore	Yes	Yes	Yes	Yes	Yes

[a]Hypervisors like Xen, KVM, Virtuosity (Xen-based) or Jailhouse cannot easily be categorised as type 1 or type 2. On the one hand, they extend the Linux kernel to make it a type 1 hypervisor, but the host OS remains fully functional and all other guest OSs run as Linux processes on this host, so in this sense the hypervisor should be considered as type 2.

[b]Xen supports full virtualization and paravirtualization for x86 architectures. For ARM architectures, it only supports paravirtualization.

[c]Hypervisors such as Xen, KVM or Virtuosity (Xen-based) support real-time guest OSs. However, it should be noted that all these guests (domUs) will always depend on the correct functioning of the host (dom0), which is based on the Linux kernel and is not formally deterministic.

Discussion

On the one hand, Tables 2.4 to 2.7 show that paravirtualization is the most popular method of virtualization (70% of the hypervisors analysed offer paravirtualization), so it seems

Table 2.5: Hypervisors Comparison (Part II)

	Rodosvisor	**RTZVisor**	**Xvisor**	**VOSYSmonitor**	**RTA-LWHVR**
Hyp. Type	1	1	1	1	2
Virtualization Type	Paravirtualization Full Virtualization	Paravirtualization	Paravirtualization Full Virtualization	Full Virtualization	Unknown
Supported HW Architectures	PowerPC	ARM	x86 ARM RISC-V	ARM	PowerPC
Supported Guest OS	Any unmodified OS that runs on the supported HW architectures	FreeRTOS	Any unmodified OS that runs on the supported HW architectures	Any unmodified OS that runs on the supported HW architectures	RTA-OS Unknown
Inter-Partition Communication	Yes	Yes	No	No	Yes
API	Own API (similar to APEX)	Own API	Own API	Own API	Own API
License	Unknown	Unknown	GNU GPL v2	Proprietary	Proprietary
Scheduling	ARINC 653	Round-Robin	Priority Round-Robin Priority Earliest Deadline First	Preemptive Priority	Reservation Based
Multi-Guest OS	Yes	Yes	Yes	No[a]	Yes
Real-Time Support	Yes	Yes	Yes	Up to 1 partition	No
Developers Nationality	Portuguese	Portuguese	Italian Indian	French	German
Ongoing Maintenance	No	No	Yes	Yes	Yes
Multicore	No	No	Yes	Yes	Yes

[a] VOSYSmonitor limits the number of real-time partitions that the system can host to one. By itself, it also guarantees to be able to run a single general-purpose partition but allows the use of another hypervisor (such as Xen, for example) to expand the number of general-purpose partitions on the system.

that efficiency is usually more highly valued than the flexibility of the virtualization solution. Along the same lines, type 1 hypervisors, which are more efficient because they have direct access to the hardware, are notably more common than type 2 hypervisors. Although it depends on the consideration given to Xen, KVM, Virtuosity or Jailhouse, which cannot be easily classified between type 1 and type 2, only 25% of hypervisors are undoubtedly type 2: Quest-V, RTA-LWHVR, NOVA, PharOS and HTTM.

On the other hand, we can see that Linux is clearly the most common guest OS supported by the hypervisors analyzed. Counting hypervisors offering full virtualization, up to 85% of hypervisors support an embedded Linux distribution as guest OS. This is not surprising in that Linux offers a robust open-source kernel, proven over many years, has a large community and provides access to a large number of software tools and development environments. As for real-time operating systems, we can find that the hypervisors analysed support some very popular ones, such as FreeRTOS, RTEMS, Zephyr or VxWorks. ARM is the hardware architecture on which most hypervisors can be deployed (70% of the hypervisors analysed have support for some ARM architecture), followed by x86 architectures (50%) and RISC-V architectures (25%) and PowerPC (clearly below with 15%). The first hypervisors to adapt to the RISC-V architecture, a modern alternative to the more classical processors that is gaining momentum in the space, aviation, and automotive sectors, are XtratuM, XVisor, Bao, PharOS and seL4.

Although in the case of open source hypervisors it is common for there to be con-

Table 2.6: Hypervisors Comparison (Part III)

	Jailhouse	**Bao**	**OKL4**	**NOVA**	**ACRN**
Hyp. Type	1[a]	1	1	2	1[a]
Virtualization Type	Paravirtualization	Paravirtualization	Paravirtualization	Full Virtualization	Paravirtualization
Supported HW Architectures	x86 ARM	ARM RISC-V	ARM x86 MIPS	x86 ARM	x86
Supported Guest OS	Linux FreeRTOS Zephyr	Linux FreeRTOS Erika RTOS	Linux Windows VxWorks	Any unmodified OS that runs on the supported HW architectures	Linux Windows VxWorks Zephyr
Inter-Partition Communication	Yes	Yes	Yes	Yes	Yes
API	Own API	Own API	POSIX	Own API	Own API
License	GNU GPL v2	GNU GPL v2	Proprietary	GNU GPL v2	BSD 3-Clause
Scheduling	No (static partitioning)	No (static partitioning)	Round-Robin Priority	Preemptive Priority	Round-Robin Priority Borrowed Virtual Time
Multi-Guest OS	Yes	Yes	Yes	Yes	Yes
Real-Time Support	Yes	Yes	Up to 1 partition	Yes	Up to 2 partitions
Developers Nationality	German	Portuguese	American	German	Chinese
Ongoing Maintenance	Yes	Yes	Yes	Yes	Yes
Multicore	Yes	Yes	Yes	Yes	Yes

[a]Although Jailhouse is a type 1 hypervisor, it requires a Linux partition that loads the hypervisor firmware. After handing over control to the hypervisor, the Linux partition must continue to run. Something similar happens with the ACRN hypervisor, which requires a pre-launched partition that has exclusive access to certain hardware elements.

tributions from different developers from all over the world (this is especially noticeable in the larger hypervisors in terms of development time and lines of code, such as Xen and KVM), it can be seen that there are certain countries that stand out above the rest in terms of the development of virtualization technologies for embedded systems: the United States, Germany and France alone account for 50% of the hypervisors analyzed. This trend is even more pronounced in the case of proprietary hypervisors with more expensive licenses: in Table 2.3, all the hypervisors analysed are American or German. In any case, these figures are not so surprising considering that these countries have a very strong embedded systems industry, especially safety-critical embedded systems, such as aviation, space, defense, or automotive. The number of Portuguese initiatives is striking, although only one of them is still actively supported (Bao). Besides them, only China is the country of origin of multiple (two) hypervisor development initiatives: ACRN and Minos.

Even with all these data, it is difficult to advise or advise against the use of one of these hypervisors over the others, because it will depend to a large extent on the particular needs of each investigation. As a general rule, hypervisors for which there is no ongoing maintenance (Quest-V, Rodosvisor and RTZVisor) should be avoided. If a low-overhead solution is required, it is better to opt for type 1 hypervisors offering paravirtualization, such as XtratuM, XVisor, OKL4, Minos or seL4. If the HW platform has sufficient re-

Table 2.7: Hypervisors Comparison (Part IV)

	Corbos	**Minos**	**PharOS**[a]	**seL4**	**HTTM**
Hyp. Type	1	1	2	1	2
Virtualization Type	Full Virtualization	Paravirtualization	Paravirtualization	Paravirtualization	Full Virtualization
Supported HW Architectures	Unknown	ARM	ARM RISC-V	ARM x86 RISC-V	MIPS
Supported Guest OS	Any unmodified OS that runs on the supported HW architectures	Linux Android Zephyr	PharOS (native) Trampoline	Linux	Any unmodified OS that runs on the supported HW architectures
Inter-Partition Communication	Yes	Yes	Yes	Yes	No
API	POSIX	Own API	OSEK	Own API	Own API
License	Proprietary	GNU GPL v2	Apache v2.0	GNU GPL v2	Unknown
Scheduling	Unknown	Unknown	ARINC 653 Preemptive Priority	Preemptive Priority	Completely Fair
Multi-Guest OS	Yes	Yes	Yes	Yes	No
Real-Time Support	Yes	Yes	Yes	Yes	No
Developers Nationality	German	Chinese	French	Worldwide	Pakistani
Ongoing Maintenance	Yes	Yes	Yes	Yes	Yes
Multicore	Yes	Yes	Yes	Yes	Yes

[a]PharOS is originally an open source RTOS. However, Lemerre et al. added a virtualization layer to the RTOS in order to deploy VMs using another RTOS (Trampoline) [115], which technically makes PharOS a type 2 hypervisor.

sources, especially in terms of processing cores, it might be interesting to opt for simpler solutions offering static partitioning to obtain even lower overhead, such as Bao or Jailhouse. Even so, in the case of Jailhouse, as in the case of Xen, KVM or Virtuosity, it must be taken into account that their use is associated with the mandatory deployment of at least one Linux partition, so they might not be interesting in case the system to be developed has real time requirements. These hypervisors, as well as type 2 hypervisors (RTA-LWHVR, NOVA, PharOS or HTTM) and type 1 hypervisors offering full virtualization (Corbos and VOSYSmonitor) can be interesting in case system resources are not a very limiting factor, and a flexible solution is sought.

Finally, as regards their application in safety-critical systems, XtratuM, which has been applied in several aerospace projects and research works and is in the process of certification in some of them, as well as static partitioning solutions (Bao and Jailhouse) for simpler systems, would be particularly recommendable. In all these cases there is support for real-time operating systems, although XtratuM would be the most recommendable due to its API, which is much more similar to the APEX required by the ARINC 653 standard.

2.2. Safety-Critical Embedded Systems Standards

The present thesis is submitted in partial fulfilment of the requirements for the degree of an Industrial Ph.D. According to art.15 bis of RD 99/2011 of January 28, the designation *"Doctorado Industrial"* (Industrial Ph.D.) applies to those programmes that are totally or partially carried out within a company and aims to promote research in companies in technological sectors. Given this special characteristic, we consider it interesting to dedicate a few pages to present some of the standards that apply to software development in the space, aviation, and automotive sectors. Note that these are some of the standards that the hypervisor and the rest of the software developed in this, or other research works would have to comply with, if it were to be put into practice in a real commercial system.

2.2.1. Aviation Software Standards

FACE Technical Standard

The Future Airborne Capability Environment (FACE) Technical Standard defines an open reference architecture aimed at increasing the reusability and portability of software components and creating software product lines across the military aviation domain [122]. The FACE approach enhances portability from one operating environment to another through the application of modular architecture, hardware abstraction, standardized interfaces, and data models. In the long term, the objective of FACE is to reduce development costs, integration costs and time-to-field for avionics capabilities.

FACE is not the only standard oriented to develop an Open System Architecture (OSA). Tokar [123] analyzes, together with FACE, two other initiatives that have the same objective: Unmanned Aerospace Systems Command and Control Standard Initiative (UCI) and the Open Missions Systems (OMS) initiative. While the latter are more focused on UAVs, Tokar concludes in his article that FACE is a standard with a broader and more complete scope, and that it has a great momentum in the current military aviation industry, especially in the American Army. The National Defense Authorization Act for Fiscal Year 2017 requires all defense acquisitions to use the Modular Open-System Approach (MOSA) to the maximum extent possible as of 2019. MOSA is not a standard, technically speaking, but a business and technical strategy. FACE could be considered a concrete example of a standard that fits within MOSA [124].

DO-178C

DO-178C, Software Considerations in Airborne Systems and Equipment Certification (and its European equivalent, ED-12), is the certification standard and main document by which certification agencies, such as the FAA or EASA, approve flight software. The current edition was developed in 2011 and approved in 2012, and is the successor to DO-

178B, which dates back to 1992 [125]. Although there is no publicly experimental data to demonstrate its effectiveness, it is clear that the standard has been a success: since its inception, there has not been a single fatal accident on a commercial aircraft attributed to a software error [126].

The objective of DO-178C is to provide guidance for the development of flight systems software, so as to ensure that it performs its intended function with an appropriate level of confidence. DO-178C is goal-based and companies can use a variety of means to achieve compliance as long as they meet the goals in question. To demonstrate compliance with the standard, companies must provide various documents (the nature and number of these documents varies, depending on the criticality of the software in question) related to their development processes. To establish the criticality of the software, the Design Assurance Level (DAL) defined in ARP4754 is used, in which each level corresponds to the different effects that a failure condition in the system can cause [127]. Specifically, all software whose anomalous behavior, as shown by the system safety assessment process, would cause or contribute to a failure of system functions...

- ...resulting in a catastrophic failure condition for the aircraft, will be categorized as **DAL A**.
- ...resulting in a hazardous/sever-major failure condition for the aircraft, will be categorized as **DAL B**.
- ...resulting in a major failure condition for the aircraft, will be categorized as **DAL C**.
- ...resulting in a minor failure condition for the aircraft, will be categorized as **DAL D**.
- ...with no effect on aircraft operational capability or pilot workload, will be categorized as **DAL E**.

In turn, failure conditions are categorized according to their consequences [128]:

- Minor failure conditions would not significantly reduce airplane safety, and would involve crew actions that are well within their capability.
- Major failure conditions would reduce the capability of the airplane or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example, a significant reduction in safety margins or functional capabilities, a significant increase in crew workload or in conditions impairing crew efficiency, or discomfort to occupants, possibly including injuries.
- Hazardous failure conditions would reduce the capability of the airplane or the ability of the crew to cope with adverse operating conditions to the extent that there

would be (a) a large reduction in safety margins or functional capabilities; (b) physical distress or higher workload such that the flight crew cannot be relied on to perform their tasks accurately or completely; or (c) serious or fatal injury to a relatively small number of the occupants.

- Catastrophic failure conditions would result in multiple fatalities, usually with the loss of the airplane.

As can be deduced, a hypervisor or an operating system that provides time and space partitioning should be certified to the same level of criticality as the most critical of the functionalities that runs on top of it. The great advantage offered by the use of these elements is that software of different DALs could coexist on the same platform, since the hypervisor/RTOS guarantees that the failure of one partition would not affect the rest of the partitions of the system.

2.2.2. Space Software Standards

There are several standards that categorize software criticality in the space industry. The two most commonly used are ECSS-Q-ST-80C, defined by the European Space Agency and commonly used in European projects, and NASA-GB-8719.13, defined by NASA and commonly used in the USA. It should be noted that, as with standards in other similar disciplines, the allocation to different levels of criticality is done by means of a safety analysis at the system level, not at the software level. Thus, the criticality level is assigned to each high-level function, and the software inherits the criticality level corresponding to the function category that it implements [129].

ECSS-Q-ST-80C

According to the ECSS-Q-ST-80C, all software that if not executed, or if not correctly executed, or whose anomalous behaviour could cause or contribute to a system failure resulting in:

- ...catastrophic consequences, will be categorized as **Category A** software.
- ...critical consequences, will be categorized as **Category B** software.
- ...major consequences, will be categorized as **Category C** software.
- ...minor consequences, will be categorized as **Category D** software.

According to the standard, the consequences are:

- Catastrophic, if there is loss of life or severe detrimental environmental effects.

- Critical, if there is loss of mission, temporarily disabling (but not life-threatening) injuries or major detrimental environmental effects.
- Major, if there is major mission degradation.
- Minor, if there is minor mission degradation or any other effect.

It is evident that there are many equivalences between the criticality levels defined by this standard and the DALs defined in ARP4753 for aviation systems. The main difference is that ECSS-Q-ST-80C does not contemplate a criticality level in which a failure has no effect that affects safety (DAL E), and that the consequences contemplated according to the space standard take more into account the achievement of the mission objective, while in aviation (especially in commercial aviation) there are more human lives at stake, so this aspect has more weight when assessing the consequences.

2.2.3. Automotive Software Standards

The two most popular standards for Automotive software are Automotive Open System Architecture (AUTOSAR) and ISO 26262 (titled "Road vehicles - Functional Safety"). They are briefly presented below.

AUTOSAR

Automotive Open System Architecture (AUTOSAR) is a worldwide partnership founded in 2003, aimed at establishing an open and standardized software architecture for automotive ECUs. AUTOSAR defines a modular architecture that includes a series of basic software modules, which can be used in vehicles from different manufacturers and electronic components from different suppliers, reducing development costs and containing the increasing complexity of electronic and software architectures for automotive. In addition, it defines the interfaces to develop applications and builds a common development methodology based on the standardized exchange. In this way, AUTOSAR encourages scalability, portability, collaboration between partners and the sustainable use of natural resources, without compromising product quality.

Today, we can distinguish between the AUTOSAR Classic Platform and the AUTOSAR Adaptive Platform. The AUTOSAR Classic Platform is the standard for OSEK-based real-time ECUs, and is divided into three main layers [130]:

- Basic Software Layer: standardized software modules that do not fulfill any specific automotive function, but are necessary for the correct functioning of the upper software layers.

- Runtime Environment (RTE): software layer that abstracts the applications from the exchange of information between them and the Basic Software and between the applications themselves. This layer represents the interface against which applications are developed.
- Application Layer: applications that interact with the RTE.

Demanding new use cases, such as highly automated driving, present a series of needs such as frequent updates, great processing capacity or communication with the changing environment, which cannot be covered with the AUTOSAR Classic Platform. For this reason, the AUTOSAR Adaptive Platform was released in 2017, a new service-oriented architecture that can meet the demands of the next generation of ECUs. One of the main functional differences with respect to the Classic Platform is that the core of the Adaptive Platform is an OS based on a POSIX subset, which opens up a range of possibilities (task creation, memory allocation, use of signals, timers ...) to the application developer [131]. Other notable differences are that the ECUs of these systems base their communication on Ethernet and have a dynamic architecture, in which applications can be updated over-the-air during the life cycle of the vehicle.

Some of the requirements that AUTOSAR imposes are covered by the inherent characteristics of hypervisors [132]. In an article proposing a Linux-based AUTOSAR platform, Kotur et al. explain that the main line of future research for their work would be the addition of a hypervisor [133]. According to them, a hypervisor would help solve several safety-related problems: for example, it helps a more efficient management of hardware resources, so it could help avoid network congestion, which can eventually lead to latencies and errors in the system. Of course, a hypervisor would also provide the necessary temporal and spatial separation so that, in the event of an application failure, this failure would not affect the rest of the system, which is a very important vulnerability if there is no hypervisor or an OS that provides these TSP features.

ISO 26262

ISO 26262 (titled "Road vehicles - Functional Safety") is, since 2011, the functional safety standard for electric and electronic systems installed in serial production road vehicles [134]. The standard defines "functional safety" as the "absence of unreasonable risk due to hazards caused by malfunctioning behavior of electrical/electronic systems" [135].

Similar to other critical real-time systems standards, such as DO178-C in aviation systems or ECSS-Q-ST-80C in space systems, ISO26262 uses a tiered risk categorization: Automotive Safety Integrity Level (ASIL). ASILs establish the safety requirements for automotive components in accordance with the ISO 26262 standard, based on the probability and acceptability of the automotive hazard. There are four ASILs: ASIL A, ASIL B, ASIL C and ASIL D, with ASIL D being the one with the most stringent integrity

requirements (highest risk) and ASIL A the least stringent (least risk). Examples of elements of a car that usually require ASIL-D grade would be airbags or electric power steering system. For the power windows, ASIL-A grade is sufficient [136]. The headlights could have an ASIL-B grade and, for example, the active suspension of the vehicle is considered in many cases to be ASIL-C grade. Note that the maximum level of risk, according to the standard, is not equivalent with respect to those defined in its aviation and space counterparts. While ASIL D refers to, at most, a loaded passenger van, the potential danger from an aircraft loaded with passengers and fuel is greater.

2.3. Software-Based Fault Tolerance

This thesis focuses on using redundancy to propose a fault-tolerant architecture in a safety-critical embedded system. Therefore, it is important to know some concepts related to software-based redundancy techniques, since these concepts are mentioned both in the works related to this research, in chapter 3, and in the solution itself proposed in this research, in chapters 4 and 5. This section provides a brief description of the essential concepts:

Triple Modular Redundancy (TMR) is a common scheme to protect computer systems against failures. In its original version, proposed by Von Neumann in 1956 [137], three modules or black boxes (they can be simple units or complex computers) calculate a single output. Another element, called *majority organ* by Von Neumann, accepts the inputs from these modules and provides the majority opinion as the final output.

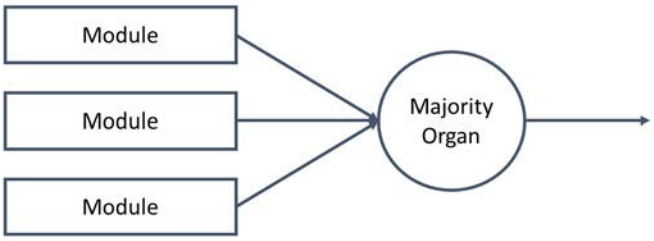

Figure 2.8: Triple Modular Redundancy

Traditionally, in sectors such as space or aviation, TMR has been applied to hardware elements to protect a system against Single Event Upsets (SEUs [138]) caused by radiation. As electronic components (processors, memory, etc.) have become more powerful, the application of TMR to software has become progressively more feasible and popular.

There are many variations and extensions to the original TMR concept in order to make it more robust. For example, one of the most common is to extend the number of modules used to redundant a single task to more than three (**N-Modular Redundancy** [139]). On other occasions, in order to solve the problem of the majority organ

(more commonly called *voter*, nowadays) being a single point of failure, the voter itself is also redundant [140], as shown in Figure 2.9.

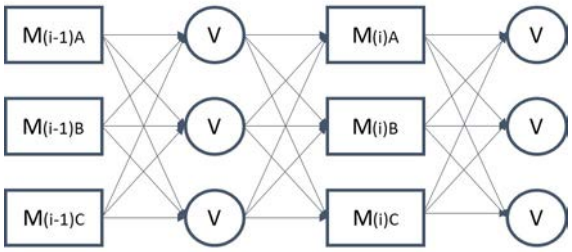

Figure 2.9: TMR with redundant voter

Another variation, first presented by Czajkowski et al. and especially interesting for systems with low computational resources, is **Temporal Triple Modular Redundancy (TTMR)** [141]. The main difference between TTMR and TMR is based on the fact that a redundant task with TTMR is executed only twice in the absence of error. As long as there is no disagreement between the two tasks, it will continue to be executed twice: only in case of disagreement will the task be executed a third time, to break the tie.

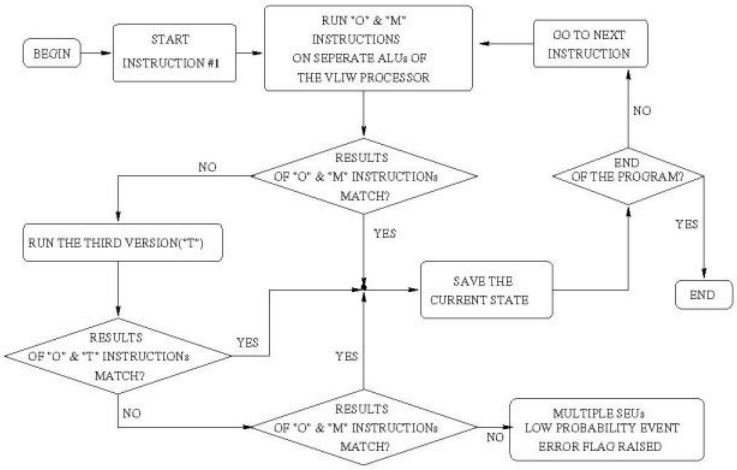

Figure 2.10: Flowchart of the TTMR algorithm [141]

2.4. Hypervisor Fault Tolerance

This thesis focuses on proposing a fault-tolerant architecture, through redundancy of critical tasks by means of virtualization, for a real-time embedded platform (such as those found in the space, aviation and automotive sectors). Therefore, the hypervisor is the main enabler tool and is assumed to be a secure and reliable element. However, the reality is that virtualization infrastructure, including the hypervisor, are not exempt from hardware errors, which can affect different virtual machines and can even lead to a total

system failure [142]. In order to provide as complete a view of the state of the art as possible, as well as to outline possible lines of future work, this section collects some of the main studies focused on recovery techniques and protection of the hypervisor itself. Note that these studies are **complementary** to the work done in this thesis.

Many of the hypervisor recovery techniques are based on the microreboot concept, which was introduced by Candea et al. in 2004 [143]. The main idea behind the microreboot is to avoid the expensive cost of completely rebooting a system after a failure, by designing a much more fine-grained and selective mechanism, which allows recovering the failed elements without harming the rest of the system elements, which continue to run normally. Since then, there are works that have applied the use of microreboot to recover the hypervisor after a failure, such as RootHammer or ReHype, both based on Xen. RootHammer follows a simpler approach, as it does not act on hypervisor failures, but is used to periodically reboot (*rejuvenate*) the system, trying to avoid the *software aging* of the hypervisor [144]. To do this, it reboots both Xen and dom0 (see section 2.1.4 for more details about Xen) and saves the state of the other virtual machines, so that execution can resume correctly after a few seconds. ReHype [145] is a mechanism that allows the Xen hypervisor to recover from a failure, while preserving the state of the virtual machines. ReHype allows the hypervisor to recover from 90% of the failures, and has a relatively small footprint on Xen (approximately 900 LOC and 2MB of memory). Unlike RootHammer, ReHype does not require a reboot of dom0, so the down time involved in the rebooting is considerably less. Le and Tamir recently extended this work by developing recovery mechanisms for driver virtual machines and privileged virtual machines [146]. Together with ReHype, the result is a robust virtualization infrastructure based on Xen. The lines of research they leave open are related to silent failures that ReHype is not able to detect, which are estimated to be 10-36% of the total failures.

Zhou and Tamir discuss how even microreboot involves unacceptably high latency in some complex systems, and propose an alternative they call *microreset*, in which the failing component is reset to a quiescent state that is highly likely to be valid [147]. To demonstrate this, they develop NiLiHype, a mechanism that uses the microreset to implement recovery from a Xen hypervisor failure. In the tests performed, NiLiHype is able to achieve almost the same failure recovery rate, but with up to 30 times lower latency.

Tan et al. propose TinyChecker [148], a nested virtualization-based alternative to harden the protection of a hypervisor. TinyChecker is a very lightweight hypervisor that runs between the hardware and the hypervisor to be enforced, and monitors the interactions between them, so that it can detect potential errors and recover from them by creating checkpoints. However, TinyChecker has not been fully implemented, and it is expected that a fully functional version would impose a significant overhead on the system. Furthermore, while its simplicity makes it less prone to failure than a conventional hypervisor, it is still a single point of failure in the face of, for example, hardware faults.

A team from George Washington University has published a couple of papers that

also address the problem of the hypervisor being the single point of failure in a virtualized system. Xentry [149] is a framework that enables early soft error detection to prevent the propagation of errors from the hypervisor to the virtual machines. DualVisor [150] is a tool that replicates both the execution and the data of the hypervisor, to protect it from possible hardware errors. The article presents a partial prototype on which the tool's overhead is measured and presents future lines of work, which include not only completing the prototype, but also studying the protection of the framework that implements DualVisor, which is itself vulnerable to failures. Cerveira et al. follow a similar reasoning and analyse how, despite what is often assumed, a significant percentage of virtual machines running on a hypervisor remain fault-free even when the hypervisor fails, so they could continue to run if migrated to a new hypervisor [151]. Both papers are intended for application in data centres and cloud computing infrastructures and would imply a very high overhead for embedded systems, but may be of interest for specific (computationally simple) cases or as SoCs continue to increase their computational capabilities. For the same reason, although perhaps not directly applicable to embedded systems at present, it is interesting to consider work such as that of Sousa et al., in which researchers define requirements regarding the number of replicas needed to maintain availability given maximum numbers of simultaneously faulty and recovering replicas [152]. There are also studies aimed at reducing the overhead involved in Hypervisor-Based Fault Tolerance during a failure-free state, such as the one by Zhu et al [153]. Some of the techniques studied are applicable exclusively to Xen, which is the hypervisor used in the study, and others can be extrapolated to a general use case. Regarding the migration of virtual machines to a new hypervisor, Reiser and Kapitza have a paper in which they investigate how to alleviate the problem of unavailability during this migration, and show that the reboot time can be reduced if the initialisation of the new hypervisor is done in parallel to the normal execution of the old one [154].

2.5. Single Event Upset

Single Event Upset (SEU) is a term that encompasses a number of phenomena associated with the interaction of energetic particles (X-rays, gamma rays, cosmic ray neutrons and musons, alpha particles, energetic ions...) with the silicon substrate of different electronic components. The generation of charges in transistors can cause changes in internal voltages, leading to corruption of stored or transmitted data. The most typical consequences of SEUs are temporary loss of data in memories and flip-flops or transistor latchups. In a component that is performing calculations (a processor, for example) or in a memory that is storing data with which calculations are being performed, these consequences usually result in a calculation error. These errors are usually temporary, as SEUs do not normally cause permanent damage to electronic components [155].

SEUs have been detected since the 1950s, when nuclear experiments were being conducted and anomalies were observed in the electronic test monitoring equipment. In 1972,

Hughes satellite experienced problems in its connection with ground operators for more than a minute, and researchers Edward C. Smith, Al Holman, and Dan Binder first associated these problems with cosmic radiation and exposed this phenomenon in a paper in 1975 [156]. The term Single Event Upset was first used a few years later, in 1979, in a paper by Guenzer et al [157]. Since then, SEU detection and protection has been a recurring theme in various industries, including the aviation and space industries. Over the years, the components traditionally used in electronic systems associated with aircraft have been designed to be less vulnerable to this type of error, but they continue to have parts that are more sensitive than others. For example, first- and second-level caches or the state machine that controls microprocessors are usually small and designed to have a very high speed, so they do not have much charge. RAMs are another type of component that is particularly vulnerable to ionizing particles. These problems are accentuated by the trend in recent years to use COTS components to design flight systems faster and cheaper. This is where error detection and correction mechanisms such as the one proposed in this research come into play.

2.6. Summary

This chapter has explained some concepts that are indispensable to fully understand the content of the rest of the thesis. It is worth mentioning that the chapter contains an original contribution: an exhaustive survey that reviews the state of the art in safety-critical embedded hypervisors, categorises them and compares them by means of different parameters. The survey will be used to choose a hypervisor with which to prototype the solution proposed in this thesis, and the aim is that it will help other researchers to make this kind of decisions in the future.

After understanding the concepts explained in this chapter, chapter 3 will deal with the current state of the art in Hypervisor-Based Fault Tolerance techniques, presenting the problem that gave rise to these techniques and reviewing the main works that develop them. These works will be divided into two categories: those applied to safety-critical embedded systems, more closely related to this thesis, and those applied in other contexts, such as the implementation of large cloud servers, from which some contributions can be extrapolated to embedded systems.

3. PROBLEM STATEMENT AND RELATED WORK

This chapter briefly discusses the reasons why the use of virtualization techniques to implement software-based redundancy has been studied in recent years. The chapter is divided into three main sections:

- Section 3.1 details the problem that gave birth to redundancy techniques based on the use of virtualization.
- Section 3.2 reviews the work that has been done in this area, both in relation to embedded safety-critical systems and in relation to cloud servers and cluster computing. Note that, although the latter scenario is quite different from the scenario proposed for this thesis, some ideas can be extracted from the existing work that are applicable to embedded systems or, at least, allow outlining lines of future work.
- Section 3.3 briefly summarises the chapter and introduces the contents of the following chapter.

3.1. Problem Statement

Traditionally, to protect systems from soft errors, industries such as aerospace have used hardware redundancy techniques [158] or specifically hardened electronic components [159]. However, not only are these solutions too expensive for the commodity products being developed in other industries (such as the automotive industry), but even the aerospace industry is being forced to migrate to the use of COTS components, which are the only option to meet the computational demands of today's applications and, moreover, imply a considerable cost reduction. While this change in trend is positive in many respects, it implies the considerable challenge of protecting such components from failure through alternative techniques, such as software-based redundancy.

However, hosting a task plus its redundant counterparts on the same processor presents an obvious vulnerability: a hardware failure can affect all tasks at the same time. Not only that, but the corruption of one of the tasks could affect the proper execution of the others. The solution to this could be to separate the redundant tasks on different hardware, but this means that the resulting redundancy-based fault-tolerance mechanism can become too costly, both economically and in terms of weight, power consumption and efficiency. Virtualization provides an alternative solution, since a sufficiently robust hypervisor is able to guarantee the temporal and spatial separation of different tasks running on the same device, enabling the possibility of redundant tasks without having to duplicate hardware resources. In fact, a recent study by Frigerio et al. proves that virtualization is a very substantial alternative to physical separation. The study describes a methodology to

analyse a safety-critical (automotive) system from a system-level point of view through various parameters, such as cost, application failure probability, system load and cable length used [160]. This methodology is used to compare solutions where system elements are redundant through physical separation with solutions where a hypervisor is used to address redundancy on the same hardware platform. The results not only show that virtualization is a valid and comparable alternative to physical separation in terms of performance, but also that it offers significant advantages in some situations.

Throughout this chapter we will review the work related to what are called Hypervisor-Based Fault-Tolerance (HBFT) techniques, especially those applied to safety-critical embedded systems. As will be seen, **each of these works has limitations that make it difficult to apply them to actual commercial use cases. These limitations and the proposed solution to overcome them will be presented in chapter 4.**

3.2. Hypervisor-Based Fault Tolerance (HBFT)

The term *Hypervisor-Based Fault Tolerance* was first used in a paper by Bressoud and Schneider in December 1995 [161]. In it, the authors first put forward the idea of using a hypervisor to run both a primary virtual machine and its backup, so as to obtain a fault-tolerant system without having to modify the hardware, operating system or applications. They also developed a prototype for HP's PA-RISC instruction-set architecture, using two processors connected by SCSI bus and Ethernet, on which they measured the overhead of their approach to implementing fault-tolerance, and demonstrated that it was an interesting option for future research. The HBFT concept has been developed in many other papers since then and, of course, advances in recent decades (both in the software of the hypervisors themselves and, more importantly, in the computational capabilities of computers) have allowed both the approaches and the prototypes used to demonstrate them to become more sophisticated. This section reviews some of the main studies on HBFT, dividing them into two categories:

- One focused on the application of HBFT to safety-critical embedded systems. Note that this is the category in which this thesis would be classified, and it is in this specific field that we seek to extend knowledge and propose a novel solution.
- One oriented towards conventional computers and large server clusters. Most of the HBFT work falls into this category. Although not directly related to the thesis, some of the ideas developed in these papers could be extrapolated to embedded systems, so the most relevant articles will be briefly summarised.

3.2.1. HBFT for Safety-Critical Embedded Systems

Campagna et al. were among the first to apply virtualization to implement a fault-tolerant reference architecture in safety-critical embedded systems [84]. As an alternative to tradi-

tional radiation hardening, which requires redesigning the processor and other electronic components or using silicon technology specifically designed to withstand the effects of radiation, they propose to protect the system through software replication on the same COTS target, using a hypervisor to separate the redundant units. The objective of this architecture is to be able to detect Single Event Upsets (SEUs) that can be caused by ionising radiation impacting the processor. The targeted use case is payload processing in space computers.

The architecture proposed by Campagna et al. is roughly divided into three layers:

- The Hardware Layer, which comprises the typical elements of a SoC (processor, memory, I/O devices...).
- The Hypervisor Layer, which runs in the highest privilege mode on the processor, and through which all hardware resources are managed and virtual machines are scheduled, guaranteeing total independence (temporal and spatial) between them.
- The Application Layer, which includes both the application software and the error detection mechanisms.

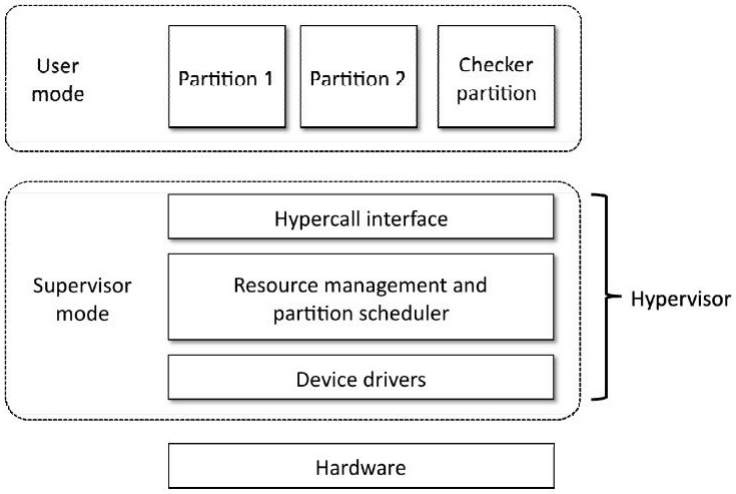

Figure 3.1: HBFT Architecture Proposed by Campagna et al. [84]

As can be seen in Figure 3.1, the error detection mechanism follows a task duplication scheme: two virtual machines run an identical copy of the application software, and a third virtual machine acts as a voter, checking the results and approving them as correct, if they are the same, or sending an error status to the platform computer if they are not. With the prototype they implement (using a COTS based on the LEON3 processor and XtratuM as a hypervisor), they calculate that the overhead of the architecture is 108% (closest to the minimum possible, taking into account that the architecture implies duplication of tasks) and that it is potentially capable of detecting 96% of the SEU that can affect the system

(without needing the timeout of a watchdog timer, which would allow detecting the rest of the failures).

In a similar vein, Sabogal and George have an article in which they present Virtualized Space Applications (ViSA), a framework that extends the Xen hypervisor to achieve HBFT in flight systems [59]. The framework thus consists of three main layers: the Xen hypervisor, the ViSA middleware and the mission-specific flight software, which is based on NASA's core Flight System (cFS).

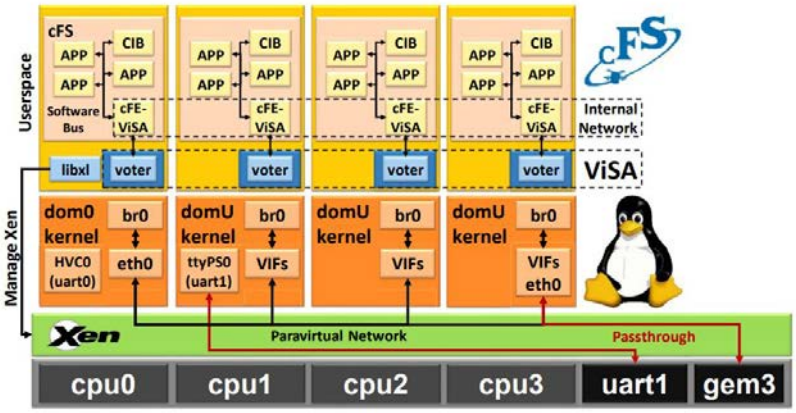

Figure 3.2: HBFT Architecture Proposed by Sabogal and George [59]

As in any system using Xen, the first domain to be created is the Linux-based dom0 kernel, which in turn launches the ViSA service as a single process. From this domain the rest of the domUs will be launched, and each domU runs an instance of ViSA middleware and flight software. ViSA implements N-Modular redundancy between the domains, and allows voting to take place at two different layers: at the ViSA level or at the cFS level.

Missimer et al. present a real-time system called Quest-V, which uses hardware virtualization extensions to divide machine resources into different partitions, which they call sandboxes [89]. For the most part, Quest-V works like a statically partitioned hypervisor (i.e., it allocates resources to partitions statically, so that the hypervisor does not need to schedule the different sandboxes over time). There are exceptions to this approach, and one of them is precisely when a fault recovery procedure needs to be triggered: Quest-V implements a N-Modular redundancy voting algorithm in which one of the sandboxes, acting as a referee, captures the states of all sandboxes running a redundant algorithm, and reaches a consensus on the outcome. Sandboxes that do not match the consensus are assumed to be malfunctioning, and are restored to the state of all other functioning sandboxes.

Quest-V allows to select between three different configurations, depending on where the voting mechanism resides:

- **The voting resides on the hypervisor**. The main advantage of this configuration is that there is no need to modify the sandbox operating system. On the other hand,

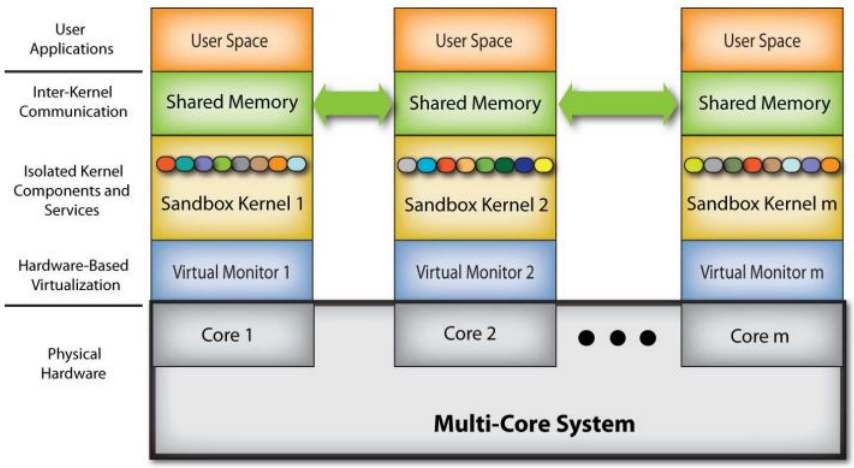

Figure 3.3: Quest-V Architecture [89]

the voting mechanism is a single point of failure and makes it necessary to duplicate the entire guest (sandbox).

- **The voting resides in a sandbox**. This configuration implies a simpler design and redundancy is done at the application level, so it is not necessary to redundant the entire guest (sandbox). The main disadvantage is that it involves modifying the guest and that an additional sandbox is required just for the voter. This involves an especially large overhead considering that Quest-V performs static partitioning, so resources (such as CPU core) allocated to the voter cannot be used by the other sandboxes.

- **The vote is distributed across different sandboxes**. This configuration causes the modification of the guest operating systems, and also implies extra complexity, as there has to be a physical device shared between different sandboxes. On the other hand, it has some of the advantages of the other two configurations: no extra sandbox is required for the voter, and redundancy is done at the application level.

Esposito et al. also use TMR (as well as TTMR) to protect multicore systems against Single Event Upsets (SEUs) caused by radiation [162]. As in other similar solutions, the architecture they propose can be roughly divided into three layers:

- The Hardware Layer includes, in addition to the resources of the COTS SoC on which the system is based, three IP blocks running on the FPGA: a Watchdog Processor to implement a signatures-based Control Flow Check, a majority voter that communicates to the user the correct result after applying the TMR technique and a System Watchdog Timer that allows to restart the system in case of error.

- The Platform Software Layer has a type 1 hypervisor as its main element. The hypervisor provides the time and space partitioning necessary to isolate each partition

from the others and is in charge of providing communication mechanisms between partitions, so that they can exchange data securely.

- The Application Software Layer implements the fault tolerance mechanisms using the two previous layers.

To choose the appropriate fault tolerance strategy, between TMR and TTMR, they divide tasks between those with hard real-time requirements (failure to meet the task deadline causes catastrophic consequences), to which they assign a TMR strategy, and tasks with soft real-time or firm real-time requirements (failure to meet the task deadline causes service degradation to some degree), to which they assign TTMR. As can be seen in Section 2.3, the main difference between TMR and TTMR is that, in TTMR, the redundant task is executed a third time to break the tie only if the first two executions do not yield the same result, while in TMR it is always executed three times and the winning result is voted by majority. TTMR is more efficient, since the task is executed only twice if there is no error, but TMR has the advantage of being time deterministic.

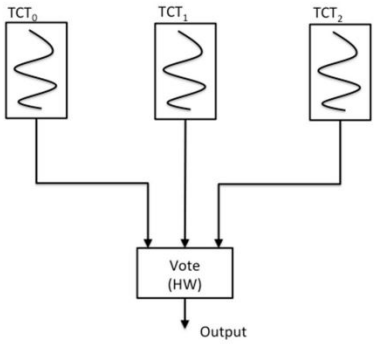

Figure 3.4: TMR Strategy Workflow (figure from [162])

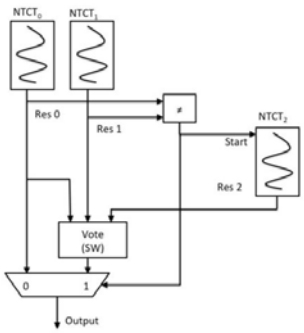

Figure 3.5: TTMR Strategy Workflow (figure from [162])

To test the architecture, they implement a prototype with two tasks: a hard real-time task of Control and a soft real-time task of Vision. For the vision task, which follows a TTMR scheme, the third execution is carried out (if necessary) in the iteration after the error detection (for the iteration in which the error is detected, data from the previous iteration is used). The logic behind this is that such algorithms often rely on the history of the results, so it is less harmful to use data from previous iterations than to run the risk of introducing completely erroneous data, which will propagate over time.

Using this architecture and a fault injection simulation campaign, they found that the vast majority of SEU faults are masked or silent by the mechanism. Of the faults detected, only <0.3% and <0.6% respectively on CPU registers and Configuration registers put the system in a state from which it could only be recovered through a reset.

3.2.2. HBFT for Cloud Servers and Cluster Computing

Prior to its application to embedded safety-critical systems, there are works that apply HBFT to general-purpose computers. In recent years, moreover, with the rise of Cloud Servers and Cluster Computing, HBFT has gained momentum as an important mechanism to harden systems against failures through software replication. Although the vast differences between embedded processors and the computing systems that can be found in a modern Cloud Server mean that the studies are not comparable, there are some ideas that have fed into the studies presented in section 3.2.1, and some others that can be applied in the future. Because of this, this section brings together some of the most relevant studies with respect to HBFT techniques on non-embedded computers and systems.

The work by Reisser et al. is probably the first to consider virtualization for redundant applications on a single host [163]. The architecture proposed by the researchers is called RESH (Redundant Execution on Single Host), and the prototype developed runs on the Xen hypervisor. The fault tolerance mechanism is based on an N-Modular Redundancy scheme in which voting takes place on dom0. In turn, to mitigate the typical Xen vulnerabilities (complexity and dependency on dom0), they divide dom0 into two parts: the DomainNV, which provides low-level network drivers, a network protocol stack and the voting mechanisms; and the rest of the Domain 0, which provides the rest of the privileged tasks (virtual drivers, monitoring, management of the guest virtual machines...). Subsequently, some of the RESH authors extended the work to offer replication not only on a single host, but also on different heterogeneous nodes hosted on different hosts and communicated through asynchronous communication networks, to reinforce the system against potential hardware failures [164]. This new solution, called VM-FIT, complements active replication with proactive recovery mechanisms. Both RESH and VM-FIT are based on Xen, but similar solutions exist on other hypervisors, such as KVM (Jeffery et al. [165]) or VmWare (Scales et al. [166]).

There are a couple of papers that also propose a virtualization and fault tolerance technique to improve the response time and availability of a cloud computing system, adding an interesting new feature to make the algorithm adaptive. Malik and Huet's work [167] proposes a model called Adaptive Fault Tolerance in Real-time Cloud computing (AFTRC), which measures and updates the reliability of each of the nodes in the system using an algorithm called Reliability Assessor (RA). In their Virtualization and Fault Tolerance (VFT) model, Das and Khilar [168] propose a similar actor that they call Decision Maker (DM), which tracks the Success Rate (SR) of each of the nodes and the Performance Record (PR) of the physical servers on which these nodes are hosted. An algorithm called Decision Maker (DM) checks that each node has completed its assigned task correctly and on time, and increases or decreases its SR accordingly. If a node has failed, it is reported to another actor, the Fault Handler (FH), which is in charge of trying to recover the node to make it available again to the system. Although this type of scheme is not directly applicable to an embedded system, as it has far fewer hardware resources, the idea

of tracking the health of virtual machines (both primary and redundant) and trying to recover them after a failure is transferable and novel when applied to an embedded system, although it is necessarily applied in a simpler way. There are also several other works related to improving the availability of cloud data centres through redundancy at virtual machine level in different host servers following complex algorithms [169] [170] [171]. However, unlike those mentioned above, these do not have features that can be extrapolated to safety-critical embedded systems.

3.3. Summary

This chapter has analysed the state of the art in Hypervisor-Based Fault Tolerance techniques, the main topic of this thesis. First, the motivation for the birth of these techniques has been explained. Then, the main related works in safety-critical embedded systems have been detailed, exposing their main advantages and disadvantages. Finally, some works applying HBFT techniques in other contexts have been analysed, from which some features can be extracted to propose a novel solution in embedded systems, such as health and reliability monitoring of each partition of the system.

After reviewing the state of the art, Chapter 4 discusses the methodology we propose to implement a comprehensive fault tolerance solution, which takes into account and addresses the main weaknesses of the works we have analysed in this chapter.

4. PROPOSED SOLUTION

This chapter presents our proposed solution to the problem posed in Chapter 3. The chapter is divided into five main sections:

- Section 4.1 summarises the characteristics of our related work, which have been described in detail in Chapter 3. From this related work we extract the requirements that our solution must meet in order to be more complete and more applicable in a real use case.
- Section 4.2 describes the overall architecture of the proposed solution, its layers and elements.
- Section 4.3 describes in detail the voter used in the solution, including its internal structure and modes of operation. In addition, the possibility of redundancy of the voter so that it does not involve a single point of failure is discussed.
- Section 4.4 describes the Health Monitoring partition, the other key element of the solution, and the processes it carries out to ensure the correct functioning of the system.
- Section 4.5 briefly summarises the chapter and introduces the contents of the following chapter.

4.1. Requirements

As discussed in Chapter 3, there are several works related to the application of virtualization techniques to implement fault tolerance mechanisms. However, only a few of these works are oriented towards their application in safety-critical embedded systems that can be found in industries such as space, aviation or automotive. Table 4.1 shows, as a summary and to the best of our judgement, the strengths and weaknesses of the proposals presented in each of these papers.

The aim of this thesis is to propose an HBFT solution that can potentially improve the performance of previous work. To do this, we first use the information gathered in table 4.1 to establish some of the basic requirements that our solution should offer. The most obvious conclusions are:

- The solution must use, at a minimum, TMR to redundant a task. This will allow both error detection and identification of the malfunctioning partition.
- The voting mechanism should not involve a single point of failure.

Table 4.1: Advantages and disadvantages of the related work

	Advantages	**Disadvantages**
Campagna et al. [84]	- Hypervisor-independent architecture (although it uses the inter-partition communication mechanisms of XtratuM, so it needs a similar mechanism in case of migration to another hypervisor).	- Errors can be detected, but it is not possible to identify which partition is failing. - The voting partition is a single point of failure.
Sabogal and George [59]	- N-Modular Redundancy: not only the error is detected, but the malfunctioning partition can be identified. - The voting mechanism is distributed, so it is not a single point of failure.	- Dependent on a specific hypervisor: Xen. In addition, Xen is based on the Linux kernel and therefore presents significant difficulties in dealing with certification processes for real-time safety-critical systems.
Missimer et al. [89]	- N-Modular Redundancy: not only the error is detected, but the malfunctioning partition can be identified. - Possibility of distributing the voting mechanism to prevent it from becoming a single point of failure.	- Dependent on a specific hypervisor (Quest-V) and therefore also on specific hardware. - Static partitioning also limits the target to be used, since there can be a maximum of as many partitions as processing cores in the target (important limitation for monocore/dual-core processors).
Esposito et al. [162]	- N-Modular Redundancy: not only the error is detected, but the malfunctioning partition can be identified. -The voter is partially implemented in the FPGA, which potentially speeds up the voting process and lightens the corresponding overhead on the processor.	- The voting partition is a single point of failure.

- The proposed solution must be independent of the hypervisor used. In case it is tied to a hypervisor, it is preferable that it is not based on the Linux kernel, but that it is as simple and safety-oriented as possible.

In addition, two features are proposed that would help boost the performance and safety of the proposed solution:

- The solution should implement the complete voter logic on the FPGA, including its redundant counterparts. This will improve the performance of the voter, lighten the CPU load (especially since the voter is intended to be redundant) and allow the voter to be made barely vulnerable, since it can be protected (in addition to redundancy) by using a radiation tolerant FPGA.
- The architecture must include a Health Monitoring partition that allows, in case of malfunctioning of a partition, to take the appropriate measures to try to recover it and put it back in proper working order. These measures include tracking the health of partitions, so that they can be replaced by healthy ones if they start to malfunction frequently.

4.2. Architecture Definition

Similar to the solutions discussed above, the proposed architecture consists of at least three different layers:

1. **Hardware Layer**. The software architecture is designed to be deployed on a SoC/MPSoC. Note that, the more resources the SoC/MPSoC has (processor cores, FPGA capacity, memory size...), the more partitions and applications can potentially be deployed on it.
2. **Hypervisor Layer**. The hypervisor is one of the key elements of the architecture, since it allows the isolation between software partitions, which is key to (1) offer secure redundancy between applications, (2) allow the coexistence of applications of different criticality on the same hardware platform.
3. **Partition Layer**. This layer encompasses everything that runs on top of the hypervisor. It could be further divided into:

   OS Layer. This layer is optional, in fact, since there will be occasions in which simplicity/determinism is sought and the choice is made not to use an operating system, but rather that the applications run directly on the hardware virtualized by the hypervisor.

   Application Layer: This layer would encompass everything that runs on top of the operating system (or the hypervisor, in the case of bare-metal applications).

Note that this layer could also be split into other layers depending on the specific use case, but is simplified to one for the sake of clarity.

The types of partitions, depending on whether they are oriented to host safe/secure applications or general/non-safe applications, as well as whether they use an operating system or not, are shown graphically in Figure 4.1.

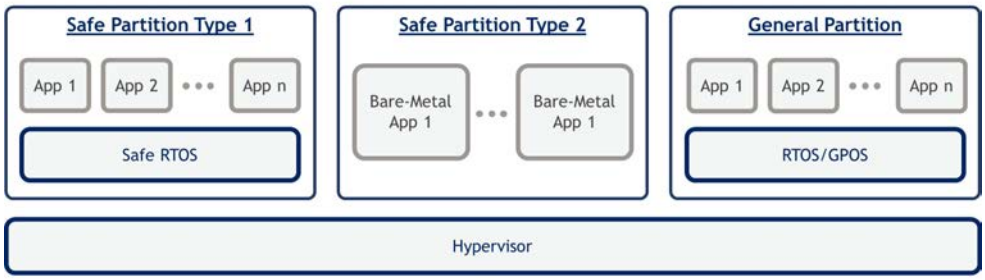

Figure 4.1: Possible partition types

The Partition Layer can be divided into different domains, transversal to the layers. In our architecture, we can distinguish a **Safe Domain**, in which the application to be protected by TMR (as well as its redundant counterparts) runs, and a **General Domain**, in which all applications that do not require special protection and are not redundant run. Depending on the type of system, applications with and/or without hard real-time requirements (on an RTOS such as RTEMS or FreeRTOS or on a GPOS such as Linux) can run in the General Domain. Finally, a **Health Monitoring** partition is responsible for monitoring the correct functioning of the hypervisor and the partitions that make up each of the domains. The layers of the Partition Layer architecture and domains can be seen graphically in Figure 4.2.

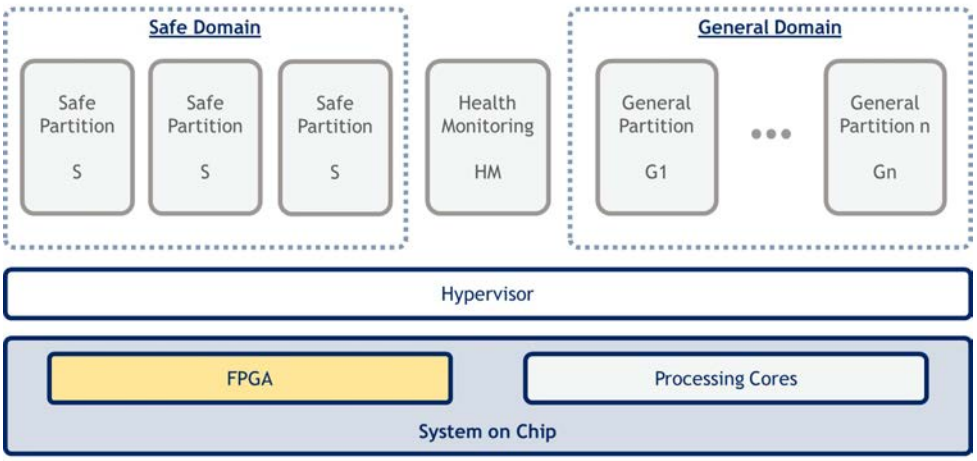

Figure 4.2: Architecture of the proposed solution

As explained in section 2.3, there are many options for redundant software tasks in order to obtain a fault-tolerant system. The most traditional approach is TMR, while the different alternatives that have emerged subsequently are usually aimed at reducing redundancy overhead while sacrificing as little performance as possible. However, for **our solution we are going to opt for redundancy through TMR** for two main reasons:

- Firstly, because in the scenario we are contemplating, the COTS components on which the software runs are powerful enough so that the **overhead caused by redundancy does not cause any harm to the system**.
- Secondly, because in this type of system, **application determinism is more highly valued**, meaning that it is possible to know exactly when and for how long each application will run. In this way, the rest of the system is easier to plan if the same applications are always run for the same time (unlike, for example, in the case of TTMR).

It might seem sensible to use **NMR instead of TMR**, to redundant Safe Partitions. Indeed, it is an option, since, normally, applications running on a Safe Partition are not very heavy in terms of memory occupation. To give an example, if we encapsulate a benchmark representative of a real space application, such as NIR HAWAII-2RG (more details on this benchmark can be found in section 5.3), in a partition on the XtratuM hypervisor and a COTS such as the Zynq-7000 SoC, we observe that the partition occupies a total of 215,596 bytes. This size, compared to the size of the memory devices next to which this SoC is usually deployed (typically, 1 GB of DDR3 SDRAM), can be considered almost **negligible, since it is barely 0.02% of the total available memory**. Even in more complex cases (for example, if we were to add an RTOS to the Safe Partition), the partition size is still very small compared to the total available memory. For this reason, it would be perfectly reasonable not to skimp on the number of redundant Safe Partitions. However, what can be a limiting factor in many occasions is the number of partitions we can fit in the same scheduling plan, taking into account that, sometimes, Safe Partitions must run several times per second (for example, processes associated with attitude and orbit control of a satellite usually run in ranges between 0.5-10Hz [172] [173]). Although the power of COTS processors is also significantly higher than the power of processors traditionally used in the space sector, if we add to this the fact that the scheduling cycle must include Health Monitoring tasks and other system applications, such as General Partitions, the ability to combine all the applications in a single scheduling plan is clearly a more limiting factor than the available memory.

For this reason, we henceforth opt for TMR to expose this solution, and in section 4.4.2 we propose cold redundancy methods to strengthen the TMR scheme without saturating the scheduling plan. Note, however, that using NMR in cases where a very powerful processor is available can be a good way to further strengthen the solution. Note also that the logic of this solution is directly extrapolable to a use case where NMR is applied,

having in the latter case a majority/plurality [174] voter to decide which partitions are operating correctly and which are not.

In addition, as suggested in section 2.3, the voter is also redundant to prevent it from being a single point of failure. In order for this not to incur a very large overhead (in terms of code size and execution time), the triple voter will be implemented in the FPGA. Thus, as shown in Figure 4.3, the triple-voter will coexist in the FPGA with the IP cores that implement the specific functionality required by each mission.

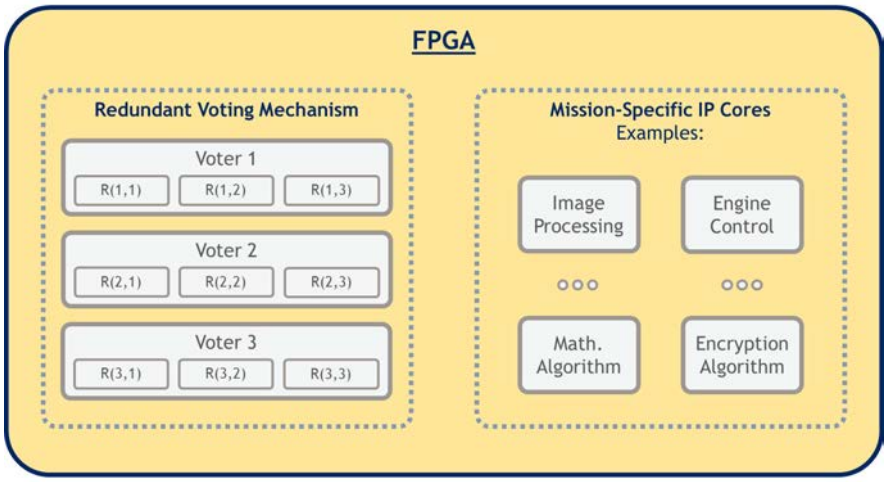

Figure 4.3: FPGA Components

4.3. Voter

As described in previous sections, the voter logic, including its redundant counterparts, will be implemented in FPGA. Therefore, it has been decided to keep its logic as simple as possible. The voter will receive two inputs from each of the active Safe Partitions that will participate in the voting process:

1. The identifier of each of the partitions participating in the voting process.
2. The result calculated by each of the partitions that have identified themselves.

With this, it executes a series of conditional statements and reaches a conclusion, which it will communicate at each iteration to the Health Monitoring partition. The decisions made on the basis of this conclusion are part of the remit of the Health Monitoring partition, not of the voter. However, like the Health Monitoring partition, **the voter will keep his own tally of the number of times a partition has failed, so that tie-breakers can be established in cases of disagreement**.

4.3.1. Voter Operating Modes

As will be seen later, there are cases where partitions can be halted, if they show anomalous behaviour on too many occasions and pose a risk to the rest of the system. Therefore, it is possible that, at certain times, the voter may not have three partitions to compare results. Depending on the number of active partitions, we define three modes of operation of the voter:

- **Normal mode**: this is the optimal case, in which the voter has the results of **three partitions** (one partition plus two redundant counterparts).
- **Degraded mode**: this is the case where there are only **two active partitions** (one partition plus one redundant counterpart).
- **Very degraded mode**: this is the case where only one active partition remains and has **no redundant** counterparts.

Normal Mode

In normal mode, the voter counts the votes of three active partitions. From each of them it receives an identifier and a result, so the voter receives a total of **six inputs**. The voter's task is simple: compare the results of each of the partitions and reach a conclusion, in the form of **three different outputs**.

1. **Result**. The result is decided by majority vote. In case of a tie (which can occur only if the three partitions compute three different results), the voter chooses the result of the partition that has failed the least number of times previously. In case of a tie in the number of times two or three partitions have failed, it is decided randomly.
2. **Reliability level**. The reliability level depends on the number of partitions that agreed on the result decided by the voter:
    - If all three partitions agree on the calculated result, the voter sets a **VERY HIGH** reliability level.
    - If two of the three partitions agree on the calculated outcome, the voter sets a **HIGH** reliability level.
    - If all three partitions disagreed on the calculated result, the voter sets a **LOW** reliability level.
3. **Faulty partition**. The voter also reports which partition is considered to have failed. Note that this depends directly on the reliability level:

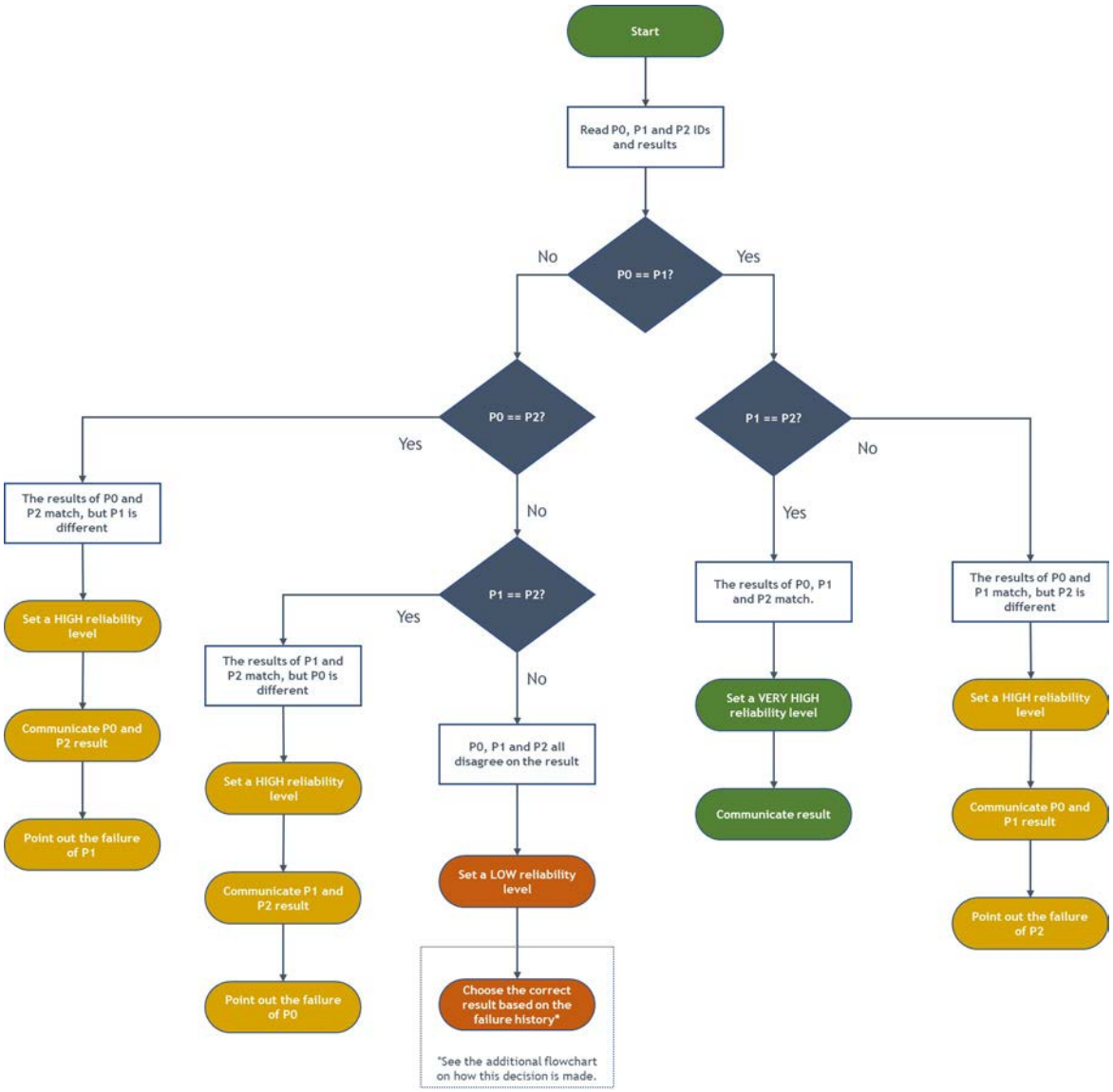

Figure 4.4: Voter logic - Normal Mode

- If the reliability level is very high, no partition has failed, so this field will be empty or it should report that no partition has failed.
- If the reliability level is high, only one partition has failed, so the voter will report the ID of the failed partition.
- If the reliability level is low, even if the result of one of them is chosen based on the failure history or on chance, the failure of the other two is not recorded, because there are not enough guarantees and it could be detrimental to reduce the reliability level of these two partitions.

This logic is represented graphically in the flowchart in Figures 4.4 (which represents the general Normal Mode scheme) and 4.5 (which represents the tie-breaker when the three active partitions yield different results).

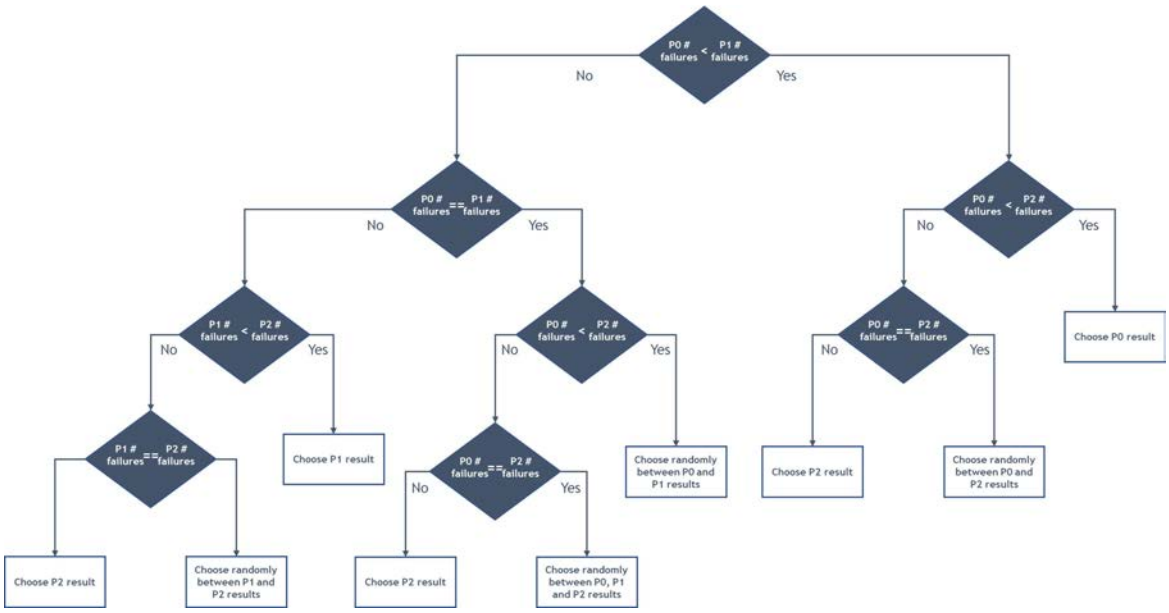

Figure 4.5: Voter logic - Tie-breaker in Normal Mode

Degraded Mode

The degraded mode is a simplification of the normal mode. In this case, because the Health Monitoring partition has decided to halt all other partitions, there are only two active partitions left that can participate in the vote. Therefore, the voter receives a total of four inputs: from each of these partitions, an identifier and the result it has calculated. Similar to the normal mode, the voter compares the results of each of the partitions and reach a conclusion, but in this case in the form of **two different outputs**.

1. **Result**. Note that, by having two partitions, the degraded mode still allows us to check that, in case both agree, there are enough guarantees of the correctness of the result. In case of disagreement, the result chosen will be that of the partition that has failed the least number of times previously. In case both have failed the same number of times, it will be chosen randomly between the two.

2. **Reliability level**. In degraded mode, the voter only offers two different reliability levels:

   - If the two partitions agree on the calculated result, the voter sets a **HIGH** reliability level.
   - If the two partitions disagree on the calculated result, the voter sets a **LOW** reliability level.

Again, **although having two partitions allows to guarantee a certain level of assurance regarding the correctness of the result when the partitions agree, it is impossible to guarantee which of the two partitions is failing when they do not agree.** Therefore,

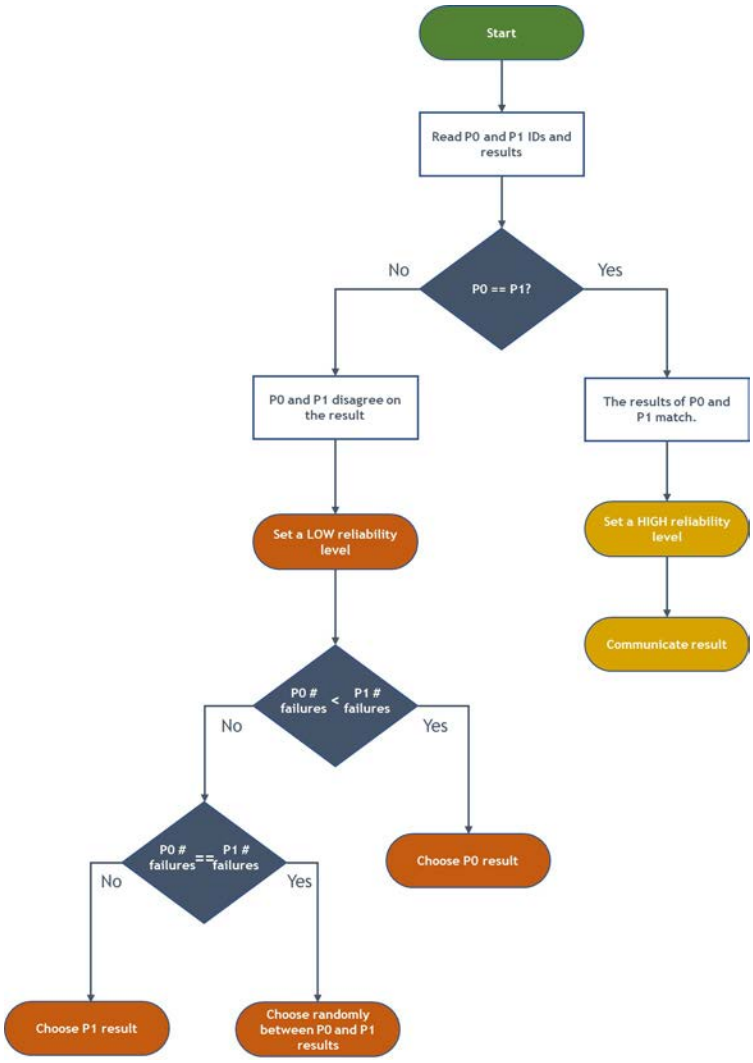

Figure 4.6: Voter logic - Degraded Mode

even if the result is decided by the failure history of the partitions, the voter will not decide which is the faulty partition (note that it would not make sense to keep increasing the failure counter of the partition that has failed more times in the past, since it would always be decided that the same one is failing).

The logic of the degraded mode is represented graphically in the flowchart in Figure 4.6.

Highly Degraded Mode

The highly degraded mode is an extreme case in which the voter is simplified as much as possible. Note that, in this case, there is only one active partition, so the voter receives only two inputs: the partition identifier and the calculated result.

This mode of operation has no internal logic and simply produces two outputs:

1. **Result**. Having no other results to compare with, the voter assumes as **correct the**

result of the only remaining active partition.

2. **Reliability level**. The voter **always** assumes that the reliability level for that result is **LOW**, as it has not been able to corroborate it with other partitions.

The simple logic of the highly degraded mode is represented graphically in the flowchart in Figure 4.7.

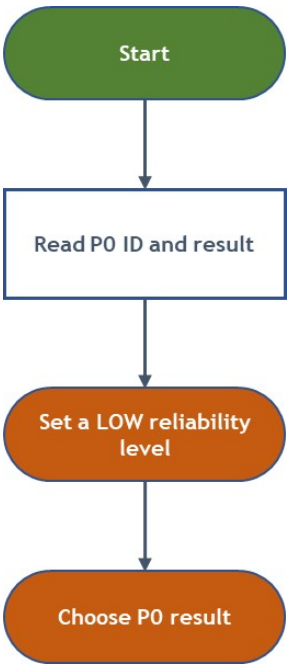

Figure 4.7: Voter logic - Highly Degraded Mode

Behaviour in Cases of Low Reliability

As can be seen, although at least three redundant partitions are used to avoid such cases, there are occasions when the voters calculate a result but give it a **low level of reliability. The decision of how to operate** in these cases will be made at the system level and **depends entirely on the nature of the mission or objective of the system**. It is clear that this solution is oriented to safety-critical systems and what the voters decide will always be relevant to the safety of the system, but within this category we can obtain a wide range of scenarios. Mainly, with respect to how to act when the reliability of the calculated result is low, we can classify them in two types:

- **Scenarios in which the least risky thing to do is to ignore the calculated result and wait for another iteration**. Examples of these scenarios could be aircraft guidance functions, which often use algorithms (such as Kalman Filters) and in which an incorrect result would be carried forward in successive iterations and yet the results of previous iterations could alleviate the fact that no calculation was performed in the current iteration.

- **Scenarios where it is equally or more detrimental to not reach a conclusion on the value of the calculated result**. An example of this scenario could be a real-time communications decryption algorithm, where failing to decrypt the frame received by the system in one iteration can be just as detrimental as decrypting it wrong.

Note that, in the first case, it may be most reasonable not to use the result that the voters produce in a given iteration when it has a low reliability, to avoid higher risks; while in the second case, it may be worthwhile to perform a best-effort guess and consider

the result calculated by the voters as valid despite its low reliability. Again, this decision will depend on the specific mission or objective of the system and must be evaluated on a case-by-case basis.

4.3.2. Redundant Voters

As stated in Section 4.1, one of the keys to differentiate it from similar works such as Campagna et al. [77] or Esposito et al. [162] the voting mechanism should not involve a single point of failure. In this case, by implementing the voter in the FPGA, it is especially easy (and implies a very low overhead) to redundant the mechanism, which will be replicated three times in three different areas of the FPGA. This will allow to remain confident in the outcome of the voter even if one of them is damaged. However, it is expected that all the three voters will always agree, since the same partitions provide them with inputs, so the decision logic when comparing the results of the three voters will be simpler:

1. The **result** is decided by majority. If two of the voters agree on a result, it is established as the correct result. If there is a total disagreement between the three voters, this is a very anomalous case, since very different things would have to happen within each voter (who, in addition, receive inputs from the same sources), so it is considered a critical failure of the system and would cause a reset of the hypervisor. If the hypervisor has been reset too many times, as stated above, the system would enter Safe Mode.

2. The **reliability level** of the result is decided by **majority** vote. If all three voters agree on the result but there is total disagreement on the level of reliability (i.e., one reports a reliability level VERY HIGH, one HIGH and one LOW), the reliability level is set to HIGH. If only two voters agree on the result and disagree on the reliability level, the lower reliability level is chosen from among the two voters.

3. The **faulty partition** is decided by **majority** vote. If at least two of the voters indicate that one or more partitions are faulty, it is decided that they are faulty. If only one partition indicates that any of the partitions are faulty, no action will be taken. Note that this means that no action will be taken outside the voters' domain (i.e., Health Monitoring will not assume failure on the part of that partition), but internally that voter will have increased the failure counter of the partition it has considered faulty.

This logic is represented graphically in the flowchart in Figure 4.8.

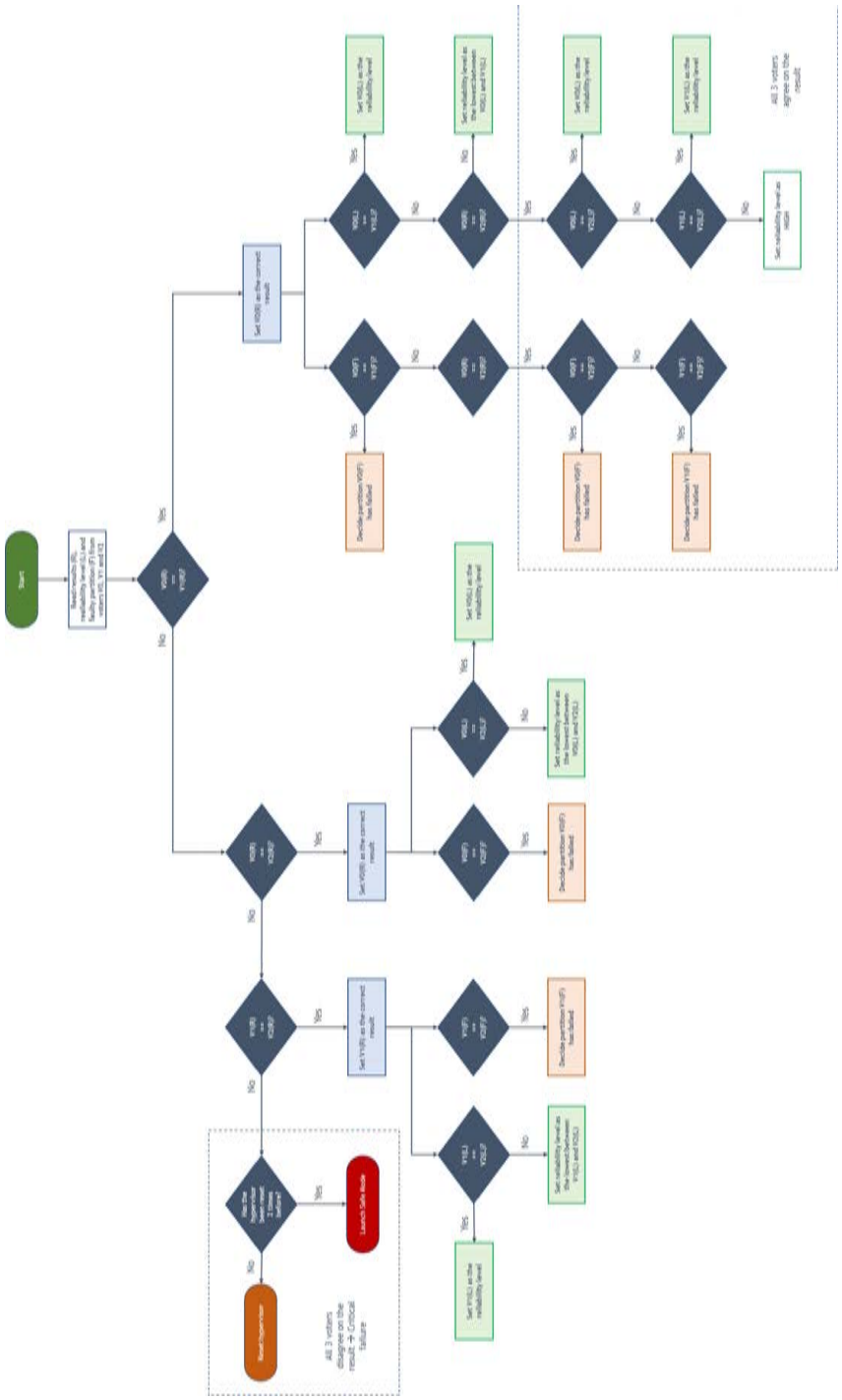

Figure 4.8: Management of redundant voter decisions

4.4. Health Monitoring

The Health Monitoring partition is an essential element of the architecture. Its function is twofold: on the one hand, to constantly monitor the health and correct functioning of the hypervisor and of each of the partitions. On the other hand, either because it detects an error or because the *voter* informs it that a partition has failed in its calculations (which could indicate a malfunction of that partition), the Health Monitoring partition is responsible for taking the appropriate measures to restore the health of the system as far as possible. The measures can vary from resetting a partition, restarting the hypervisor itself or even completely changing the system configuration.

4.4.1. Hypervisor health check process

In the work presented in this thesis, the hypervisor is the means used to provide a fault-tolerant solution at the application level. For this reason, hypervisor fault detection and correction are outside the scope of the research. In addition to the mechanisms that each hypervisor implements to protect itself, in section 2.4 a series of investigations aimed at strengthening the correct execution of the hypervisor by means of different techniques have been presented. Even so, the solution presented in this thesis contemplates a basic check of the hypervisor's health, which is explained in this section.

Typically, hypervisors follow a finite-state machine model in which at least a normal operational state, an initialization state and an error state can be distinguished. The HM partition must constantly check that the hypervisor is in the expected operating mode. If it is not possible to access this information, or even in parallel to the hypervisor status check, the HM partition shall (after identification) monitor the variables/parameters that allow verifying that the hypervisor is operating correctly.

In the event of a hypervisor error, the HM partition will reset the hypervisor and restart the same partitions that were running up to that moment, provided that the nature of the mission allows it. If the HM partition detects several consecutive errors (the exact number will depend on the specific scenario and mission), it will restart the hypervisor by loading the safe configuration. This configuration (often referred to as safe mode) must be planned prior to each mission and must be triggered automatically in case of critical failure, without confirmation from the ground. The safe mode should maintain the system in a state of reduced but stable functionality until appropriate actions are taken remotely to restore normal operation by the operations team. Although not every failure triggers the launch of safe mode, the HM partition must communicate any actions it takes on the hypervisor or partitions to the operators, even when it is able to autonomously recover the system.

This whole process can be seen graphically represented in the flowchart in Figure 4.9.

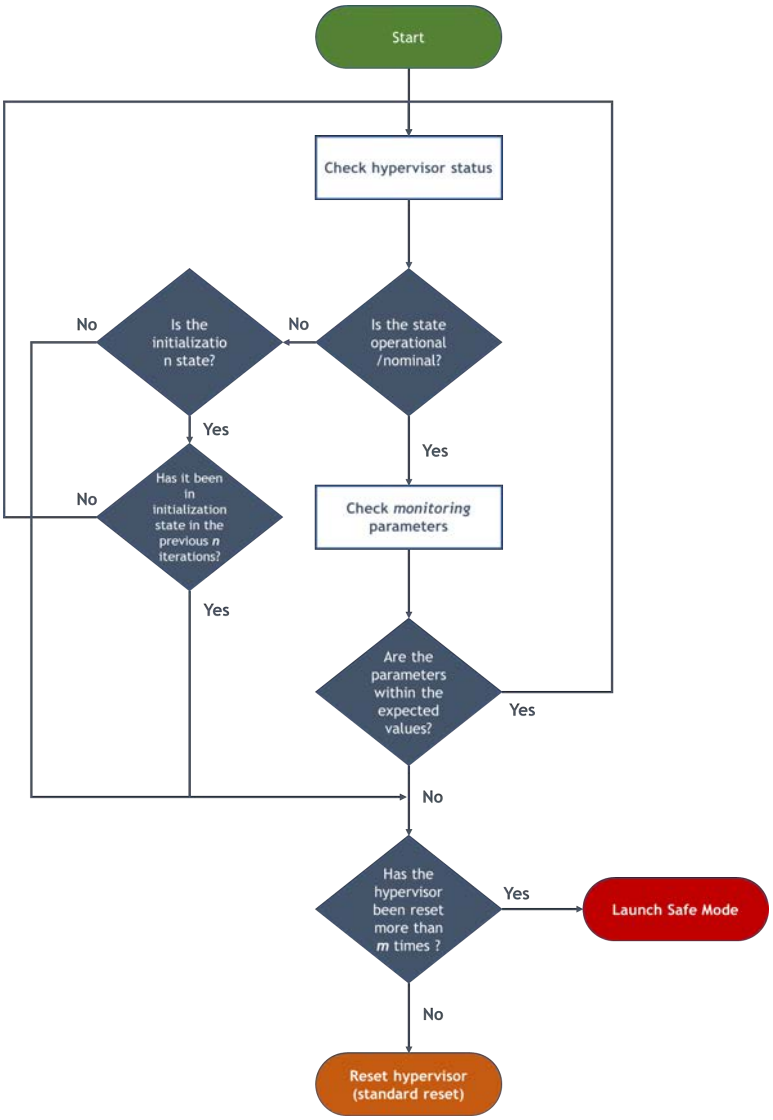

Figure 4.9: Hypervisor health check process flowchart

4.4.2. Partitions health check process

As explained above, we have opted for a TMR scheme rather than an NMR scheme to expose the solution. This is because, although the memory of the COTS devices targeted by this solution is usually large enough to accommodate a high number of redundant copies, in many occasions we can find a limiting factor in the number of partitions that can be scheduled in one cycle of the scheduling plan. However, one way to strengthen the TMR scheme without further increasing the number of partitions running simultaneously is to use cold redundancy mechanisms. In other words: in addition to applying TMR, it is perfectly reasonable to store in memory one or more Safe Partitions that remain on standby and are activated when the Health Monitoring partition considers that they should replace one of the Safe Partitions currently running. In this way we obtain a double advantage:

1. Further strengthen the hot redundancy that we get by running several Safe Partitions

in parallel with cold redundancy, by having more Safe Partitions in standby.

2. Allow to permanently replace any of the Safe Partitions running in hot redundancy, in case it shows a malfunction during long periods of time, without degrading the performance of the system.

After the health of the hypervisor is checked, the Health Monitoring partition should also check that the health status of each partition is correct at each iteration. To do this, it can check monitoring parameters and the internal parameters of each partition, which should show the expected values (the specific details of this check depend on the particular hypervisor being used). In case we use a simple hypervisor that does not provide too much information about partition status, or even in addition to these checks, partitions can be instructed to send a *keepalive* message to the HM at each iteration, so that the HM notices if a partition is down without having to check any of its parameters.

If any Safe Partition is not in the expected state or has crashed and its *keepalive* message is not received, the HM resets the partition. Resetting a partition should be an anomalous and very occasional situation, so if the same partition has been reset several times (the exact number depends on the particular system, we will call it K), it is assumed that it is severely malfunctioning, so the partition is halted in order not to compromise the health of the rest of the system and to prevent it from consuming resources unnecessarily. In case the partition is halted, the HM immediately checks if there is another Safe Partition in cold redundancy that can replace it. If there is no other partition in cold redundancy, the failed partition will be halted anyway, and the HM will configure the Voter to run in Degraded Mode (if there are two Safe Partitions left) or Highly Degraded Mode (if there is only one left). It is possible, depending on the hypervisor, the partitions themselves and the power of the platform on which they run, that the process of resetting a partition or replacing an active partition with a partition that is in cold redundancy may not be immediate. Therefore, as part of the process of **resetting or replacing a partition, the HM will instruct the voter to enter degraded mode until the new partition is fully reset and operational**.

The same logic applies to General Partitions, keeping in mind that there may be no hot or cold redundant partitions for these partitions. In this case, the HM will simply check their health, reset the partition in case of error and halt the partition if the error is repeated too many times.

Regarding itself: for obvious reasons, the HM cannot be trusted to monitor its own health correctly (since, in case of malfunction, we would not be able to trust its decisions). Still, as an additional precaution, the HM performs a check of its current state. If it finds any error, given the importance of the HM in the system, it will be considered a critical error and the hypervisor will be reset. Just as when checking the health of the hypervisor, there is a maximum number of times the hypervisor can be reset before launching Safe Mode. It is important to note, however, that the rest of the partitions could run correctly without Health Monitoring, although there would be no protection in case any of these

partitions fail. It is also important to clarify that the way to add an additional layer of protection would be to establish a new agent (it can be another partition) with minimalist functionality, which exclusively monitors the health of the HM and resets it if necessary.

This logic is represented in the flowchart in Figure 4.10.

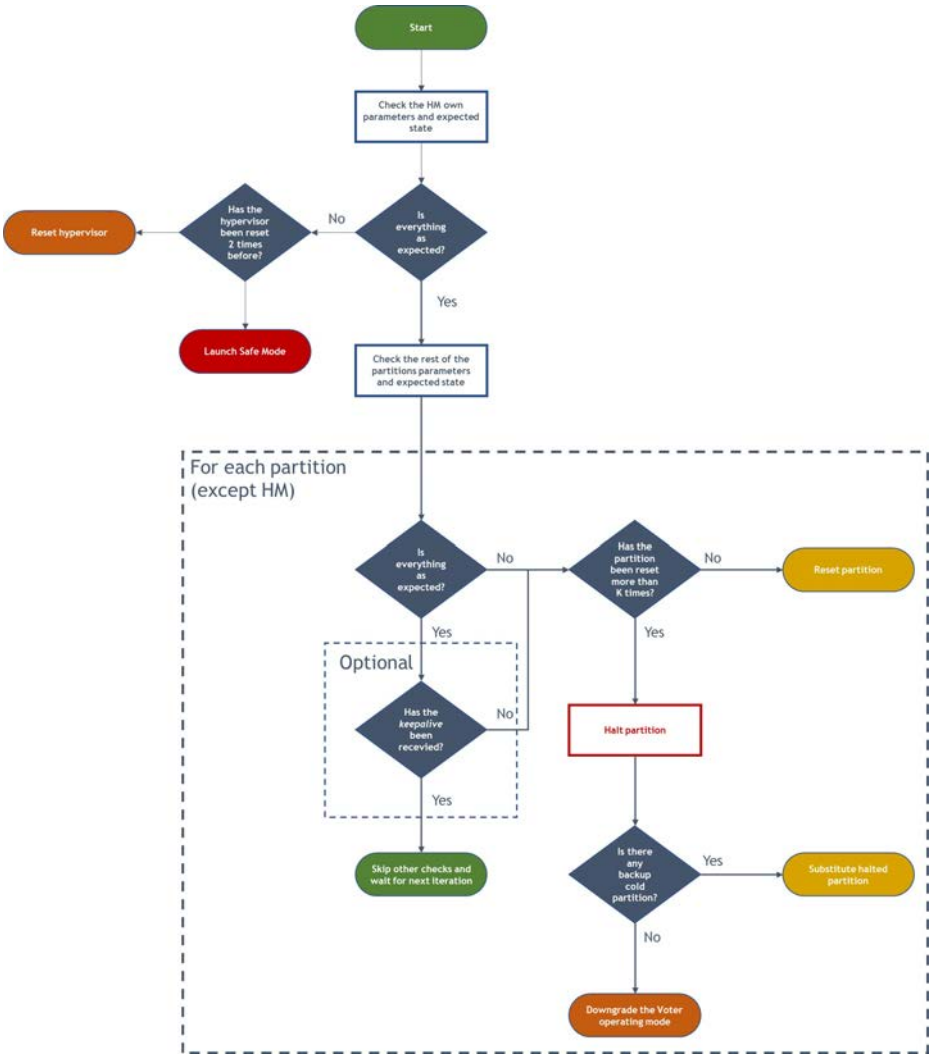

Figure 4.10: Partitions health check process flowchart

4.4.3. Partitions results check process

Finally, after checking the health of the hypervisor and each partition on the system, the Health Monitoring partition checks if any of the redundant safe partitions have calculated an erroneous result, as judged by the Voter (see section 4.3 for more details on the voting process).

Like the Voter, the Health Monitoring partition keeps track of the number of times each partition has failed. However, these counts are independent of each other and are used for different purposes: while the Voter uses the number of failures of a partition to quantitatively assess its reliability, in order to choose the result calculated by that parti-

tion over the result calculated by another redundant partition, the HM uses the count to qualitatively assess whether the partition is healthy enough to continue running without compromising the safety of the system.

The first step in the result checking process is the reading of the Voter outputs, which will signal a partition in case it has failed. If a partition has failed, the HM will increment its failure counter. If this counter exceeds a safety number (which will depend on the particular use case), the partition will be assumed to be continuously malfunctioning, and it will be reset. As in previous processes, if the partition has been reset too many times, it will be halted permanently. In this case, the halted partition will be replaced as long as there are partitions in cold redundancy that can take its place, and if not, the system operation mode will be degraded.

After this, the HM will check, for each active partition, if there is a partition in cold redundancy whose failure counter is less than the number of failures of the active partition. If so, it will proceed to replace the active partition with the partition with fewer failures. The goal of this action is to always run the three partitions that have historically failed the least.

This process is illustrated graphically in Figure 4.11.

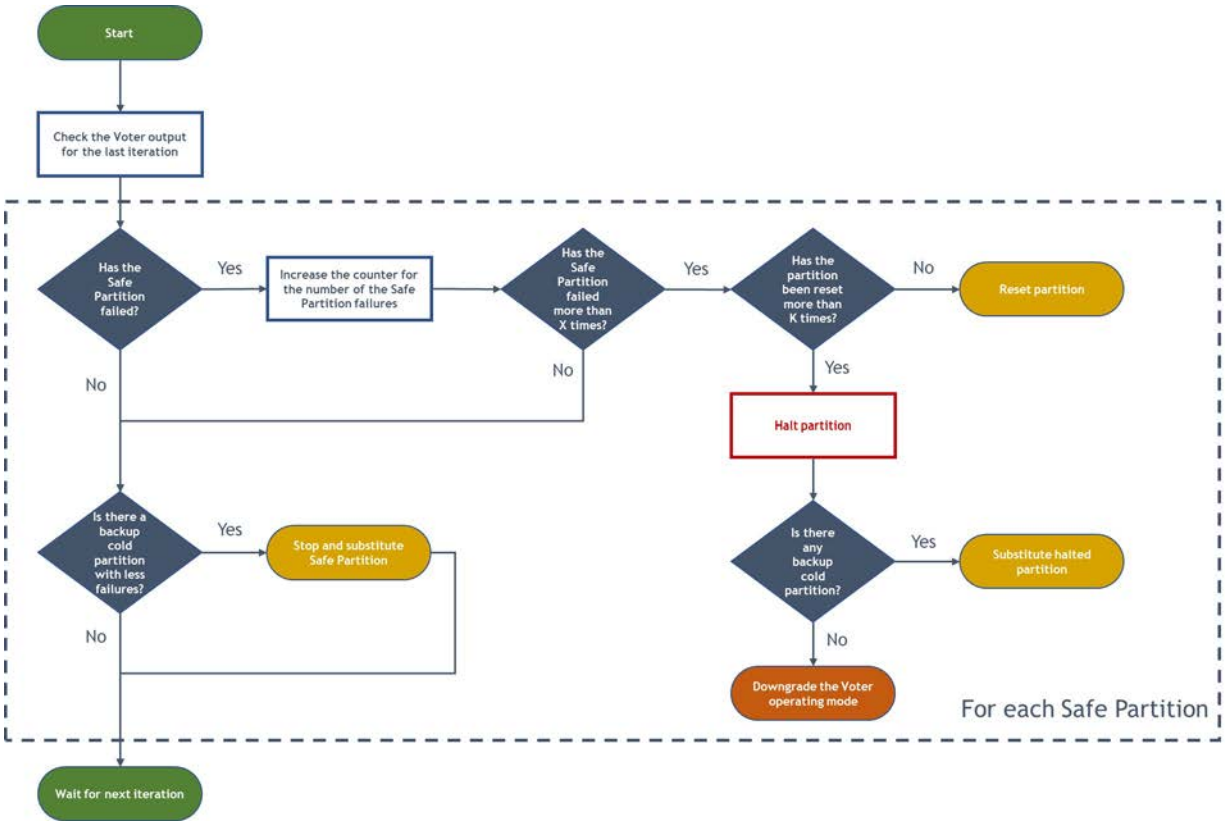

Figure 4.11: Partitions results check process flowchart

4.5. Summary

In this chapter, we have outlined our proposed solution for implementing a novel HBFT mechanism. On the one hand, a novel contribution of our solution is the Health Monitoring partition, which is responsible not only for monitoring the state of the hypervisor and the other partitions, but also for taking appropriate recovery measures if any part of the system is malfunctioning. Another novel aspect of the solution is the deployment of cold redundancy partitions, which are on standby until Health Monitoring commands them to replace any of the active partitions if they are failing too frequently. Finally, our system's Voter, which runs entirely on the FPGA, is a complex element with several modes of operation, depending on the number of partitions participating in the voting process. This Voter also tracks the success rate in the calculations of each partition, so that it can assign a reliability to each partition. This novel feature allows the system to be robust to multiple failures, as we will see later.

After outlining our solution, Chapter 5 explains the development of a simplified but representative prototype of all the functionality explained in this chapter.

5. PROTOTYPE

This chapter presents a prototype demonstrating the solution proposed in Chapter 4. The chapter is divided into seven main sections:

- Section 5.1 contains the description of the general architecture of the prototype, which is a particularisation of the architecture presented in chapter 4. It includes the selection of the hypervisor, according to the particular needs of the prototype and to the state-of-the-art study carried out in chapter 2.
- Section 5.2 describes the hardware used to develop the prototype (based on SoC technology) and the bootloaders used to configure the hardware at boot time.
- Section 5.3 comprehensively describes the aerospace software used to demonstrate the fault tolerance of the solution: the NIR HAWAII-2RG Benchmark.
- Section 5.4 describes the two modes of operation of the prototype. This section includes the memory allocation of each partition of the modes of operation and their scheduling plan.
- Section 5.5 describes a series of tests performed on the prototype's Health Monitoring partition to verify that it responds adequately to the different failures that may occur in the system.
- Section 5.6 describes the prototype voter, a simplification of the voter discussed in Chapter 4. This section includes both a software-implemented version of the prototype and a version implemented in the programmable logic of the chosen SoC.
- Section 5.7 briefly summarises the chapter and introduces the contents of the following chapter.

5.1. Prototype Architecture

According to section 4.2, the prototype architecture will have three clearly differentiated layers:

- **Hardware Layer**. The board chosen to deploy the prototype is the **MicroZed**, a low-cost development board based on the **Xilinx Zynq-7000 SoC**. Some details about this board are explained in section 5.2.
- **Hypervisor Layer**. In Chapter 4, an exhaustive study of the existing literature on the application of hypervisors to real-time safety-critical systems has been carried

out (the result of this study is summarized in Tables 2.4, 2.5, 2.6 and 2.7). Among all the options, the **XtratuM hypervisor has been chosen** to implement the prototype, for several reasons:

  - It is a hypervisor conceived from the outset for real-time safety-critical systems.
  - It offers paravirtualization, which should potentially alleviate the hypervisor's footprint on the system[1] .
  - It offers, among others, support for ARM Cortex-A9 processors, which is the one chosen to develop the prototype due to availability at the time of the research.
  - It allows to deploy, in addition to real-time partitions (bare-metal, RTEMS, LithOS), partitions with Linux as operating system, which allows reusability and deployment of legacy applications.
  - Widely promoted in the aerospace industry, both by private companies and public bodies such as ESA and CNES.
  - Its internal structure, which follows the ARINC 653 style (see section 2.1.3 for more details on this specification), makes it easily adaptable to aviation systems, beyond the space sector.
  - There is work, such as that of the OVERSEE [79] project, that looks at the possibility of applying XtratuM to automotive systems and has done work in that direction.
  - Finally, by ease in the framework of the project in which this industrial doctorate is developed. XtratuM is a Spanish company and there are currently working ties with both SENER Aeroespacial and Universidad Carlos III de Madrid, the two entities involved in this thesis.

- **Partition Layer**. A total of **five partitions** will be deployed for the prototype:
  - The **three Safe Partitions** instances will run three identical applications, based on the **HAWAII-2RG NIR space benchmark**. This benchmark will allow us to base the prototype on an application that is used in current space observing systems (more details on the benchmark are discussed later in section 5.3).
  - **One partition** will be dedicated to running the **Health Monitoring**, following the processes described in section 4.4.
  - A final partition will be dedicated to a dual objective. In a **first version** of the prototype, in order to be able to start testing it as soon as possible, this partition will implement a **software version of the voter**. **Later on**, the voter will

[1] There is no evidence on the footprint implied by the use of XtratuM on any platform or architecture. Part of the research will be dedicated to perform overhead tests on the hardware platform we have chosen, so that the first data can be obtained.

be implemented on the FPGA as described in chapter 4, so that, in addition, its performance can be compared with respect to the software version. At that time, the resources (in terms of memory and CPU utilization) dedicated to the voter partition will be used to implement **a general partition** (i.e., as explained in section 4.2, a partition hosting a general/non-safe application).

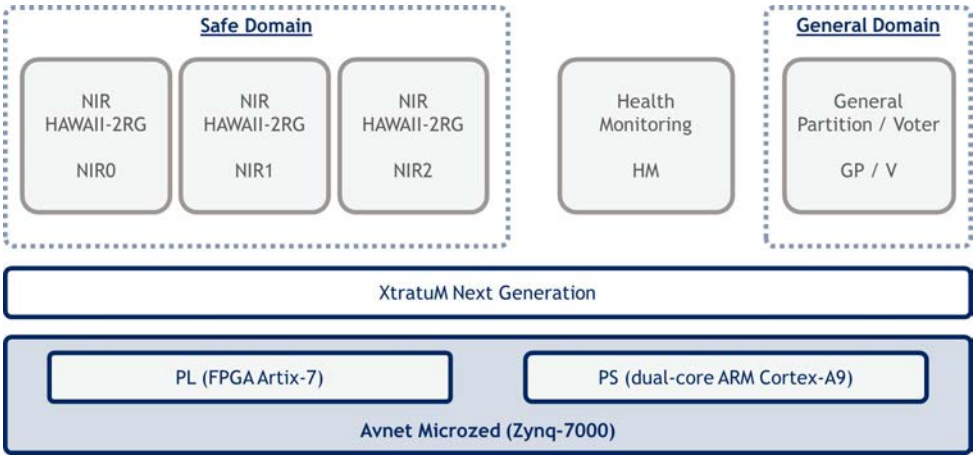

Figure 5.1: Prototype Architecture

5.2. Hardware Layer

The prototype used to demonstrate the solution proposed in this thesis is deployed on the low-cost development board Microzed, from Avnet. In turn, this board is based on the Zynq-7000 SoC, from Xilinx. This section is dedicated to briefly outline the characteristics of the hardware used, as well as to present the bootloaders used to configure it before the hypervisor, as the main software element, takes control of the system.

5.2.1. System on Chip

In section 2.1.3 we discussed how industries such as space, aviation or automotive are tending to follow other industries in using System on Chips (SoCs) to implement safety-critical system solutions. A SoC is nothing more than a chip on which all the typical components of a complete electronic system are integrated [175]. This usually includes a CPU, FPGA, memory interfaces, I/O devices and sometimes more specific devices such as ADC converters, transceivers or graphics processing units. When an SoC has more than one processor among its components, they are often referred to as MPSoCs (Multi-Processor System on Chip) [176]. Today, there are commercial SoCs that offer very high performance, meeting the demand of more and more sectors (including safety-critical ones) to be able to host high-demand applications in very little physical space, which means that they offer very high efficiency, both economically and in terms of SWaP (Size, Weight and Power consumption).

The adoption of these devices in safety-critical industries involves a series of important challenges that are being and will continue to be investigated in the coming years, mainly related to interference between internal components if one of them fails. Something similar happens at the processor level: the high processing capacity of the processors that these SoCs integrate makes it possible to deploy a lot of functionality in a single processing core, so isolation and redundancy techniques that guarantee the safety of each application are essential. This is the focus of the research reflected in this thesis and, of course, the prototype used to test it should be deployed in an SoC. The SoC chosen, due to its characteristics, the maturity of the associated software tools and its availability at the time of the research, is the Zynq-7000 from Xilinx. Its main features are described below.

Zynq-7000

Zynq-7000 refers to a whole family of SoCs that integrate, among others, a processing system (PS) based on a single-core or dual-core ARM Cortex-A9 processor and a programmable logic (PL) based on a 28nm FPGA from Xilinx. Both domains (PS and PL) are connected by more than 3,000 interconnects offering a bandwidth of up to 100Gb/s, which is above what a solution with separate chips could offer.

The differences between the different models covered by this family are centered on the number and power of the processing cores and the capacity of the FPGA they integrate. Avnet's Microzed board is based on one of the Zynq-7000 models. Specifically, on the XC7Z010-1CLG400C SoC, details of which are reflected in Tables 5.1 (Processing System) and 5.2 (Programmable Logic).

Table 5.1: Z-7010 Processing System characteristics

Device Name	Z-7010
Part Number	XC7Z010
Processor	Dual-Core ARM Cortex-A9 MPCore (up to 866MHz)
Processor Extensions	NEON™ SIMD Engine and Single/Double Precision Floating Point Unit per processor
L1 Cache	32KB Instruction, 32KB Data per processor
L2 Cache	512KB
On-Chip Memory	256KB
External Memory Support	DDR3, DDR3L, DDR2, LPDDR2
External Static Memory Support	2x Quad-SPI, NAND, NOR
DMA Channels	8 (4 dedicated to PL)
Peripherals	2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO
Peripherals w/ built-in DMA	2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO
Security	RSA Authentication of FSBL, AES and SHA 256b Decryption and Auth. for Secure Boot
PS to PL Interface Ports	2x AXI 32b Master, 2x AXI 32b Slave, 4x AXI 64b/32b Memory, AXI 64b ACP, 16 Interrupts

Table 5.2: Z-7010 Programmable Logic characteristics

7 Series PL Equivalent	Artix-7
Logic Cells	28K
Look-Up Tables (LUTs)	17,600
Flip-Flops	35,200
Total Block RAM (# 36Kb Blocks)	2.1Mb (60)
DSP Slices	80
PCI Express	-
AMS/XADC	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs
Security	AES & SHA 256b Decryption & Auth. for Secure PL Config
Speed Grades	-1

Avnet Microzed

The Microzed is an Avnet development board based on the Zynq-7000 SoC from Xilinx. In addition to this SoC, it contains the components and interfaces typically required to support SoC designs: among them, memory devices, USB and Ethernet communication interfaces and clocks.

Table 5.3 lists some of the main features of the Avnet Microzed used to implement the prototype [177]:

Table 5.3: Avnet Microzed characteristics

SoC	XC7Z010-1CLG400C
Memory	1 GB of DDR3 SDRAM
	128 Mb of QSPI Flash
	Micro SD card interface
Communications	10/100/1000 Ethernet
	USB 2.0
	USB-UART
User I/O	108 User I/O (100 PL, 8 PS MIO)
	PL I/O configurable as up to 48 LVDS pairs or 100 single-ended I/O
Other	2x6 Digilent Pmod® interface (8 PS MIO connections for user I/O)
	AMD Xilinx PC4 JTAG configuration port
	PS JTAG pins accessible via Pmod or I/O headers
	33.33 MHz oscillator
	User LED and push button

5.2.2. Bootloaders

As a previous step to hand over control to the hypervisor, there must be a basic application that is normally known as bootloader, which contains all the software needed to boot the system. In the case of the work of this thesis, the bootloader is divided into two distinct parts:

- The **First-Stage BootLoader (FSBL)** is a basic bootloader provided by Xilinx, the SoC manufacturer, that is responsible, among other things, for bringing the processor out of reset, programming the FPGA with the appropriate bitstream and hands over the control to the next bootloader.
- The **Second-Stage BootLoader (SSBL)** is a more complex bootloader developed for this research, in which the necessary tests are performed to verify that the system is ready to start with its nominal operation (Initiated Built-In Tests -IBIT-). In addition, before finishing its execution, this bootloader is responsible for configuring the hypervisor and giving it control. This control flow can be seen graphically in Figure 5.2.

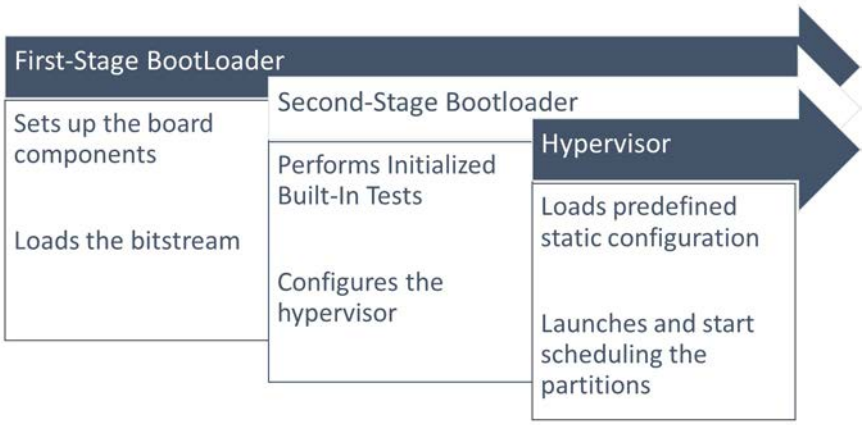

Figure 5.2: Bootloaders Control Flow

IBITs include:

- **DDRAM** memory integrity test.
- **Interrupt Controller** test.
- **DMA** (Direct Memory Access) test.
- **QSPI** (Quad Serial Peripheral Interface) test.

- **Device Configuration Interface** driver test.
- **SCU** (System Controller Unit) **Private Timer** test using a counter (polling mode).
- **SCU Private Timer** test using interrupts (interrupt mode).
- **SCU Private Watchdog Timer** test using interrupts (interrupt mode).
- **TTC** (Triple Timer Counter) device test.
- **UART** (Universal Asynchronous Receiver-Transmitter) interfaces test.
- **GPIO** (General Purpose Input/Output) interfaces test.
- **XADC** (Xiling Analog to Digital Converters) temperature and voltage sensors test.

5.3. NIR HAWAII-2RG Benchmark

Since the fault-tolerance mechanism explained in Chapter 4 is intended to operate in safety-critical embedded systems, it is desirable to implement a use case related to this type of systems in order to test the prototype. For commercial and confidentiality reasons, it is not easy to access real operational software from sectors such as aviation, space or automotive. However, the European Space Agency (ESA) provides access to the benchmarking software Near InfraRed (NIR) HAWAII 2-RG Data Processing Algorithms. As the name suggests, this software implements algorithms that allow processing of raw frames captured by the HAWAII 2-RG NIR detector. This detector is currently widely used in different space missions and projects, such as the Hubble Space Telescope (Wild Field Camera 3), the NIFS Gemini North, the James Webb Space Telescope (Near Infrared Spectrograph) or the X-Shooter of the European Southern Observatory, at Cerro Paranal.

In a simplified way, the different steps of the algorithm for processing the images (which are in the form of a two-dimensional array of pixels), once the data have been collected and converted by an analogue-to-digital converter, are described below:

- **Saturation detection**: the algorithm checks if any of the pixels are over the saturation level, to take this into account in future steps.
- **Co-Adding**: the algorithm adds the frames of each group to reduce read-out noise.
- **Super-Bias subtraction**: the algorithm subtracts a bias frame from the frame obtained by the detector, so that the pixel-to-pixel variation in offset is eliminated.
- **Non-linearity correction**: the algorithm corrects the non-linearity of the detector using a 4th degree polynomial.
- **Reference pixel subtraction**: the algorithm eliminates common noise.

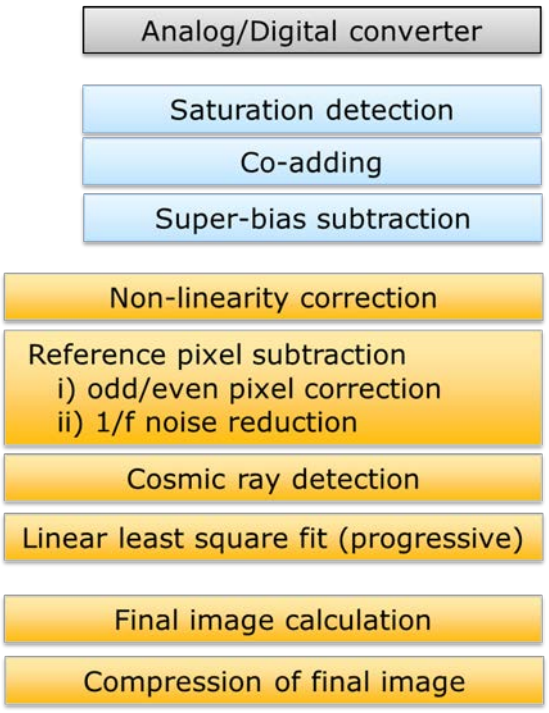

Figure 5.3: NIR HAWAII 2-RG Data Processing Algorithms Steps (ESA)

- **Cosmic ray detection**: the algorithm estimates flux and detected disturbances caused by cosmic rays and corrects them.

In order to make this software useful for our prototype testing, **two main modifications have been made to the original source code distributed by ESA**. Firstly, the necessary modifications have been adopted to make it run on the **Xilinx Zynq 7000 SoC (ARM processor)**, as the original distribution is intended to be deployed on LEON and PowerPC processors. In addition, instead of generating random data, a **fixed set of input data has been defined**, so that different partitions can redundantly run the algorithm in the same conditions and their outputs can be compared for errors.

5.4. Operational Modes

5.4.1. Partitions Memory Allocation

To configure a system running XNG (hypervisor configuration, partitions, scheduling, resource allocation, interrupts...), a series of XML files are used. Through these files and a dedicated software tool, a file called XtratuM Configuration File (XCF) is compiled and loaded into a memory area determined by the user, where the hypervisor can read the configuration and act accordingly.

Although a single XCF can offer some flexibility (e.g., more than one scheduling plan can be configured in a single XCF and the user can switch between them at run time), the

user can also load more than one XCF into memory, and completely change the system configuration during execution. Therefore, when the final executable of a system using XNG is generated, it consists of at least 3 parts:

- The executable of the hypervisor itself (there must be a bootloader which, after performing the initialisation tests, hands over the execution control to the hypervisor. This bootloader has been described in section 5.2.2).
- One or more XCFs (the bootloader must also write in a processor register the configuration of the XCF to be used by default, so that the hypervisor can find it).
- One or more partitions.

In the case of the prototype, two different XCFs are used: one describing the Nominal Mode configuration and one describing the Safe Mode configuration. The idea behind this is that in each of the modes of operation the hypervisor has no visibility or access to the partitions that do not correspond to that mode. This is especially important in the case of Safe Mode, where we want to run basic functionality while waiting for an external agent to reset the system (see section 5.4.3 for more details). In addition, six partitions are loaded into memory:

- **Partition 0**: NIR#0, Safe Partition (first copy).
- **Partition 1**: NIR#1, Safe Partition (second copy).
- **Partition 2**: NIR#2, Safe Partition (third copy).
- **Partition 3**: Health Monitoring partition.
- **Partition 4**: Voter (General Partition when Voter is running on FPGA).
- **Partition 5**: Safe Mode.
- **Partition 6**: NIR#3, Safe Partition in cold redundancy (can replace partitions 0, 1 or 2 when the HM partition deems it necessary).

Each of these partitions is allocated 64MB of memory. This is more than enough for the tasks they have to perform, but even so, most of the Avnet Microzed's DDR3 SDRAM memory (1GB total memory) is still note used. Table 5.4 contains the memory allocation of the different elements of the final system executable.

Table 5.4: Memory allocation of the system executable

	Start Address	Reserved Size (kB)	Real Size (kB)
Hypervisor Executable	0x200000	-	1.55
XCF (Nominal Mode)	0x300000	-	69.70
XCF (Safe Mode)	0x400000	-	45.24
Partition 0	0x4000000	65,536	137.45
Partition 1	0x8000000	65,536	137.45
Partition 2	0xC000000	65,536	137.45
Partition 3	0x10000000	65,536	15.47
Partition 4	0x14000000	65,536	17.21
Partition 5	0x18000000	65,536	6.48
Partition 6	0x1C000000	65,536	137.45

5.4.2. Nominal Mode

The Nominal Mode is the main mode of operation of the prototype. In it, the five partitions described in section 5.1 run at a frequency of 0.5Hz: the three Safe Partitions (#NIR0, #NIR1 and #NIR2), the Health Monitoring partition and the partition used to implement the Voter, when it runs in software (this partition is used to implement a General Partition, when the Voter runs in the FPGA). In addition, during Nominal Mode execution, there is a redundant standby partition (#NIR3) that can replace one of the active Safe Partitions (#NIR2) if the Health Monitoring partition deems it appropriate and orders this replacement. All partitions are loaded in memory in the positions indicated in section 5.4.1 and the substitution between #NIR2 and #NIR3 is executed by changing the scheduling plan: plan 0 schedules #NIR2, while plan 1 schedules #NIR3. Only the HM partition has the privileges to execute the scheduling plan switch.

The Nominal Mode scheduling plans are shown in Tables 5.5 and 5.6.

Table 5.5: Prototype scheduling policy - Plan 0

Partition Number	Partition Name	Core 0		Core 1	
		Start	Duration	Start	Duration
0	NIR #0	0ms	500ms	-	-
1	NIR #1	-	-	0ms	500ms
2	NIR #2	500ms	500ms	-	-
3	Health Monitoring	1600ms	400ms	-	-
4	Voter/General	-	-	1000ms	400ms

For the sake of clarity, Tables 5.5 and 5.6 are represented graphically in Figures 5.4 and 5.5:

Table 5.6: Prototype scheduling policy - Plan 1

Partition Number	Partition Name	Core 0		Core 1	
		Start	Duration	Start	Duration
0	NIR #0	0ms	500ms	-	-
1	NIR #1	-	-	0ms	500ms
3	Health Monitoring	1600ms	400ms	-	-
4	Voter/General	-	-	1000ms	400ms
6	NIR #3	500ms	500ms	-	-

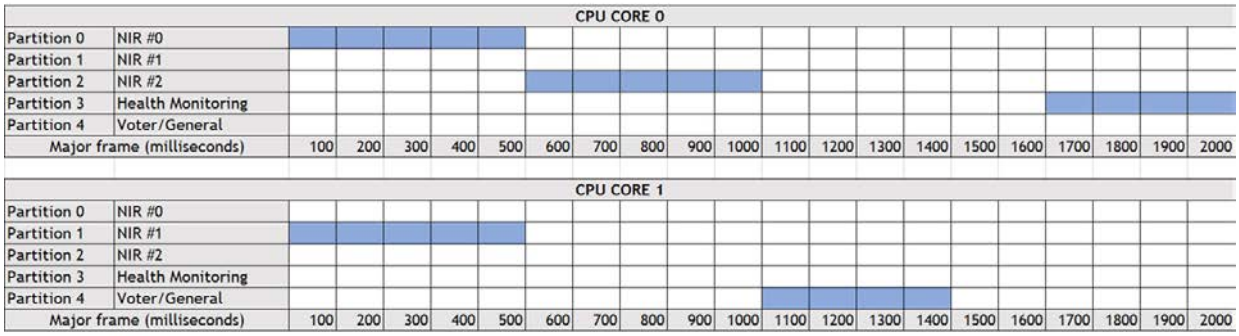

Figure 5.4: Scheduling policy for prototype partitions - Plan 0

5.4.3. Safe Mode

During the life cycle planning of a spacecraft, the possibility of the spacecraft entering a state often referred to as "safe mode" is considered, to protect the vehicle when an anomalous and potentially dangerous failure occurs to the system. In this mode of operation, faults are isolated to prevent propagation, communication with ground is established, and the vehicle is oriented in a power-positive and thermally stable configuration [178]. This mode is maintained indefinitely until operators take the necessary steps to restore the system remotely.

In the case of the prototype proposed in this chapter, it is not the specific actions performed by the Safe Mode that are of interest, but the system's ability to migrate from the Operational Mode to the Safe Mode, so that the partitions that were running in the Operational Mode stop running and consuming system resources, and one or more partitions that make up the Safe Mode start running. Therefore, the Safe Mode of the prototype will consist of a partition isolated from the rest, whose functionality is very simple: it will send a message to the console and will enter an infinite loop (in which it would wait for the reset from Ground in a real use case).

This Safe Mode has its own XtratuM Configuration File (XCF), in which the rest of the partitions (that do not belong to the Safe Mode) are not contemplated (therefore, no

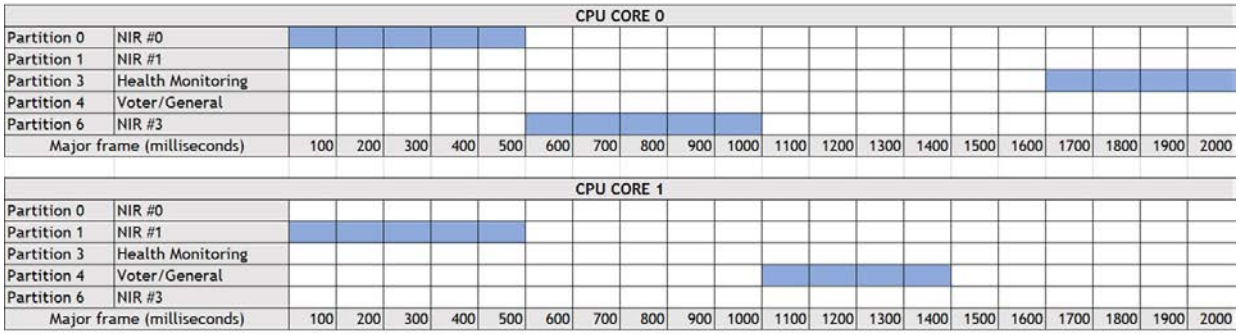

Figure 5.5: Scheduling policy for prototype partitions - Plan 1

resources are assigned to them) and in which there is only one scheduling plan that executes partition 5 cyclically with a frequency of 0.5Hz. To obtain even more assurance that Safe Mode will be executed correctly, even in the event that one of the processing cores is malfunctioning, the two available cores will execute the Safe Mode code sequentially.

Table 5.7: Prototype scheduling policy - Core 0

Partition Number	**Partition Name**	**Start**	**Duration**
Partition 5	Safe Mode	0s	1s

Table 5.8: Prototype scheduling policy - Core 1

Partition Number	**Partition Name**	**Start**	**Duration**
Partition 5	Safe Mode	1s	1s

Note that in any of the operational modes all the partitions and XCFs are loaded in memory, in the positions indicated in table 5.4, but depending on the XCF used, the hypervisor will be aware of some or others and will allocate resources to them according to what is established in the XCF. Once the system is in one of the operational modes, the only way to migrate to the other is to reset the system and configure the hypervisor with the XCF corresponding to the new operational mode.

5.5. Health Monitoring

The developed prototype includes a Health Monitoring partition that implements all the functionality described in section 4.4. A series of tests performed on this Health Monitoring partition are described below, in order to verify that it takes the appropriate measures in case of system failures. In section 5.5.1 we generate errors that affect the state of the hypervisor, in section 5.5.2 we generate errors that affect the state of one or more partitions and, finally, in section 5.5.3 we generate errors that affect the calculations performed by the Safe Partitions.

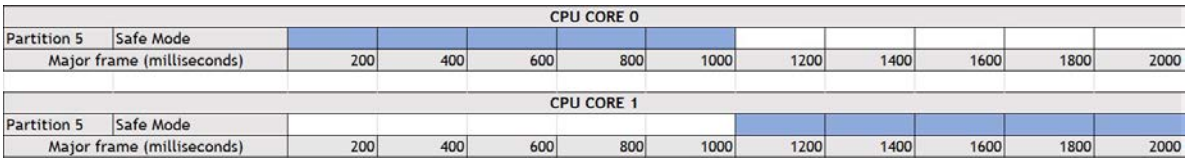

Figure 5.6: Scheduling policy for the Safe Mode

5.5.1. Hypervisor health check

Listing 5.1: Execution of the prototype and standard reset of the hypervisor

```
[P0] Initialized.
[P1] Initialized.
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Initialized.
[P2] Iteration completed.
[VOTER] Initialized.
[HM] Health Monitoring initialized.
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER] The calculations of P0, P1 and P2 agree.
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER] The calculations of P0, P1 and P2 agree.
[P1] Iteration completed.
[P0] Iteration completed.
[P2] Iteration completed.
[VOTER] The calculations of P0, P1 and P2 agree.
[HM] Hypervisor is going to be reset.
It has been reset 0 times before.
```

After restarting the hypervisor, the partitions run the same way again and the hypervisor restarts another two times. The console output is shown below when the hypervisor restarts for the third time, exceeding the maximum number of times allowed before switching to safe mode:

Listing 5.2: Execution of the prototype and launching of Safe Mode

```
[XNG-DBG:0] hypervisor reset
[P0] Initialized.
[P1] Initialized.
```

```
[P1] Iteration completed.
[P0] Iteration completed.
[P2] Initialized.
[P2] Iteration completed.
[VOTER] Initialized.
[HM] Health Monitoring initialized.
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER] The calculations of P0, P1 and P2 agree.
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER] The calculations of P0, P1 and P2 agree.
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER] The calculations of P0, P1 and P2 agree.
[HM] Hypervisor is going to be reset.
It has been reset 2 times before.
[HM] Launching safe mode...
```

Having ordered a hypervisor reset three times, the HM partition assumes that something is not working properly, so it launches safe mode. On a space system, typically, safe mode disables all non-essential system functionality and runs only essential functions such as Attitude and Orbit Control, thermal control, or ground communication, as ground operators will likely need to take action to recover the system.

Listing 5.3: Safe Mode execution

```
[XNG-DBG:0] hypervisor reset
[SAFE MODE] Non-essential functions are shut down.
Waiting to be reset from ground...
[SAFE MODE] Non-essential functions are shut down.
Waiting to be reset from ground...
[SAFE MODE] Non-essential functions are shut down.
Waiting to be reset from ground...
```

XNG provides partitions with access to two different virtual clocks:

- The **virtual system clock** is the representation of a physical real-time clock. It is a monotonically increasing clock that allows any partition to query the time elapsed since the hypervisor was last reset.

- The **virtual execution clock** is a monotonically increasing clock local to each virtual CPU, which returns the time that this virtual CPU has run since the last partition reset. This clock also takes into account the time used by the hypervisor for its own tasks (e.g. service dispatching, IRQ handling).

Using the virtual system clock, we can measure the time it takes XNG to reset the hypervisor in the use cases used to test the hypervisor health check process. Note that in this use case two different resets occur: in one case, a normal reset is performed (which restores the system to its nominal operation); in the other, a reset is performed that launches Safe Mode. Reset times have been measured for 10 standard resets and 10 resets that launch Safe Mode. These figures, shown in Table 5.9, correspond to the time elapsed between the moment when the hypervisor is reset by order of the HM partition and the moment when, once the system has been reset, the first partition (according to the scheduling plan) starts running.

Table 5.9: XNG reset time

Reset number	**Time required to perform reset (standard) in us**	**Time required to perform reset (Safe Mode) in us**
1	20315	5006
2	20363	5005
3	20307	5006
4	20368	5011
5	20317	5006
6	20322	5005
7	20318	5011
8	20314	5010
9	20364	5007
10	20305	5011
Mean time (us)	**20,329.3**	**5,007.8**

As can be seen, there is a clear difference between the time it takes for the hypervisor to reset when loading the nominal configuration and when loading Safe Mode. This difference is not surprising, since both configurations are very different in terms of memory occupation and resource consumption (the nominal configuration not only loads more partitions, but each of these is heavier than the partition running Safe Mode). However, in either case, the time it takes for the hypervisor to reset itself is much less than the major frame of the scheduler of the use case we are applying. If we assume, that the redundancy scheme proposed in this thesis is intended for systems with low frequency cyclic behaviour (0.5-20Hz), we can derive that the impact of a system reset is minimal in terms of execution time.

5.5.2. Partitions health check

Before injecting faults continuously into the system, we test the mechanism for detecting and correcting partition health faults. Initially, the system execution starts normally. With debug traces enabled, the console output is shown below:

Listing 5.4: Normal execution of the system

```
[XNG-DBG:0] hypervisor reset
[P0] Initialized.
[P1] Initialized.
[P1] Iteration completed.
[P0] Iteration completed.
[P2] Initialized.
[P2] Iteration completed.
[VOTER] Initialized.
[VOTER] The calculations of P0, P1 and P2 agree.
[HM] Initialized.
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER] The calculations of P0, P1 and P2 agree.
[P1] Iteration completed.
[P0] Iteration completed.
[P2] Iteration completed.
[VOTER] The calculations of P0, P1 and P2 agree.
```

In one of the iterations, the HM itself will inject a fault that will cause the suspension of partition P2. In the next iteration, therefore, the HM will detect in the usual check that P2 is not in the expected state. The voter realizes that it is not receiving inputs from P2 on the next iteration, so it automatically goes into Degraded Mode. Subsequently, the HM will reset P2 and the system will resume normal activity. The console output is shown below:

Listing 5.5: Reset of P2 partition

```
[VOTER] The calculations of P0, P1 and P2 agree.
[HM] Injecting fault in P2...
[P0] Iteration completed.
[P1] Iteration completed.
[VOTER] P2 not active: entering Degraded Mode.
[VOTER] The calculations of P0 and P1 agree.
[HM] P2 is not in the expected state.
```

```
[HM] Resetting partition P2.
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Initialized.
[P2] Iteration completed.
[VOTER] The calculations of P0, P1 and P2 agree.
[P1] Iteration completed.
[P0] Iteration completed.
[P2] Iteration completed.
[VOTER] The calculations of P0, P1 and P2 agree.
```

If we inject that error cyclically, the HM will eventually make the decision to halt partition P2 (the prototype configuration at the time of this test states that each partition can be reset a maximum of 2 times before assuming it is severely malfunctioning). After halting P2, the HM checks if there is any partition in cold redundancy (i.e., loaded in memory but not active) that can replace P2. In this case there is: partition P6. Therefore, the HM performs a scheduling plan change: it stops scheduling P2 and proceeds to schedule P6 instead. After this, the execution of the system continues normally, as indicated by the console output if we activate the debug traces:

Listing 5.6: Halt and substitution of P2

```
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER] The calculations of P0, P1 and P2 agree.
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER The calculations of P0, P1 and P2 agree.
[HM] Injecting fault in P2...
[P0] Iteration completed.
[P1] Iteration completed.
[VOTER] P2 not active: entering Degraded Mode.
[VOTER] The calculations of P0 and P1 agree.
[HM] P2 is not in the expected state.
[HM] P2 has been reset too many times. Halting partition P2.
[HM] P6 is available to substitute P2. Changing sched. plan.
[P0] Iteration completed.
[P1] Iteration completed.
[P6] Init.
[P6] Iteration completed.
[VOTER] The calculations of P0, P1 and P6 agree.
```

```
[P0] Iteration completed.
[P1] Iteration completed.
[P6] Iteration completed.
[VOTER] The calculations of P0, P1 and P6 agree.
```

After verifying that the partition health check mechanism is working and taking corresponding measures, a fault injection campaign of two types is performed:

1. **Type I Partition Health Failure**: using direct *hypercalls* to the hypervisor, a partition is suspended or shut down.
2. **Type II Partition Health Failure**: a trap is implemented on each partition, which is executed randomly, in which the partition is locked in an infinite loop. This trap is intended to simulate a failure in which the partition is not suspended or terminated in the normal way, but is blocked and stops responding to inputs.

As expected, Type I failures are detected 100% of the time by the partitions health check process by the HM. Type II failures are not detected by the standard partition health check, as the partition may freeze in a normal operational state. If communication channels are established between each partition and the HM and the *handshaking* mechanism mentioned in section 4.4.2 is implemented, Type II failures are also detected, so **the HM is able to detect 100% of partition health failures**.

5.5.3. Partitions results check process

Finally, before injecting frequent failures in the partition calculations to check that the system recovers correctly from them, we check that the failure detection mechanism works. In a first scenario, partition P6, which is the backup partition (see section 5.4 for details), is not loaded into memory. P0, P1 and P2 run continuously calculating the same results. After a certain time instant, we start forcing P2 to compute wrong results. At first, the system will simply detect these failures and will not react in any special way. After failing three times, the Health Monitoring partition will reset the P2 partition.

Listing 5.7: P2 fault inyection and reset

```
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER] The calculations of P0, P1 and P2 agree.
[HM] Injecting fault in P2...
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER] P2 has failed in its calculations.
```

```
[HM] Injecting fault in P2...
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER] P2 has failed in its calculations.
[HM] Injecting fault in P2...
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER] P2 has failed in its calculations.
[HM] P2 has failed too many times.
[HM] Resetting P2...
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
```

If we keep injecting faults into Partition P2, the HM partition will continue to reset it until, after 2 resets (the prototype configuration at the time of this test states that each partition can be reset a maximum of 2 times before assuming it is severely malfunctioning), the HM partition halts P2 partition permanently. As in this example there is no backup partition, the voter detects that there are only two active partitions and activates the Degraded Mode.

Listing 5.8: P2 fault inyection and halt

```
[HM] Injecting fault in P2...
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER] P2 has failed in its calculations.
[HM] P2 has failed too many times.
[HM] P2 has been reset too many times.
[HM] Halting P2...
[P0] Iteration completed.
[P1] Iteration completed.
[VOTER] Only two partitions are active.
Entering Degraded Mode...
[VOTER] The calculations of P0 and P1 agree.
[P0] Iteration completed.
[P1] Iteration completed.
[VOTER] The calculations of P0 and P1 agree.
```

In a second scenario, we do load backup partition (P6) in memory. In this case, after the first failure of P2, the system detects that there is a partition (P6) with fewer failures, so

it proceeds to replace P2 with P6. If we subsequently inject two consecutive failures in P6, the failure counter of P6 will again exceed that of P2 and the HM partition will perform the replacement again. Although there may be several substitutions at first, it takes a short time for the system to find an equilibrium where the three healthiest partitions are continuously running.

Listing 5.9: P2 fault inyection and substitution

```
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER] The calculations of P0, P1 and P2 agree.
[HM] Injecting fault in P2...
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER] P2 has failed in its calculations.
[HM] There is a healthier backup partition.
Changing scheduling plan...
[P0] Iteration completed.
[P1] Iteration completed.
[P6] Iteration completed.
[VOTER] The calculations of P0, P1 and P6 agree.
[HM] Injecting fault in P6...
[P0] Iteration completed.
[P1] Iteration completed.
[P6] Iteration completed.
[VOTER] P6 has failed in its calculations.
[HM] Injecting fault in P6...
[P0] Iteration completed.
[P1] Iteration completed.
[P6] Iteration completed.
[VOTER] P6 has failed in its calculations.
[HM] There is a healthier backup partition.
Changing scheduling plan...
[P0] Iteration completed.
[P1] Iteration completed.
[P2] Iteration completed.
[VOTER] The calculations of P0, P1 and P2 agree.
```

5.6. Voter

For the prototype, a version of the voter has been developed to test all the fault tolerance mechanism proposed in chapter 4 and to take metrics on the performance of the voter. In addition, both a software version and a hardware version (FPGA electronic design) of the voter have been developed in order to compare them and establish which version is more suitable for the fault tolerance mechanism. The following sections describe these two versions of the voter.

5.6.1. Software Voter

Since the prototype is limited (e.g., there is only one partition in cold redundancy to replace an active partition that goes down), a **limited version of the voter** has been developed in a software partition. Although not complete, this version allows to test the functionality discussed in chapter 4 and allows to measure how long it takes to run the voter as a software process on the hardware chosen for the prototype.

This *software version* of the voter was developed before the *FPGA version*, in order to start testing the fault tolerance mechanism. In these tests, the voter runs as an additional partition scheduled by the hypervisor in the manner explained in section 5.4.2. In this way, the voter communicates with the HM and with the partitions that vote using the inter-partition communication mechanisms provided by the hypervisor. When the voter is implemented on the FPGA, the resources used by the former voter partition can be repurposed for other functionality (e.g., to implement a General Partition).

Figure 5.7 graphically represents the simplified version of the prototype voter:

As can be seen, in this simplified version partitions P0 and P1 are always going to be active (note that this does not mean that they compute correct results, only that they will keep running during the whole testing time). Partition P2 is the one that can be halted by the HM, and replaced by partition P6 in case the latter is available. Therefore, to check the operating mode in which it is, the first thing the voter will do after reading the votes of P0 and P1 is to check if it is receiving data through the communication channel with P2. If not, it will try to read the results from P6, since it is possible that the HM has changed the active partition due to an error in P2. If it does not find results on this communication channel either, it will proceed to activate the Degraded Mode. Both the Normal Mode and the Degraded Mode work exactly as described in Section 4.3.1.

5.6.2. FPGA Voter

The voter prototype proposed in Section 5.6.1 serves to test the validity and effectiveness in terms of fault detection and correction of the architecture proposed in Chapter 4, as it covers the functionality needed to perform the fault injection tests proposed in Chapter 6.

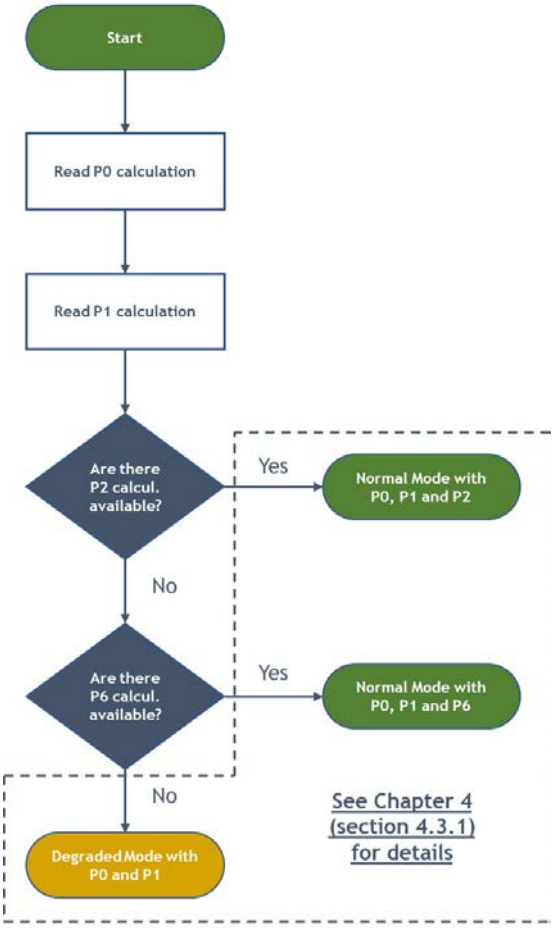

Figure 5.7: Voter prototype simplification

However, there are a number of potential advantages if this voting logic is migrated to IP blocks implemented on the FPGA. The main ones are:

- **Improve voter execution time** by **parallelizing** voter instructions and **pipelining**.
- **Robust the voting process** by parallel execution (without sacrificing execution time) of **several instances** of the voter.
- **Relieve computational load on the CPU** to execute other software partitions.

Although the FPGA digital design does not fall within our field of expertise, the voter is a relatively simple block, so we have proposed a first approach to the FPGA voter prototype. For this prototype we have implemented the *Normal Mode* of the voter, since the other modes are simplifications of it. Figure 5.8 represents the RTL design of the resulting digital circuit:

As can be seen, the design has three inputs (the values calculated by each of the three voting partitions) and three outputs (the final valid value, the reliability of this value and the ID of the failed partition, if any). In addition, there are inputs and outputs necessary

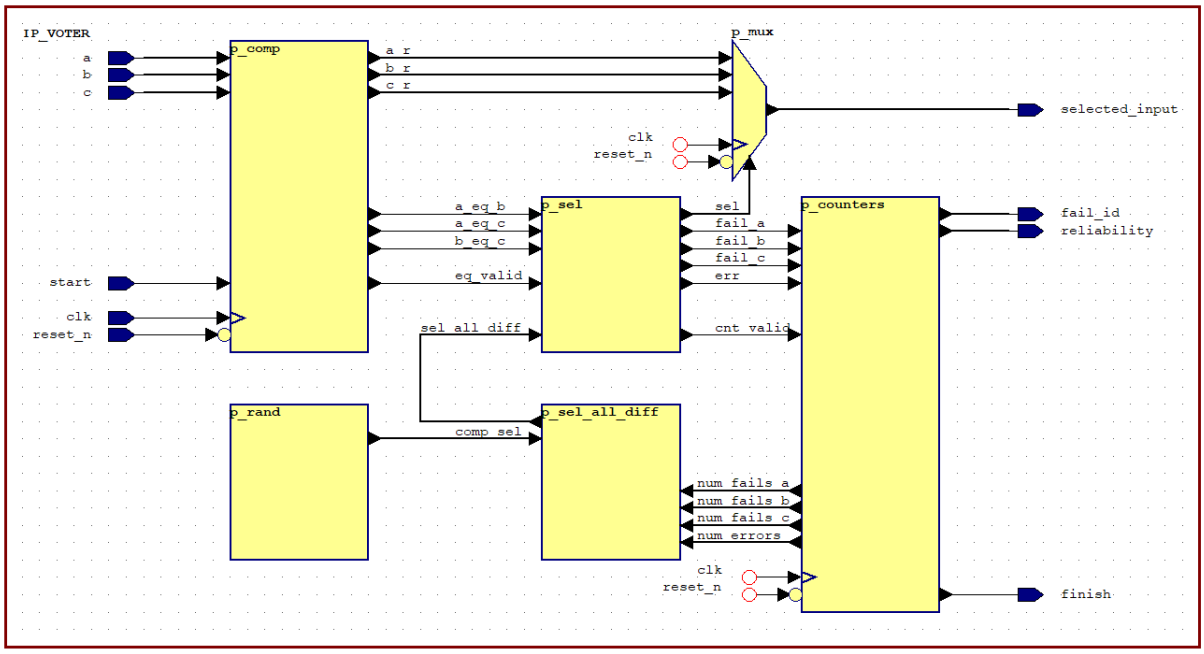

Figure 5.8: RTL description of the voter prototype

for the correct signalling and operation of the design (*start, clk, reset_n, finish*). The design is divided into several blocks:

- The ***p_comp*** block records the inputs and makes a comparison between them to check if they match each other and generate signals according to this comparison.
- The ***p_sel*** block reads the outgoing signals from the *p_comp* block and decides if any partition has failed. It also generates the signal that will allow deciding the correct result.
- The ***p_mux*** block uses the decision signal generated in the *p_sel* block to define the final output of the system.
- The ***p_counters*** block keeps the internal count of the failures of each partition, identifies both the faulty partition (if any) and the reliability of the final decision, and signals the end of a voting iteration.
- The ***p_sel_all_diff*** block is responsible for performing tie-breaking in case the three outputs are different. The tie-breaking is based on the failure history of each partition and, ultimately (in case of a total tie), on random decisions.
- The ***p_rand*** block generates pseudo-random numbers for tie-breaking in case of a total tie (all inputs differ and there is a tie in the failure history of at least two partitions).

5.7. Summary

In this chapter, we have explained the development of the prototype that demonstrates the capabilities of the solution described in chapter 4. We have proposed an architecture, which is a particularisation of the architecture of the solution, and we have justified the choice of a specific hardware and hypervisor (with the help of the survey carried out in chapter 2). We have explained the distribution in memory of each of the components of the architecture and the prototype's modes of operation. Finally, for the Voter, we have proposed and developed two different versions: one running in a software partition, like the rest of the components of the architecture, and the other running on the FPGA.

Chapter 6 will test the prototype. First, by unit testing the error detection and correction capabilities of the Health Monitoring partition, and then by performing a fault injection campaign to test the fault detection and correction rate of the solution in different scenarios. We perform two additional tests: one that measures the footprint of the hypervisor and another one that compares the performance and overhead of the two versions of the Voter.

6. EVALUATION

This chapter contains the evaluation of the proposed solution and, in particular, of the prototype developed to implement it. The chapter is divided into four main sections:

- Section 6.1 is dedicated to measure the overhead involved in using the XtratuM hypervisor with respect to not be using it. Since virtualization is one of the main added values of the proposed solution, and since there is practically no evidence on the overhead of virtualization in general (and XtratuM in particular) on embedded systems, it is convenient to perform a couple of experiments to obtain an estimation.
- Section 6.2 is dedicated to set up and run a fault injection campaign in which, according to different statistical distributions, faults are randomly generated in the partition calculations. With the results, an estimation of the fault detection and correction capability of the proposed solution in different scenarios can be obtained.
- Section 6.3 is dedicated to compare the performances of the software prototype and the hardware prototype of the Voter, as well as to draw conclusions on the convenience of using one or the other.
- Section 6.4 briefly summarises the chapter and introduces the contents of the following chapter.

6.1. XtratuM Performance Measurement

XNG is a paravirtualization hypervisor (see section 2.1.2 for more details), so it is expected that its use does not involve a large overhead. However, there is no actual published information, neither in the developer's sources nor in research papers, about the CPU overhead involved in using XNG on any platform. As part of the work performed and reflected in this thesis, this overhead has been measured using a couple of different benchmarks and under different conditions, in order to obtain an estimate as representative as possible.

- First, the Dhrystone benchmark on the Linux operating system was used to measure the difference between the number of instructions required on one of the MicroZed board cores when XNG virtualizes the hardware and when running without hypervisor. For this test, a single processor core is used since, currently, XNG does not support Symmetric Multi-Processing mode when running Linux as a guest OS on an ARM platform.

- Secondly, the NIR HAWAII-2RG benchmark is used, to obtain an estimate in a scenario closer to a real case in which the MIA architecture could run. In addition, the most recent version of the hypervisor (which allows the use of the two processor cores of the MicroZed) is used in this test, since the benchmark runs without the need of an operating system and, therefore, we are not bound to the limitations imposed by the Linux BSP.

Some details of the HAWAII-2RG NIR benchmark have been explained in Chapter 5, section 5.3, since it is the same benchmark used to implement the critical partitions of the prototype. A brief description of the Dhrystone benchmark is given below. Subsequently, the results obtained in the performance tests will be presented.

6.1.1. Dhrystone Benchmark

Dhrystone is a synthetic benchmark created by Reinhold Weicker in 1984 [179]. This means that, like other synthetic benchmarks, it is a simple program that mimics the processor usage of other software programs to measure the performance of a machine. Instead of using MIPS (Million Instructions Per Second), which may not be meaningful when comparing different instruction sets (such as RISC or CISC), Dhrystone uses the unit DMIPS (Dhrystone MIPS) to reflect the average value of the number of instructions executed per second.

Although it has been a widely used benchmark for decades, Dhrystone has a number of limitations that make it desirable to also use other benchmarks to complete the measurement of a machine's performance [180]. For example, it places too much weight on integer computation and the small size of its code means that it can sometimes be absorbed by the instruction cache of a modern CPU, altering the measurement. Although many of its limitations are alleviated due to the nature of our measurements (we want to measure the performance of the same physical platform with and without hypervisor, not compare two completely different platforms), we will complete the measurement using a benchmark more representative of a space application.

6.1.2. Performance Results

Once the benchmarks to be used have been explained, the conditions under which they are run and the results obtained by comparing the execution with and without hypervisor are shown below.

Drhystone Results

The Dhrystone benchmark has been used to measure the difference between the number of instructions that can be performed by a processing core of the MicroZed card (based

on the Zynq-7000 SoC [181]) when XNG virtualizes the hardware and when running without hypervisor. A single processor core is used because, currently, XNG does not support Symmetric Multi-Processing mode when running Linux as a guest OS on an ARM platform. The same benchmark is executed twice under the following conditions:

- **Benchmark**: Dhrystone.
- **Dhrystone version**: 2.1 (Language: C).
- **Number of iterations through the benchmark main loop**: 100,000,000.
- **Target platform**: MicroZed (a single ARM Cortex-A9 core is used).
- **Operating System**: Linux.
- **SMP**: Disabled.
- **Linux kernel version**: 5.4.

In addition to using the same kernel, the file system of the Linux image on which the benchmark runs is exactly the same, to ensure that both runs run in exactly the same environment, with the only exception of the hypervisor.

The output from the benchmark is expressed in number of Dhrystones per second (the number of iterations of the main code loop per second). Using XNG to virtualize the hardware, the processor core was able to run **1,376,841.5** Dhrystones per second, compared to the **1,427,551.8** Dhrystones per second it runs without a hypervisor. From this it can be concluded that, for the operations proposed by this benchmark and under the conditions described above, **XNG causes an overhead of 3.683% in the processor**.

Table 6.1: XNG overhead using the Drhystone benchmark

Dhrystsones per second with XNG	Dhrystone per second without hypervisor	XNG overhead
1,376,841.5	1,427,551.8	3.683%

NIR HAWAII-2RG Results

The HAWAII-2RG NIR benchmark is a bare-metal benchmark, i.e. it does not run on any operating system. Therefore, since we are not subject to the restrictions imposed by the Linux BSP, we can use the most modern version of XNG for Zynq-7000 (XNG SMP). As mentioned above, in addition, in order to make the XNG footprint measurement as accurate as possible, the necessary modifications have been made to the code to run the benchmark with a fixed and deterministic set of input data. The benchmark is run with the following configuration parameters:

- Fixed-Point algorithms.
- Version: 20120629.
- Image Size: 1024x1024.
- Number of groups per exposure: 5*.
- Number of frames per group: 1.
- Number of discard frames between groups: 0.
- Time between groups: 18 seconds.
- Offset due to UINT.
- Threshold for cosmic ray: 6.

**This determines the number of times that the main loop of the benchmark will be executed.*

In addition, compiler optimizations (-O0) are disabled and the following compiler options are used:

Listing 6.1: Compilation options for measurement on the NIR benchmark HAWAII-2RG

```
-mfloat-abi=hard -mthumb -fno-builtin
-fno-short-enums -ffreestanding
```

The first measurement of the difference between using the hypervisor and not using it to run the benchmark is surprisingly high:

Table 6.2: XNG overhead measured with the NIR benchmark HAWAII-2RG

Execution time w/o hypervisor	**Execution time using XNG**	**XNG overhead**
33,260,972 us	46,339,466 us	+39.32%

Since this result is not in agreement with either the overhead calculated with the Dhrystone benchmark or with what would be expected from a paravirtualizing hypervisor, we decided to take further steps to help determine what might account for this large difference. First, the main loop of the benchmark, which is described in section 5.3 and depicted in Figure 5.3, is simplified. To do this, we first run the loop only executing the first functionality (saturation detection). After measuring the overhead in this case, the loop is run again executing also the next function (co-adding), and functions are added in consecutive runs of the benchmark. The objective of this test is to check if the overhead produced by XNG is constant throughout the algorithm or if it is produced by some specific functionality. The results obtained are shown in Table 6.3.

It can be clearly observed that the overhead produced by the hypervisor remains at the expected levels (below 5%) during almost the entire execution of the algorithm. It is by

Table 6.3: XNG overhead along the different steps of the algorithm

	Accumulated execution time w/o hypervisor (us)	**Accumulated execution time with XNG (us)**	**XNG overhead (accumulated)**
detectSaturarion(...)	163,045	166,350	+2.02%
subtractSuperBias(...)	433,732	405,415	-6.53%
nonLinearityCorrectionPolynomial(...)	1,210,229	1,225,404	+1.25%
subtractReferencePixelTopBottom(...)	1,413,121	1,472,817	+4.22%
subtractReferencePixelSides(...)	1,632,519	1,662,486	+1.84%
detectCosmicRay(...)	20,585,551	21,510,220	+4.49%
progressiveLinearLeastSquaresFit(...)	33,125,011	46,376,317	+40%
calculateFinalSignalFrame(...)	33,180,992	46,568,883	40,34%
Total time	**33,180,992**	**46,568,883**	**40,34%**

running the progressiveLinearLeastSquaresFit(...) function that the overhead shoots up to 40%.

The first suspicion is that the type of operations performed in this function, because of the interactions they may require between the hypervisor and the hardware, are particularly costly to virtualize. However, as can be verified by a quick inspection of the code, there do not seem to be any operations in this function that do not appear in other previous functions. In any case, to rule out that this is the problem, a new test is performed in which each of the testbench functions is executed in isolation 20 times, and the execution time required for this is checked using XNG and without using it. The results obtained are shown in Table 6.4.

Table 6.4: XNG overhead for each of the steps of the algorithm measured in isolation

Functionality	**Execution time w/o hypervisor (us) – 20 iterations**	**Execution time with XNG (us) – 20 iterations**	**XNG overhead**
detectSaturarion(...)	672,975	740,683	+10.06%
substractSuperBias(...)	1,082,783	1,034,087	-4.50%
nonLinearityCorrectionPolynomial(...)	2,971,145	3,225,484	+8.56%
substractReferencePixelTopBottom(...)	945,749	1,142,183	+20.77%
substractReferencePixelSides(...)	893,973	851,632	-4.74%
detectCosmicRays(...)	110,032,418	114,140,811	+3.73%
progressiveLinearLeastSquaresFit(...)	2,270,418	2,447,094	+7.78%
calculateFinalSignalFrame(...)	68,394	74,763	+9.31%
Total time	**118,937,854**	**123,656,737**	**+3.97%**

When running the functions of the algorithm in isolation, even when they are run repeatedly (note how computationally demanding the tests are for some functions), the overhead does not skyrocket as much. In fact, if we add up the time required to run all these tests (and thus run the entire algorithm), the overhead of XNG remains at the expected levels. However, if we decide to run the entire loop 20 times consecutively (instead of running the functions in isolation, as we have done previously), the overhead rises again, as shown in Table 6.5.

Table 6.5: XNG overhead for 20 iterations of the algorithm

Execution time w/o hypervisor – 20 iterations	**Execution time using XNG – 20 iterations**	**XNG overhead – 20 iterations**
190,460,151 us	277,853,467 us	+45.58%

What seems to be clear from the results is that XNG has a low overhead similar to that measured in section 6.1.2 for most cases (<5%). However, on some occasions, when running several different functions consecutively, it seems to saturate and the overhead increases significantly. This could be due to the hypervisor, during its normal operation when interacting with the hardware and partitions, constantly occupying a portion of the processor's instruction cache, so that it saturates earlier when the partition is performing tasks. The information gathered from these tests will be made available to fentISS, the developer of XNG, for consideration in future updates of the hypervisor. Although it is not the direct objective of this research, the results obtained in this test are interesting and raise new lines of future work, as will be discussed in Chapter 7.

6.2. Fault Injection

Once we have verified the correct functioning of the testing process of the results that calculate the redundant partitions, we proceed to a fault injection campaign to evaluate the performance of the fault tolerance mechanism. First, we will analyse the ability to detect and correct Single Event Upsets, the most common error to which electronic components that are in contact with energetic particles, such as space radiation with which aircraft of various types have to deal with, are exposed. Subsequently, the ability to detect and correct an extremely rare but possible phenomenon will be analysed: the simultaneous occurrence, in different virtual machines, of two Single Event Upsets.

6.2.1. SEU Injection

During the fault injection campaign conducted to test the prototype described in Chapter 5, the effect of an SEU affecting the computations of one of the virtual machines running one of the safety-critical applications is simulated. In the prototype, the safety-critical applications are redundant instances of a space benchmark that performs computations based on images detected by an infrared camera. To simulate the effect of SEUs, at each iteration and in each virtual machine there is a probability that a bit of the variable that hosts the final computation of the algorithm will be flipped.

Table 6.6: SEU Correction Capability

# of SEU Occurrences	Correctly Masked	Incorrectly Masked	Correction (%)
214	214	0	**100%**

As expected and as reflected in the table 6.6, which collects the simulation data, the algorithm is able to detect and correct all SEUs. In the simulation it is established that there is a 25% probability of SEUs in each iteration for each virtual machine. The algorithm runs at 0.5Hz for 800 seconds. Note that, in order to obtain a large number of SEUs to corroborate the expected result, the probability of SEU in the simulation is much higher than the probability of SEU even in adverse situations of high radiation.

6.2.2. Double SEU Occurrences

In the scheme we propose, there is an extremely anomalous situation, which is that two SEUs occur in different virtual machines at the same time (during the same iteration; in the prototype, which runs applications at 0.5Hz, this means that two SEUs occur in different virtual machines in a given slot of 2 seconds). To outline how difficult it is for this to happen, we turn below to some scientific papers that study the rate at which SEUs occur in different systems. Obviously, this depends on several factors, among which we can list the electronic device that undergoes them, how well it is shielded from radiation or the environment in which it operates.

Taber and Normand analyse data from military and experimental flights in search of SEUs caused by energetic neutrons created by cosmic rays in the atmosphere [138]. The approximate altitude of these flights was 30,000 feet (over 9km) and the electronics studied were unshielded SRAM memories. After 118 total flights and 783 cumulative flight hours, 136 upsets due to cosmic radiation were observed. This implies that, under these conditions, it would take 5.76 hours on average to observe a single upset. In the 400-second simulation we ran earlier, there would be less than a 2% chance of observing a single upset.

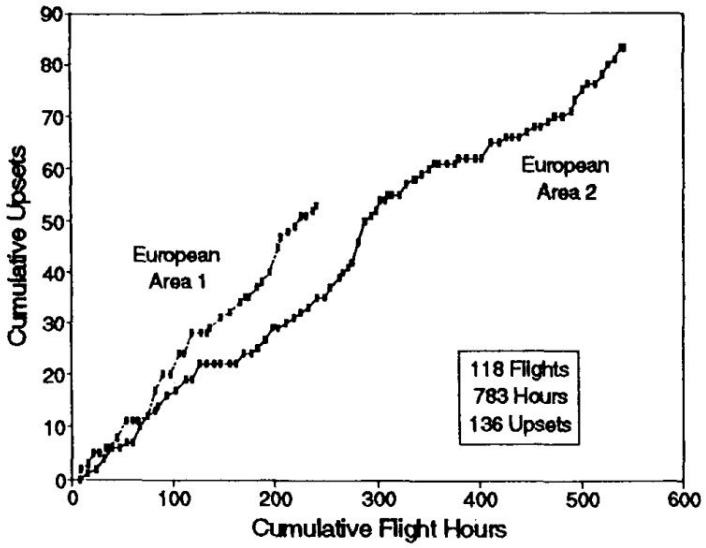

Figure 6.1: Cumulative avionics upsets (image from Taber and Normand [138])

Blake and Mandel monitored for 2 years a satellite subsystem, operating in a low polar

orbit, containing 384 Harris HM-6508 RAMs [182]. A total of 72 SEUs were observed during the satellite's 731-day flight. In their paper they analyse that, on average, one SEU is expected to occur every 12.6 days, and two SEUs are expected to occur on the same day once every two years. The probability of two independent events occurring within two seconds is therefore absurdly low.

Other more modern papers analyse the rate of SEUs in more depth according to the particles that cause them and the devices that suffer them, such as Kuznetsov's paper [183]. The conclusion is the same: the occurrence of SEUs per second in the scenarios it considers is very low. Even in very aggressive scenarios, such as the one studied by Campbell et al. [184], which analyse SEUs in particularly harsh orbits from a radiation point of view and focus on certain days when there was a solar flare, the rate at which SEUs occur is much lower than the one we use in the simulations of the next experiments.

Double SEU randomly generated

From the above, it does not make much practical sense to analyse the effect of two or more independent and simultaneous SEUs. However, to test the theory of what would happen in these cases, two scenarios are presented that force these errors to occur with some frequency:

- **Scenario 1**: each of the three active virtual machines in the prototype have a 20% chance of suffering an SEU in each iteration.
- **Scenario 2**: each of the three active virtual machines in the prototype has a 33.33% probability of suffering a SEU in each iteration.

Data on the number of independent double SEUs suffered and the system's ability to correct them are collected in Table 6.7.

Table 6.7: Correction of double SEU occurrences randomly generated

	Double SEU Occurrences	Correctly Masked	Incorrectly Masked	Correction (%)
Scenario 1	39	16	23	41%
Scenario 2	95	29	64	30%
Total	**132**	**45**	**87**	**34%**

Note that the results are as expected: being random (the SEUs occur independently in any of the three virtual machines), the correction rate of the fault tolerance mechanism tends to 33.33% since, in case the three virtual machines yield different results, it ends up deciding between them randomly.

Double SEU due to Hardware Damage

As mentioned above, SEUs generally involve a transient error and do not cause permanent damage to the devices that suffer from it. However, under the right circumstances (both because of the electronic component or circuit and the properties of the particle interacting with it), a "parasitic" thyristor inherent in CMOS designs can be activated and produce an apparent short circuit in a transistor that is often referred to as a latchup [185] and sometimes destroys the device due to overcurrent. There are other single event effects that can cause what are called *hard errors*, i.e. permanent damage to electronic devices, such as snapbacks [186], burnouts or gate ruptures [187].

It makes much more sense to consider a scenario in which the resources used by one of the virtual machines have been damaged by one of these phenomena, and in which it would be more likely to encounter double SEUs. In these scenarios, one of the virtual machines (actually, one of the resources used by this virtual machine) would have been permanently damaged. If this error causes the virtual machine to start failing permanently going forward, the HM would make the decision to either halt it and replace it, if there is any partition in cold redundancy, or halt it and set the voter to degraded mode, if there is no other partition that can replace it. However, there is also the possibility that, due to permanent damage to the virtual machine, its probability of failure in the future increases. In these cases, the added feature in this work of keeping count of the failures of a virtual machine to assess its reliability greatly increases the voter's accuracy with respect to systems that do not have this feature. Over several iterations, the voter would assume that the virtual machine with the highest probability of failure is the least reliable so, in case of a triple tie (a double SEU causes the virtual machines to yield different results), it will assume the failure of the virtual machine that has been damaged. Since it has a higher probability of failure, the voter will usually get it right (it will be wrong only when the *healthy* virtual machines fail, an event which, as we have seen, is extremely rare).

Again, in the following experiments we have greatly exaggerated the probability of SEUs in each virtual machine, especially in those that have not been damaged by a latchout or some similar phenomenon. This is in order to force the occurrence of double SEUs and to be able to test the algorithm in the worst and most unlikely situations.

In the following experiment, we consider a scenario in which one of the virtual machines (P2) has been permanently damaged and its failure probability is high (it fails in one out of three iterations). The other two virtual machines (P0 and P1) maintain their *low* (albeit exaggerated for the experiment) probability of failure (failure in one out of 10 iterations). Table 6.8 shows the number of double SEUs originated under these conditions and the correction capacity of the mechanism for this type of errors. Although the table does not reflect it, since all these cases have been combined in table 6.6, the correction capacity of SEUs in this scenario is still 100%.

It can be deduced both from the data and from the debug traces of the program that the voter immediately detects that the P2 virtual machine is failing more than the others, so

Table 6.8: Correction of double SEU occurrences when one VM has a higher failure probability due to HW damage

Simulation time (s)	# of double SEU Occurrences	Correctly Masked	Incorrectly Masked	Correction (%)
200	16	6	10	**37,5%**
400	34	14	20	**41,18%**
600	48	22	26	**45,83%**
800	69	34	35	**49,28%**

that its reliability level drops considerably with respect to the other two. In this situation, in case of double SEU (and therefore discrepancy between the results returned by each virtual machine), the voter will tend to randomly choose (since the failure probabilities of the two healthy virtual machines, P0 and P1, are equal) the other malfunctioning virtual machine between P0 and P1. This makes that, as the simulation gets going, the correction capability of double SEUs tends to 50%.

We can also consider a scenario in which two virtual machines have been permanently damaged and, therefore, their probability of failure increases (failure in one out of three iterations). This situation can occur, for example, if two of the virtual machines share a processing core that has been damaged. The other virtual machine, which, for safety reasons, uses the other processor core, maintains its (albeit exaggerated for the experiment) *low* probability of failure. We propose two different scenarios: one in which the healthy virtual machine has in each iteration a 10% probability of failure and another in which it has a 5% probability of failure. The data from both experiments are reflected in tables 6.9 and 6.10, respectively.

Table 6.9: Correction of double SEU occurrences when two VMs have a higher failure probability due to HW damage - Scenario 1

Simulation time (s)	# of double SEU Occurrences	Correctly Masked	Incorrectly Masked	Correction (%)
200	16	11	5	**68,75%**
400	30	19	11	**63,33%**
600	48	31	17	**64,58%**
800	63	44	19	**69,84%**

We can analyse that the data makes sense: as single SEUs occur, the corrupted virtual machines lose credibility in the eyes of the voter. Eventually, in case of a double SEU (and therefore, a triple tie between the results that the virtual machines produce), the voter will always decide that the two damaged virtual machines are wrong. This will cause the voting to fail only when the healthy partition has failed: i.e., the correction capability in this case depends directly on the failure probability of the *healthy* machine; the less the *healthy* machine fails, the less the mechanism will fail to detect double SEUs. This explains the difference in correction percentage between scenario 1 and scenario 2 of the

Table 6.10: Correction of double SEU occurrences when two VMs have a higher failure probability due to HW damage - Scenario 2

Simulation time (s)	# of double SEU Occurrences	Correctly Masked	Incorrectly Masked	Correction (%)
200	14	11	3	**78,57%**
400	26	20	6	**76,92%**
600	40	33	7	**82,50%**
800	56	47	9	**83,92%**

last experiment.

Note that the probability of failure of a SEU occurring on a virtual machine that is healthy (none of the components it uses have been permanently damaged) is extremely low, so that, in reality, the probability of correcting double SEUs in this type of scenario (two damaged virtual machines) tends to 100%. In fact, if we use a probability of failure closer to reality on the healthy virtual machine (1% probability of SEU which, despite being closer, is still well above the actual probability of failure even in very aggressive radiation situations), we obtain a 100% correction of double SEUs. The data collected by running this third scenario are shown in Table 6.11.

Table 6.11: Correction of double SEU occurrences when two VMs have a higher failure probability due to HW damage - Scenario 3

Simulation time (s)	# of double SEU Occurrences	Correctly Masked	Incorrectly Masked	Correction (%)
200	14	14	0	**100%**
400	25	25	0	**100%**
600	39	39	0	**100%**
800	55	55	0	**100%**

It is important to emphasize that the data collected in this section do not correspond to the fault correction capability of our solution, but to the correction capability of double SEUs, an extremely rare situation, as reasoned above. The correction capacity of SEUs (much more frequent than double SEUs, although still infrequent) is, in any scenario, 100%.

6.2.3. Test Conclusions

Several conclusions can be drawn from the tests performed:

- The **correction of SEUs** due to radiation is, similar to other studies, **100%.**
- The correction capacity of the mechanism when **two SEUs randomly occur at the same time** (although this is a practically impossible situation) is, similarly to other studies, **33%.**

- The **great advantage of this mechanism** lies in its ability to correct for **double SEUs when they occur due to hardware damage or degradation**. This situation, unlike the previous one, is plausible, and the correction capacity of the mechanism increases as we approximate the statistical distributions of SEU occurrence to real values. In the scenarios proposed, we have obtained correction capacities of approximately 50%, 70%, 84% and, **in the most realistic case, 100%**. In other similar studies, failure correction mechanisms masked 33% of failures.

6.3. Trade-Off between SW and FPGA Voters

In this section we collect data on the performance of both the software voter and the FPGA (IP) voter implemented in the prototype. Then, based on the data, some conclusions about the suitability of using one or the other will be drawn.

6.3.1. Software Voter

Obviously, the voting process will require a different amount of time depending on the inputs received and the operating mode of the voter (*Normal*, *Degraded* or *Very degraded*; see section 4.3.1 for more details). **For this experiment, the *Normal* operating mode** has been used, as the other two are simplifications of it. With respect to the inputs, three different scenarios are considered:

- **Scenario 1**: the vote of the three partitions is identical.
- **Scenario 2**: the vote of two partitions coincides and that of the remaining one differs.
- **Scenario 3**: the vote of the three partitions is different.

The voting process runs on the same SoC used to test the prototype (its characteristics are shown in Table 5.1). The default **clock rate of the processor (666.67 MHz)** is used. Note that the Zynq-7000 SoC is a relatively powerful SoC compared to other embedded devices, especially in safety-critical industries such as space or aviation, so the chosen clock rate, without being maximum, is still high. In slower devices, the voting process could take longer.

Table 6.12: Voting process execution time in Scenario 1

	Scenario 1 (100 iterations)	**Scenario 1 (mean)**
Clock cycles	4,704	47
Time (ns)	7,060	71

First, the time required for the software voting process is evaluated when the three inputs are identical. Note that this case will be the usual case since, as explained in section 6.2.2, the presence of errors is in principle an unusual phenomenon. Table 6.12 shows the execution time of the voting process under these conditions for 100 iterations, as well as the average time for each vote.

As expected, this scenario is the fastest of the three, because it involves the least checks. Table 6.13 show the execution time of the voting process in scenarios 2 and 3. It is noteworthy that scenario 3 is the slowest, since it always involves a tie-breaking process based on the failure history of the partitions and, in case of a tie in the failure history, on a random decision. Details on the voting process can be found in Section 4.3.

Table 6.13: Voting process execution time in Scenarios 2 and 3

	Scenario 2 (100 iterations)	**Scenario 2 (mean)**	**Scenario 3 (100 iterations)**	**Scenario 3 (mean)**
Clock cycles	5,942	60	6,788	68
Time (ns)	8,910	89	10,180	102

6.3.2. FPGA (IP) Voter

As explained in Section 5.6.2, the RTL design of the *Normal* operating mode of the voter has been implemented for the hardware voter prototype. A series of snapshots of simulations done on this RTL design under different conditions are shown below. For these simulations, the same three scenarios have been considered as in the previous section. First, Figures 6.2 and 6.3 show the simulation when the inputs of the three partitions participating in the voting process are equal.

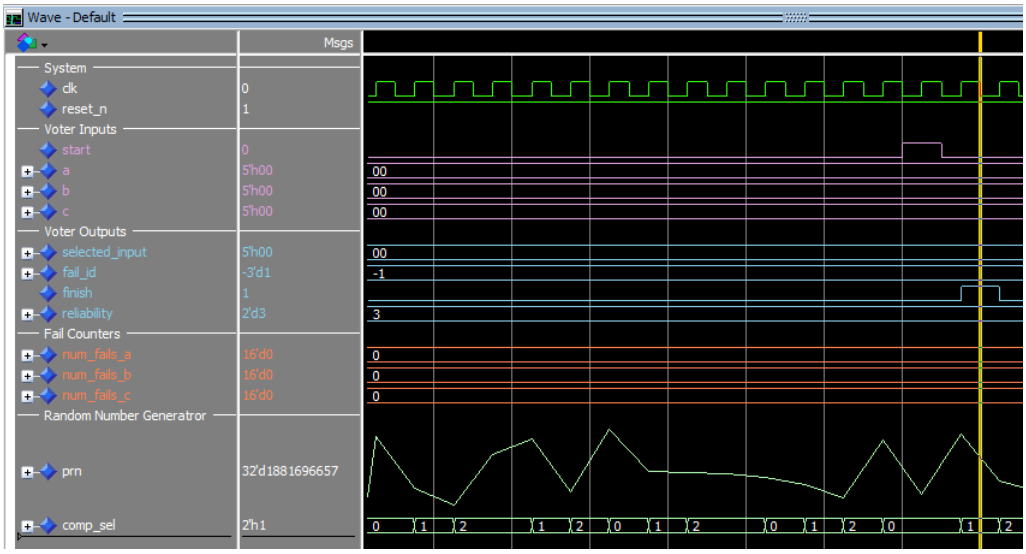

Figure 6.2: Simulation when every input is equal (I)

Figure 6.2 shows how at the beginning all inputs are 0. If the start of the simulation is signalled, in just a couple of clock cycles the end is indicated: the correct result is 0 and

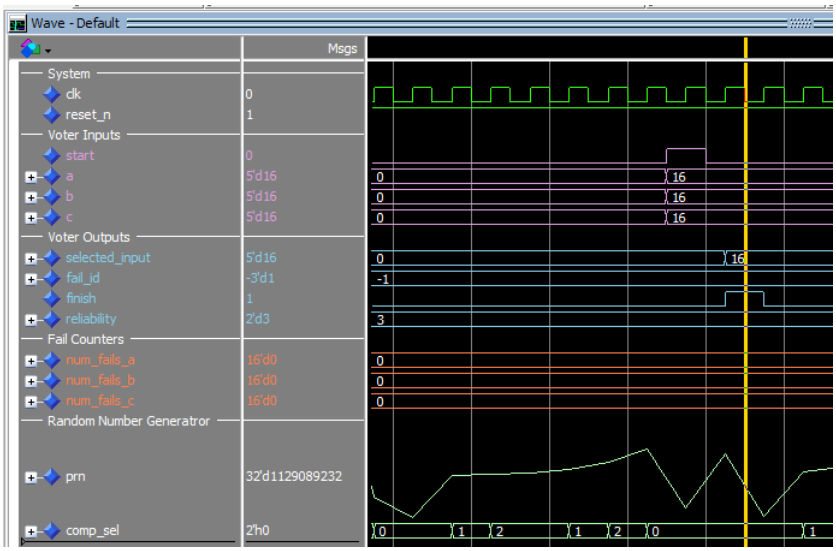

Figure 6.3: Simulation when every input is equal (II)

the reliability is 3 (VERY HIGH). Figure 6.3 shows how, by changing the input inputs (all to 16) and signaling the start, again, in a couple of clock cycles the system is calculating the output: the correct result is selected (16), reliability level 3 (VERY HIGH) is set and it is indicated that no partition has failed (-1).

Figures 6.4 and 6.5 show the behaviour of the system when one of the inputs is different from the other two.

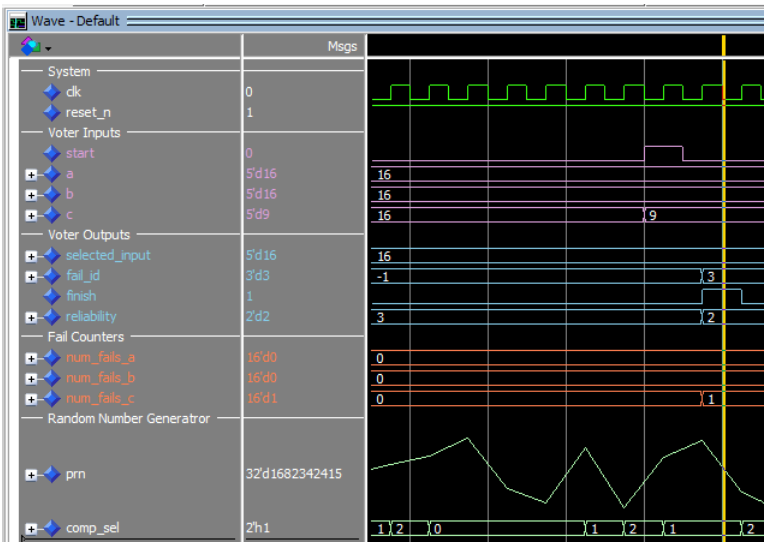

Figure 6.4: Simulation when one of the inputs is different (I)

In the case of Figure 6.4, the system comes from an iteration in which all inputs were equal (16). However, in the next iteration the value of partition C changes to 9. The final result selected is still 16 but, in this case, it is indicated that partition 3 has failed and the result is given a reliability level of 2 (HIGH). Similarly, Figure 6.5 shows an iteration in which the inputs of partitions A and C are equal (15), but partition B differs (9). In this case, the final result is 15, again with reliability level 2 (HIGH), and it is indicated that partition B has failed. It can also be seen how, in these scenarios, the failure counter of

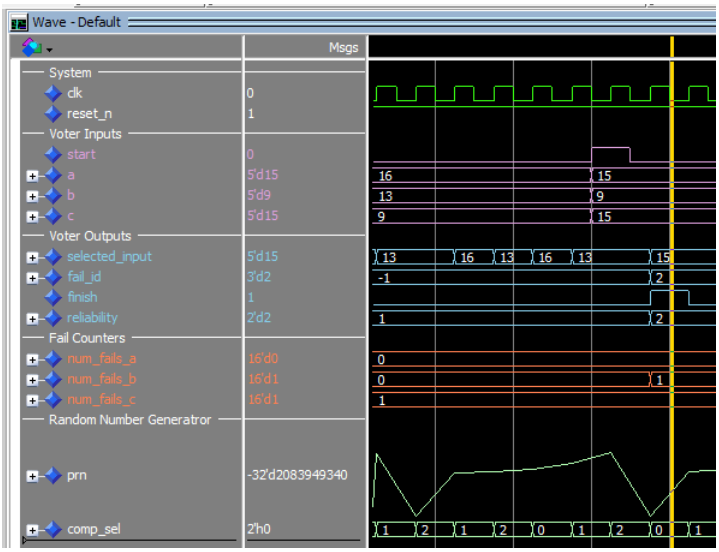

Figure 6.5: Simulation when one of the inputs is different (II)

each of the faulty partitions is increased internally.

Finally, Figures 6.6 and 6.7 show the behaviour of the system when all inputs are different.

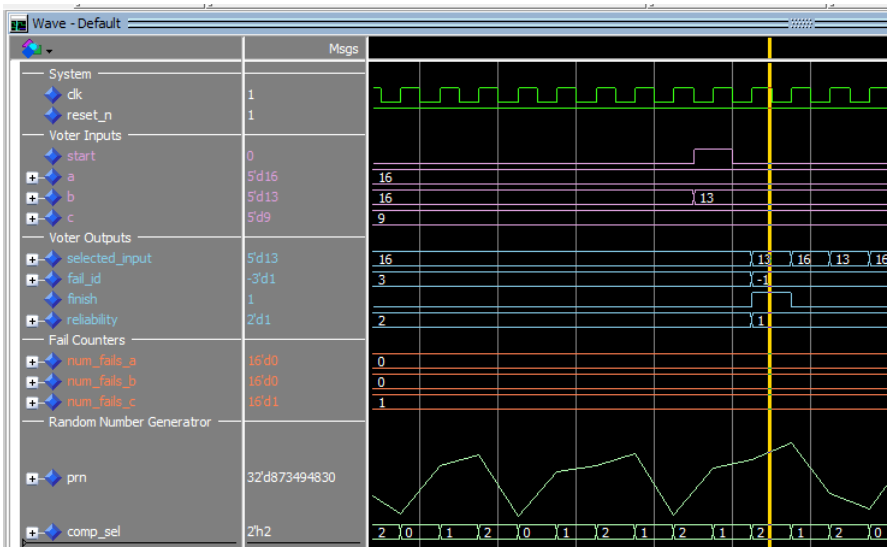

Figure 6.6: Simulation when all the inputs are different (I)

Figure 6.6 shows a scenario in which the failure counter of partition C is higher than that of partitions A and B, whose failure history is identical (none of them has failed so far). Therefore, when the three partitions yield different results (16, 13 and 9, respectively), the system will randomly decide between partition A and partition B inputs. This alternation in the decision can be observed in the *selected_inputs* signal. However, in Figure 6.7 the scenario is different: the failure counter of partition A is lower than the rest, so when a tie occurs, the system will decide to give validity to the input of partition A. In both cases, the reliability level is 1 (LOW) and, precisely because of the reliability level of the assumption, neither faulty partition is indicated, nor the fault counter of any partition is increased (see section 4.3 for more details about this logic).

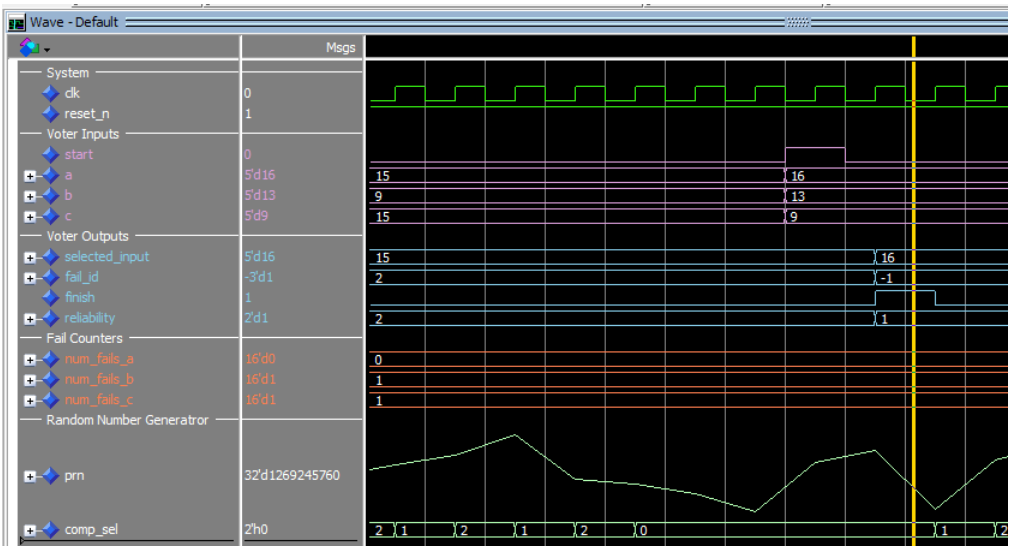

Figure 6.7: Simulation when all the inputs are different (II)

As can be seen from the figures presented in this section, the Normal operating mode of the voter takes 2 FPGA clock cycles to execute, regardless of the inputs. If the clock is at 100 MHz, a very conservative frequency given the low complexity of the block, these two cycles would take 20 nanoseconds to execute. The clock would allow a higher frequency: for example, at 150 MHz these two cycles would take 13.33 nanoseconds. Furthermore, if the information flow at the input of the block is continuous, a pipelined structure would allow valid outputs in half the time, at 6.67 nanoseconds. All this information is shown in Table 6.14.

Table 6.14: Execution time of the hardware (FPGA) voter

	Scenario 1	**Scenario 2**	**Scenario 3**	**Mean**
FPGA clock cycles	2	2	2	2
Execution time (ns) at 100 MHz	20	20	20	20
Execution time (ns) at 150 MHz	13.33	13.33	13.33	13.33
FPGA clock cycles using pipelining	1	1	1	1
Execution time (ns) at 150 MHz and using pipelining	6.67	6.67	6.67	6.67

6.3.3. Trade-Off Conclusions

From the experiments conducted, it is evident that implementing the voter in programmable logic as hardware, rather than as a software application, makes sense and has multiple advantages. First, it frees the software workload that can be used for other applications that

cannot be implemented in hardware. Also, the execution of **a voter iteration is almost 72% faster in its hardware version** than in its software version, even if we choose a very conservative frequency for the FPGA clock and the case that runs faster in software for the comparison.

To all this we must add the **biggest advantage of implementing the voter in FPGA**: the voter itself can be **redundant** so that it does not represent a single point of failure **without causing a time overhead**. Thus, two instances of the voter in software will take twice as long to run as one, three instances will take three times as long as one, and so on. In FPGA, using sufficient resources and running the redundant voters in parallel, the voting process will take the same even if it is redundant several times. Table 6.15 reflects the time savings using the hardware voter when the voting process is redundant.

Table 6.15: Execution time for multiple redundant voters

Number of voters	Hardware Voter (scenario 1) in ns	Software Voter (scenario 1) in ns	Time reduction using the hardware voter
1	20	71	71.83%
2	20	142	85.92%
3	20	213	90.61%
...	...	...	...
n	20	71 * n	1 - (20 / (71*n))

Therefore, the main advantages we hypothesized in section 5.6.2 are fulfilled when using the hardware voter instead of the software voter:

- Improve voter execution time by parallelizing voter instructions and pipelining: **At a minimum, the hardware voter runs almost 72% faster than the software voter.**
- Robust the voting process by parallel execution (without sacrificing execution time) of several instances of the voter: **The time savings increase more and more as redundant voters are added, since in the FPGA they do not necessarily imply a time overhead. By making the voter redundant by applying TMR, for example, with the hardware versions we can obtain an execution time more than 90% lower.**
- Relieve computational load on the CPU to execute other software partitions: **There no longer has to be a software partition in charge of the voting process, so the memory and processor resources allocated to another task can be reused.**

6.4. Summary

In this chapter, the different tests carried out on the prototype developed in chapter 5 are explained. Firstly, an experiment is carried out to try to measure the overhead involved in

the use of the XNG hypervisor in two different conditions: with and without the operating system. The results are inconclusive, but a hypothesis is put forward that could explain them. Secondly, the fault detection and correction capabilities of the prototype are unit tested by injecting faults in isolation and checking that the system responds as expected. Then, an exhaustive fault injection campaign is run to test the system's error correction and recovery rate. We have found that the system recovers from 100% of SEUs and its ability to recover from double SEUs is higher than normal in certain cases, due to partition health monitoring. Finally, the SW and FPGA versions of Voter are compared in terms of voting process speed and system overhead. We have found that, at a minimum, the FPGA voter runs 72% faster than the software voter, and this difference is even greater when the number of voters is increased, so that this is not a single point of system failure.

Once the prototype and solution have been evaluated, Chapter 7 will draw the overall conclusions of the research and set out possible lines of future work.

7. CONCLUSIONS AND FUTURE WORK

This chapter contains the conclusions drawn from the research work carried out and sets out some lines of future work. The chapter is divided into three main sections:

- Section 7.1 reviews the objectives set at the beginning of the thesis and justifies how these have been met. Furthermore, it compares the solution proposed and developed during this research work with the solution offered by related works.
- Section 7.2 collects the main contributions of the research carried out, dividing them into several categories:
  - Publications in academic journals.
  - Specialized conferences in which there is an associated publication.
  - Specialized conferences in which there has only been a presentation of the work carried out, without an associated publication.
  - Some projects in the aerospace sector in which the results obtained in the research carried out during the doctoral thesis have been used.
- Section 7.3 establishes some of the lines of work that we consider most interesting to continue the work carried out during the investigation.

7.1. Conclusions

During this doctoral thesis, a fault tolerance mechanism based on the use of a hypervisor has been proposed, developed and tested. Both this final objective and the specific objectives that were set to achieve it have been fulfilled, as justified below:

O1 Study and selection of the appropriate tools and environments for the research. For price, ease of use and availability of associated development environments, the Zynq-7000 family of SoCs has been chosen as the hardware environment to deploy the mechanism. In addition, an exhaustive review of the state of the art in embedded hypervisors has been carried out to select the most appropriate one for our use case. The chosen one has been XtratuM, from the company fentISS, but the survey will serve as a basis for similar decisions in other research projects. An experiment has also been conducted to measure the footprint of the XtratuM hypervisor on our platform, as there are no studies on the subject. Although we obtain some results, they are inconclusive and are proposed as the basis for a future line of work.

O2 Co-design and implementation of a hypervisor-based space system that will enable the deployment of mixed-criticality applications.

A space system has been designed and developed that allows the deployment of critical partitions and non-critical or general-purpose partitions. In addition, the system has a partition that monitors the health status of the hypervisor and the rest of the partitions and takes the appropriate measures (from resetting the hypervisor itself, a partition or even replacing one of the partitions with another partition that is on standby if the first one is not working properly or is giving wrong results too often) if it detects any malfunction, as well as a voter that allows executing the fault tolerant mechanism. Two different versions of the voter have been implemented to compare their performance: a first version on a software partition and a second one on a IP core running on the FPGA.

O3 Definition and implementation of a space use case for applying the Hypervisor-Based Fault Tolerance (HBFT) mechanism.

To test the developed fault tolerance mechanism, we decided to use a real use case of space systems: the ESA Near InfraRed (NIR) HAWAII 2-RG Data Processing Algorithms benchmarking software, which allows the processing of raw frames captured by the HAWAII 2-RG NIR detector. This detector is being used in several real space missions, such as the Hubble Space Telescope and the James Webb Space Telescope. This data processing algorithm is the functionality to be made redundant by the mechanism.

O4 Investigation of the effect of the HBFT mechanism on system safety and reliability.

We have performed an exhaustive fault injection campaign and it has been found that, as expected, the fault tolerance mechanism is able to detect and correct 100% of single SEUs occurring in one of the redundant partitions. Furthermore, we have contemplated a scenario that other similar studies do not contemplate, which is the potential occurrence of double SEUs if part of the hardware has been permanently damaged (in case there is no damage, the probability of two SEUs appearing at the same time is negligible). Because we have proposed and implemented a voter that tracks the health of each partition based on its success history, the system quickly detects the partition that has been damaged, so it assigns a low reliability to that partition and the probability of detection and correction of failures also tends to 100% (see tables 6.8, 6.9, 6.10 and 6.11). Finally, we have compared the footprint of using the fault tolerance mechanism when the voter runs as a software partition or when it runs in the FPGA. The conclusion is that the FPGA voter has a much smaller footprint, especially if it is decided to redundant the voter itself, which is necessary so that the voter does not become a single point of failure of the mechanism.

O5 Dissemination of results in specialized congresses related to the space industry and in research journals.

The work done during the PhD has been widely disseminated both in research journals (an article has been published in IEEE Access, journal with an impact index of Q1 according to the Journal Citation Reports, and another article is currently under review in that same journal) and in renowned conferences of the space sector, such as the Flight Software Workshop organized by NASA, in specialized boards of the European Space Agency or in the Space Engineering Congress organized by the Spanish Institute of Engineering.

O6 Strengthening of the links between the university (Universidad Carlos III de Madrid) and the company (SENER Aeroespacial) to allow future collaborations in space projects.
The work carried out during the thesis has been framed within the Madrid Flight on Chip project, in which SENER Aeroespacial and Universidad Carlos III de Madrid have collaborated closely. The possibility of both institutions continuing the work in a continuation of Madrid Flight on Chip is currently being considered. The work done during the thesis is being directly applied in space projects (such as SAFEST or ORU, described in sections 7.2.4 and 7.2.4) and Defence projects (details of which cannot be given for confidentiality reasons).

The academic and industrial nature of our work has allowed it to be very practically oriented and it differs from related works, since it combines certain advantages that facilitate its application in real projects. The work of Campagna et al. [84] is the first to consider the use of HPBF mechanisms in safety-critical embedded systems, but these mechanisms have several limitations: among others, there is no triple redundancy, so the failing partition cannot be detected (only the existence of the failure), the voting mechanism is a single point of failure because it is not redundant, and no health monitoring measures are taken when an error is detected. Sabogal and George [59] and Missimer et al. [89] propose solutions in which the vote is distributed and, therefore, the voting mechanism is no longer a single point of failure. However, both solutions are extensions of existing software products (Xen and Quest-V, respectively) and are limited to the use of these products, so they are not a usable mechanism in most commercial safety-critical systems. Moreover, as in the case of the Campagna et al. work, the voting mechanism consumes software resources like the other applications. To date, the most complete work according to our criteria is that of Esposito et al. [162] but there are some substantial differences between our work and theirs. For example, they contemplate the use of TTMR, which we have discarded for determinism reasons, and we have contemplated the possibility and effect of redundancy of the voter in the FPGA, increasing the reliability of the system without sacrificing performance. Table 7.1 summarizes some of these differences.

The main advantage (that is not fully reflected on table 7.1) of our work over the others, including that of Esposito et al., is the Health Monitoring measures taken. We have applied a concept that until now had not been applied in embedded systems, but only in cloud servers, which usually have much greater resources and run many virtual

Table 7.1: Advantages of our work with respect to previous solutions

	Campagna et al. [84]	Sabogal and George [59]	Missimer et al. [89]	Esposito et al. [162]	Our solution
Independent of a specific hypervisor	Yes	No	No	Yes	Yes
Detects the faulty partition	No	Yes	Yes	Yes	Yes
Redundant voting mechanism	No	Yes	Yes	No	Yes
HW-Accelerated voting mechanism	No	No	No	Yes	Yes
Allows for dynamic partitioning	Yes	Yes	No	Yes	Yes
Health Monitoring actions[a]	No	No	Yes	Yes	Yes

[a]Note that although this category indicates whether there is any health monitoring action in the solution, the actions differ for each solution.

machines in parallel: monitoring the health of each of these machines. A priori, applying this concept to an embedded system where redundant partitions are reduced to three does not seem to make much sense. However, if cold redundant partitions are configured and loaded in memory, which do not consume processing resources when in standby, this model has great potential: if any of the partitions (or any of the resources they use) is damaged, it can be replaced by a healthy one, so that the healthiest partitions are always running and offering a higher percentage of success in their calculations. This feature is what allows our solution to dramatically increase the system's success rate in the event that the effects of radiation or other accidents permanently damage part of the hardware, as discussed in section 6.2.2.

7.2. Contributions

7.2.1. Journals

During the development of the thesis, an article [188] has been published in **IEEE Access (journal with an impact index of Q1** according to the Journal Citation Reports):

Tamara Lugo, Santiago Lozano, Javier Fernández, and Jesús Carretero. A Survey of Techniques for Reducing Interference in Real-Time Applications on Multicore Platforms. *IEEE Access*, 10:21853–21882, 2022. doi: 10.1109/ACCESS.2022.3151891.

This survey reviews and categorizes the techniques that have tried to reduce interference in real-time multicore systems, especially those reflected in the existing scientific

literature since 2015. In addition, it presents proposals that use interference reduction techniques without considering the predictability issue. The review is analytical and inspects the advantages and disadvantages of each studied proposal.

Another article is currently under review in IEEE Access:

Santiago Lozano, Tamara Lugo and Jesús Carretero. A Comprehensive Survey on the Use of Hypervisors in Safety-Critical Systems.

The article provides a comprehensive review of the use of hypervisors in safety-critical embedded systems. In addition to reviewing all the articles published in recent years on the subject, hypervisors are categorised and compared according to different parameters, so that the survey can be used as a basis for future research or real industry projects.

7.2.2. Conferences with Publication

III and IV Space Engineering Congress

The Institute of Engineering of Spain (IEE), through its Space Committee, organizes biannually the Space Engineering Congress (CIE), an event that has the objectives of encouraging and contributing to the progress of space engineering, promoting the contact and development of space engineering and increase social interest, especially among young people and women, and promoting and contributing to the improvement of engineering education and research.

During the III edition of the congress, in 2020, we had the opportunity to present the objectives of the PhD and the Madrid Flight on Chip project, which had just started. This edition of the congress was held telematically due to the COVID-19 pandemic situation. In addition, during this edition a book of extended abstracts was collected, for which we also sent a contribution. In 2022, the IV edition of the congress was held and the Space Interdisciplinary Week (SIE) was established. On this occasion, we were invited to make a face-to-face presentation at the Spanish Engineering Institute in Madrid, and we had the opportunity to present the work done to date, after having presented the objectives two years earlier.

The paper published in the 2020 session (**Santiago Lozano, Juan Fombellida, Carlos Rodríguez, Cristina Tato, Jesús Carretero. MFOC Project: MPSoC-Based Multi-Purpose Execution Platform. In *III Congreso de Ingeniería Espacial : El espacio, la última frontera*, 27-29 Octubre 2020, Madrid, España. ISBN: 978-84-09-31948-0. pp. 120-122**) can be consulted in the book of abstracts.

For the 2022 session, the program is available on the congress website (`https://www.eiecongress.com/`). Our presentation was held on June 23rd in *session 02 B - Space Infrastructure*.

4S Symposium 2020 (Cancelled)

The Small Satellites, Systems and Services (4S) Symposium is a biannual event organized by the French Space Agency (CNES) and the European Space Agency (ESA), where aerospace industry players from all over the world, especially Europeans, meet to address technical discussions about systems and mission analysis, small satellite applications such as Earth Observation, science, telecommunication and navigation. The 2020 edition was to be held in May in Vilamoura, Portugal. For that edition, we submitted an abstract and were accepted as speakers, given the potential applicability of the work developed in this thesis for missions carried out by small satellites. We were also invited to submit a paper on the subject. The paper was called:

Santiago Lozano, Aranzazu Fernández, Juan Fombellida, Jesús Carretero. Integrated Solution for Space Applications Deployment in MPSoC. In *Proceedings of the 'The 4S Symposium - Small Satellites Systems and Services'*.

Unfortunately, the event was postponed twice due to the pandemic situation by COVID-19 and, finally, due to the Organizing Committee's refusal to hold it online, it was cancelled.

7.2.3. Conferences (Presentation-Only)

Flight Software Workshop 2022

The Flight Software Workshop is a presentation-only conference held annually in the United States (although the 2021 and 2022 editions were held online, due to the COVID-19 pandemic situation) and brings together engineers from leading companies and organizations from around the world to discuss flight architectures, software development, validation and development techniques and other challenges related to flight software. The importance of this workshop can be deduced from its Organizing Committee, formed in 2022 by NASA's Jet Propulsion Laboratory (JPL), Johns Hopkins University Applied Physics Laboratory, Southwest Research Institute and The Aerospace Corporation.

During 2021, we attended the workshop as listeners. After the good experience and after verifying the importance of the workshop and the quality of the material shared in it, during the year 2022 we sent an abstract exposing the work done in the PhD so far in the context of the Madrid Fligh on Chip project, and we were invited to make a 25 minute presentation for all workshop attendees.

The lecture was recorded and is available on YouTube under the title *FSW 2022: MIA: Multi-Purpose Space Platform using cFS and TSP - Santiago Lozano*: https://www.youtube.com/watch?v=SSXHYNODE9M.

Go2Space-HUBs Hackaton

Go2Space-HUBs is a collaborative project between Madrid (Spain), Coimbra (Portugal) and Tallinn (Estonia) that aims to secure the creation and prosperity of European companies related to the space sector, fostering the creation of networks between these companies to create value and innovation. In addition to facilitating the transfer of knowledge and technology, this project places great emphasis on the creation of new companies and assistance to newly formed small and medium-sized enterprises and start-ups. As part of the project, three hackathons were organized, bringing together companies and organizations with a long history in the sector, such as SENER Aeroespacial and the Universidad Carlos III de Madrid, with small and medium-sized companies. One of the hackathons was held in Madrid in February 2021, and we were invited to disseminate the work done during the PhD and the Madrid Flight on Chip project.

The hackathon website, where you can find my name and that of Víctor Pedro Gil (UC3M) among the attendees and presentations, is the following: https://go2space-hubs.eu/hackathon-2021/

ESA Joint Board on Communication Satellite Programmes 5G Advisory Committee

The European Space Agency has a committee called the Joint Board on Communication Satellite Programmes 5G Advisory Committee (5JAC), which is dedicated to monitoring and discussing the current status of communication satellite technologies, especially those dedicated to providing 5G cell phone coverage. Part of this committee's remit is to promote and disseminate initiatives that help develop these types of technologies. As a result, we were invited to present to the committee the results of the PhD and MFOC project and their possible applicability to communications satellites, and it is hoped that this may bear fruit in the form of future collaborations between ESA and UC3M/SENER.

7.2.4. Projects and Proposals

As previously mentioned in Chapter 2, when we exposed some of the most used software standards in the space, aviation and automotive industry, this thesis is delivered as a requirement to obtain the degree of an Industrial Ph.D. According to art.15 bis of RD 99/2011 of January 28, the designation "Doctorado Industrial" (Industrial Ph.D.) applies to those programs that are totally or partially carried out within a company and aims to promote research in companies in technological sectors. In our particular case, the thesis has been carried out both within SENER Aeroespacial and Universidad Carlos III de Madrid. Due to this collaborative nature, the contributions that this research has been able to offer to industrial activities are especially interesting. The projects and project proposals in which the results of this research have been used, totally or partially, are listed below.

Madrid Flight on Chip

The Madrid Flight on Chip (MFoC) project is the main project in which the work reflected in this thesis has been developed. MFoC is a research and innovation project co-funded by the Community of Madrid and the European Union. The main objective of MFoC is the development of the techniques and methodologies needed to improve the production of future generation aerospace satellite systems. These new techniques and methodologies will allow much more cost-effective space missions than in the past and with a much shorter development time, using SoC technology but maintaining the high level of reliability characteristic of this type of mission. In the long term, this could place Madrid in a privileged position in the small satellite market.

The project has explored modern hardware architectures, including high-capacity FPGAs and multicore processors, in search of techniques to address typical space mission problems such as power consumption and resistance to cosmic radiation. In the software area, some of the most important challenges faced by the project are the protection of applications through virtualization-based redundancy (work reflected in this thesis), the streamlining of model-based design, the promotion of component reusability between different space missions, and the automatic generation and testing of code.

S.A.F.E.S.T.

SAFEST (Smart Avionics for Flight Termination System) is a proposal to implement an Autonomous Flight Termination System, i.e. a system that is able to decide on-board and autonomously when and how to terminate a flying vehicle (e.g. a launcher). This system is based on low-cost COTS avionics and uses the platform developed in MFOC (including the virtualization provided by the hypervisor) as the basis for developing the system, thus reusing both the work done in the project and the work done in the PhD.

Currently, the proposal has been accepted and SAFEST will start as a SENER Aeroespacial led project in 2023.

ORU

ORU-BOAS is a proposal to implement a modular Orbital Replacement Unit (ORU) concept at TRL 6 compatible with different standard interfaces. This ORU will be a plug and play module for in-orbit demonstrators and will serve as a Satellite Construction Kit for future space missions. This system is based on low-cost COTS avionics and uses the platform developed in MFOC (including the virtualization provided by the hypervisor) as the basis for developing the system, thus reusing both the work done in the project and the work done in the PhD.

Currently, the proposal has been accepted and ORU will start as a SENER Aeroespacial led project in 2023.

Defence projects

The know-how acquired during the research described in this thesis, as well as some of its results, have been applied in several defence projects in which SENER has participated, directly or indirectly. For confidentiality reasons, details of these projects and the way in which the know-how acquired and the results obtained have been used cannot be given.

7.3. Future work

The work done in this research can be continued in different directions. Below are some of what we consider to be the most interesting ones:

- As we have explained in section 2.3, there are variations to Triple Modular Redundancy (TMR), usually oriented either to robusten redundancy or to lighten the overhead that redundancy implies. For the latter case, there are **Approximate TMR** mechanisms, which use approximate and lighter copies of the software to be redundant instead of exact copies, or **Reduce Precision Redundancy** (RPR), which use low-precision data and operations, simplifying the operations with the aim of obtaining calculations that are similar (not exactly the same) to the original. For simplicity and determinism, and given that the software to be redundant is not particularly heavy, during this research we have chosen to implement a classical TMR mechanism, but the implications of its alternatives on the overhead, determinism and robustness of the fault tolerance mechanism implemented could be studied in more depth, making a trade-off between the reliability they achieve versus the overhead they imply, and assessing in which use cases they could be recommended.
- **NMR** is a mechanism similar to TMR but in which N copies of the redundant software are used, instead of only three. Each redundant copy not only takes up memory space, but has to have an associated time slot in which this copy gets exclusive access to platform resources, such as the processor. In our use case, memory was not a limiting factor, but the time given to each partition by the scheduler is, so we have opted for a TMR mechanism that we make more robust by means of cold redundant partitions (i.e. redundant partitions in standby, which are activated only in case they are going to replace one of the partitions that are running at a given time). However, it would be interesting to perform a **trade-off to establish when NMR should be applied** (and how many copies would be used) based on the size of the redundant partition and the time it takes for each of these partitions to run a full cycle.
- To provide a comprehensive fault tolerance solution, the mechanism proposed in this thesis could be applied together with one or more **Hypervisor Fault Tolerance** (HFT) mechanisms (see section 2.4), so that not only the software applications are protected by using HBFT techniques, but also the hypervisor execution itself is protected by HFT techniques.

- We have conducted an experiment to **measure the overhead involved in using XNG** as a hypervisor (see section 6.1 for more details). Although we obtained interesting results, they are inconclusive: in most cases, the hypervisor overhead is in a reasonable range: between 3% and 5% more execution time. However, in certain cases, this overhead soars up to 40%. One hypothesis we have put forward is that the hypervisor may be occupying a significant part of the processor's instruction cache and, if an application contains a large number of different instructions in a short period of time, this cache saturates and the application's execution time is severely affected. This hypothesis has not been tested, as it deviates from the main objective of the research, but it is an interesting line of work for the future, as it would not only apply to XNG, but it is a problem that could be generalised to other hypervisors or operating systems. We are currently considering the continuation of this research in a Final Degree Project for a student of the Bachelor in Computer Science and Engineering at UC3M.

- Finally, it would be desirable to repeat the fault injection experiments discussed in chapter 6, but applying **real irradiation campaigns** with low-energy protons or heavy ions, rather than simulations.

BIBLIOGRAPHY

[1] J. A. N. Lee, "Claims to the term 'time-sharing'," *IEEE Annals of the History of Computing*, vol. 14, no. 1, pp. 16–54, 1992.

[2] C. Strachey, "Time sharing in large fast computers," in *Communications of the ACM*, ASSOC COMPUTING MACHINERY 1515 BROADWAY, NEW YORK, NY 10036, vol. 2, 1959, pp. 12–13.

[3] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley, "An experimental time-sharing system," in *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, ser. AIEE-IRE '62 (Spring), San Francisco, California: Association for Computing Machinery, 1962, pp. 335–344. DOI: 10.1145/1460833.1460871. [Online]. Available: https://doi.org/10.1145/1460833.1460871.

[4] J. A. N. Lee, R. M. Fano, A. L. Scherr, F. J. Corbato, and V. A. Vyssotsky, "Project mac (time-sharing computing project)," *IEEE Annals of the History of Computing*, vol. 14, no. 2, pp. 9–13, 1992.

[5] J. G. Kemeny and T. E. Kurtz, "Dartmouth time-sharing," *Science*, vol. 162, no. 3850, pp. 223–228, 1968.

[6] J. E. O'Neill, "'prestige luster' and 'snow-balling effects': Ibm's development of computer time-sharing," *IEEE Annals of the History of Computing*, vol. 17, no. 2, pp. 50–54, 1995.

[7] L. A. Belady, R. P. Parmelee, and C. A. Scalzi, "The ibm history of memory management technology," *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 491–492, 1981.

[8] R. J. Creasy, "The origin of the vm/370 time-sharing system," *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 483–490, 1981.

[9] J. Siebert, "Instructional use of a mainframe interactive image analysis system," *Photogrammetric engineering and remote sensing*, vol. 49, no. 8, pp. 1159–1165, 1983.

[10] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[11] R. R. Schaller, "Moore's law: Past, present and future," *IEEE spectrum*, vol. 34, no. 6, pp. 52–59, 1997.

[12] H.-m. D. Toong and A. Gupta, "Personal computers," *Scientific American*, vol. 247, no. 6, pp. 86–107, 1982.

[13] D. M. Lee, "Usage pattern and sources of assistance for personal computer users," *MiS Quarterly*, pp. 313–325, 1986.

[14] V. Aggarwal, V. Gopalakrishnan, R. Jana, K. K. Ramakrishnan, and V. A. Vaishampayan, "Optimizing cloud resources for delivering iptv services through virtualization," *IEEE Transactions on Multimedia*, vol. 15, no. 4, pp. 789–801, 2013.

[15] L. Qu, C. Assi, and K. Shaban, "Delay-aware scheduling and resource optimization with network function virtualization," *IEEE Transactions on Communications*, vol. 64, no. 9, pp. 3746–3758, 2016.

[16] E. Ray and E. Schultz, "Virtualization security," in *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, ser. CSIIRW '09, Oak Ridge, Tennessee, USA: Association for Computing Machinery, 2009.

[17] Yichao Jin, Yonggang Wen, and Qinghua Chen, "Energy efficiency and server virtualization in data centers: An empirical investigation," in *2012 Proceedings IEEE INFOCOM Workshops*, 2012, pp. 133–138.

[18] A. N. Al-Quzweeni, A. Q. Lawey, T. E. H. Elgorashi, and J. M. H. Elmirghani, "Optimized energy aware 5g network function virtualization," *IEEE Access*, vol. 7, pp. 44 939–44 958, 2019.

[19] C.-W. Lin, B. Kim, and S. Shiraishi, "Hardware virtualization and task allocation for plug-and-play automotive systems," *IEEE Design Test*, pp. 1–1, 2019. DOI: 10.1109/MDAT.2019.2932936.

[20] A. Desai, R. Oza, P. Sharma, and B. Patel, "Hypervisor: A survey on concepts and taxonomy," *International Journal of Innovative Technology and Exploring Engineering*, vol. 2, no. 3, pp. 222–225, 2013.

[21] A. Carvalho *et al.*, "Full virtualization on low-end hardware: A case study," in *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, 2016, pp. 4784–4789. DOI: 10.1109/IECON.2016.7794064.

[22] J. Sahoo, S. Mohapatra, and R. Lath, "Virtualization: A survey on concepts, taxonomy and associated security issues," in *2010 Second International Conference on Computer and Network Technology*, 2010, pp. 222–226. DOI: 10.1109/ICCNT.2010.49.

[23] B. Đorđević, V. Timčenko, S. Savić, and N. Davidović, "Comparing hypervisor virtualization performance with the example of citrix hypervisor (xenserver) and microsoft hyper-v," in *2020 19th International Symposium INFOTEH-JAHORINA (INFOTEH)*, 2020, pp. 1–6. DOI: 10.1109/INFOTEH48170.2020.9066288.

[24] A. Chierici and R. Veraldi, "A quantitative comparison between xen and kvm," in *Journal of Physics: Conference Series*, vol. 4, Bristol, England: IOP Publishing, 2010, p. 042 005.

[25] W. Chen, H. Lu, L. Shen, Z. Wang, N. Xiao, and D. Chen, "A novel hardware assisted full virtualization technique," in *2008 The 9th International Conference for Young Computer Scientists*, IEEE, New York, NY, USA: IEEE, 2008, pp. 1292–1297.

[26] J. Nakajima *et al.*, "Optimizing virtual machines using hybrid virtualization," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, 2011, pp. 573–578.

[27] C. Boettcher, R. DeLong, J. Rushby, and W. Sifre, "The mils component integration approach to secure information sharing," in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, 2008, pp. 1.C.2-1-1.C.2–14. DOI: 10.1109/DASC.2008.4702758.

[28] B. Sutterfield, J. A. Hoschette, and P. Anton, "Future integrated modular avionics for jet fighter mission computers," in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, 2008, 1.A.4-1-1.A.4–11. DOI: 10.1109/DASC.2008.4702749.

[29] L. M. Kinnan, "Use of multicore processors in avionics systems and its potential impact on implementation and certification," in *2009 IEEE/AIAA 28th Digital Avionics Systems Conference*, 2009, 1.E.4-1-1.E.4–6. DOI: 10.1109/DASC.2009.5347560.

[30] R. Wilhelm, C. Ferdinand, C. Cullmann, D. Grund, J. Reineke, and B. Triquet, "Designing predictable multicore architectures for avionics and automotive systems," in *Proc. of the Workshop on Reconciling Performance with Predictability (RePP)*, Oct. 2009.

[31] B. Annighoefer *et al.*, "Challenges and ways forward for avionics platforms and their development in 2019," in *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, 2019, pp. 1–10. DOI: 10.1109/DASC43569.2019.9081794.

[32] C. Cullmann *et al.*, "Predictability considerations in the design of multi-core embedded systems," in *Proceedings of Embedded Real Time Software and Systems*, May 2010, pp. 36–42.

[33] D. Kliem and S. Voigt, "A multi-core fpga-based soc architecture with domain segregation," in *2012 International Conference on Reconfigurable Computing and FPGAs*, 2012, pp. 1–7.

[34] P. Bieber, F. Boniol, M. Boyer, E. Noulard, and C. Pagetti, "New challenges for future avionic architectures," *Aeropsacelab Journal*, vol. 04, May 2012.

[35] M. A. Sánchez-Puebla and J. Carretero, "A new approach for distributed computing in avionics systems," in *Proceedings of the 1st international symposium on Information and communication technologies*, 2003, pp. 579–584.

[36] P. J. Prisaznuk, "Integrated modular avionics," in *Proceedings of the IEEE 1992 National Aerospace and Electronics Conference, NAECON 1992*, 1992, 39–45 vol.1.

[37] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," in *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, New York, NY, USA: IEEE, 2007, 2.A.1-1-2.A.1–10. DOI: 10.1109/DASC.2007.4391842.

[38] P. J. Prisaznuk, "Arinc 653 role in integrated modular avionics (ima)," in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, 2008, 1.E.5-1-1.E.5–10. DOI: 10.1109/DASC.2008.4702770.

[39] A. E. E. Committee *et al.*, *Arinc 653: Avionics application software standard interface (draft 15)*, 1996.

[40] A. Cook and K. Hunt, "Arinc 653 — achieving software re-use," *Microprocessors and Microsystems*, vol. 20, no. 8, pp. 479–483, 1997.

[41] S. H. VanderLeest, "Taming interrupts: Deterministic asynchronicity in an arinc 653 environment," in *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, 2014, 8A3-1-8A3–11. DOI: 10.1109/DASC.2014.6979531.

[42] S. H. VanderLeest and D. White, "Mpsoc hypervisor: The safe and secure future of avionics," in *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, 2015, 6B5-1-6B5–14. DOI: 10.1109/DASC.2015.7311448.

[43] N. Diniz and J. Rufino, "Arinc 653 in space," in *DASIA 2005-Data Systems in Aerospace*, vol. 602, Paris, France: ASD Eurospace, 2005.

[44] T. Gaska, C. Watkin, and Y. Chen, "Integrated modular avionics - past, present, and future," *IEEE Aerospace and Electronic Systems Magazine*, vol. 30, no. 9, pp. 12–23, 2015. DOI: 10.1109/MAES.2015.150014.

[45] J. Windsor and K. Hjortnaes, "Time and space partitioning in spacecraft avionics," in *2009 Third IEEE International Conference on Space Mission Challenges for Information Technology*, 2009, pp. 13–20. DOI: 10.1109/SMC-IT.2009.11.

[46] J. Andersson, M. Hjorth, F. Johansson, and S. Habinc, "Leon processor devices for space missions: First 20 years of leon in space," in *2017 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, IEEE, New York, NY, USA: IEEE, 2017, pp. 136–141.

[47] D. Paikowsky, "What is new space? the changing ecosystem of global space activity," *New Space*, vol. 5, no. 2, pp. 84–88, 2017.

[48] M. Höyhtyä, M. Corici, S. Covaci, and M. Guta, "5g and beyond for new space: Vision and research challenges," in *Advances in Communications Satellite Systems. Proceedings of the 37th International Communications Satellite Systems Conference (ICSSC-2019)*, 2019, pp. 1–16. DOI: 10.1049/cp.2019.1236.

[49] S. Trujillo, A. Crespo, A. Alonso, and J. Pérez, "Multipartes: Multi-core partitioning and virtualization for easing the certification of mixed-criticality systems," *Microprocessors and Microsystems*, vol. 38, no. 8, Part B, pp. 921–932, 2014. DOI: https://doi.org/10.1016/j.micpro.2014.09.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933114001380.

[50] A. Hughes and A. Awad, "Quantifying performance determinism in virtualized mixed-criticality systems," in *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, New York, NY, USA: IEEE, 2019, pp. 181–184. DOI: 10.1109/ISORC.2019.00041.

[51] H. Guissouma, H. Klare, E. Sax, and E. Burger, "An empirical study on the current and future challenges of automotive software release and configuration management," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, New York, NY, USA: IEEE, 2018, pp. 298–305. DOI: 10.1109/SEAA.2018.00056.

[52] R. Chitkara, W. Ballhaus, B. Kliem, S. Berings, and B. Weiss, "Spotlight on automotive pwc semiconductor report," PriceWaterhouseCoopers, Report, 2013.

[53] M. Z. MANIC, M. Z. PONOS, M. Z. BJELICA, and D. SAMARDZIJA, "Proposal for graphics sharing in a mixed criticality automotive digital cockpit," in *2020 IEEE International Conference on Consumer Electronics (ICCE)*, New York, NY, USA: IEEE, 2020, pp. 1–4. DOI: 10.1109/ICCE46568.2020.9212310.

[54] R. Schneider *et al.*, "Efficient virtualization for functional integration on modern microcontrollers in safety-relevant domains," *SAE Technical Papers*, vol. 1, Apr. 2014. DOI: 10.4271/2014-01-0206.

[55] S. Karthik *et al.*, "Hypervisor based approach for integrated cockpit solutions," in *2018 IEEE 8th International Conference on Consumer Electronics - Berlin (ICCE-Berlin)*, New York, NY, USA: IEEE, 2018, pp. 1–6. DOI: 10.1109/ICCE-Berlin.2018.8576222.

[56] M. Strobl, M. Kucera, A. Foeldi, T. Waas, N. Balbierer, and C. Hilbert, "Towards automotive virtualization," in *2013 International Conference on Applied Electronics*, New York, NY, USA: IEEE, 2013, pp. 1–6.

[57] T. Gaska, Y. Chen, and D. Summerville, "Leveraging driverless car investment in next generation integrated modular avionics (ima)," in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, 2016, pp. 1–9. DOI: 10.1109/DASC.2016.7778073.

[58] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "Kvm, xen and docker: A performance analysis for arm based nfv and cloud computing," in *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, New York, NY, USA: IEEE, 2015, pp. 1–8. DOI: 10.1109/AIEEE.2015.7367280.

[59] D. Sabogal and A. D. George, "Towards resilient spaceflight systems with virtualization," in *2018 IEEE Aerospace Conference*, 2018, pp. 1–8. DOI: 10.1109/AERO.2018.8396689.

[60] S. H. VanderLeest, "Arinc 653 hypervisor," in *29th Digital Avionics Systems Conference*, 2010, 5.E.2-1-5.E.2–20. DOI: 10.1109/DASC.2010.5655298.

[61] S. H. VanderLeest, D. Greve, and P. Skentzos, "A safe secure arinc 653 hypervisor," in *2013 IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC)*, 2013, 7B4-1-7B4–17. DOI: 10.1109/DASC.2013.6712638.

[62] S. H. VanderLeest, "Designing a future airborne capability environment (face) hypervisor for safety and security," in *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, 2017, pp. 1–9. DOI: 10.1109/DASC.2017.8102056.

[63] A. I. Kistijantoro and A. Gilbran, "Improving arinc 653 system reliability by using fault-tolerant partition scheduling," in *2018 5th International Conference on Advanced Informatics: Concept Theory and Applications (ICAICTA)*, 2018, pp. 182–187. DOI: 10.1109/ICAICTA.2018.8541309.

[64] T. Bijlsma *et al.*, "A distributed safety mechanism using middleware and hypervisors for autonomous vehicles," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, New York, NY, USA: IEEE, 2020, pp. 1175–1180. DOI: 10.23919/DATE48585.2020.9116268.

[65] A. Avanzini, P. Valente, D. Faggioli, and P. Gai, "Integrating linux and the real-time erika os through the xen hypervisor," in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, New York, NY, USA: IEEE, 2015, pp. 1–7. DOI: 10.1109/SIES.2015.7185063.

[66] B. Schulz and B. Annighöfer, "Evaluation of adaptive partitioning and real-time capability for virtualization with xen hypervisor," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 58, no. 1, pp. 206–217, 2022. DOI: 10.1109/TAES.2021.3104941.

[67] C. Dall and J. Nieh, "Kvm/arm: The design and implementation of the linux arm hypervisor," *SIGPLAN Not.*, vol. 49, no. 4, pp. 333–348, Feb. 2014. DOI: 10.1145/2644865.2541946. [Online]. Available: https://doi.org/10.1145/2644865.2541946.

[68] J.-S. Ma, H.-Y. Kim, and W. Choi, "Kvm-qemu virtualization with arm64bit server system," in *Cloud Computing*, Y. Zhang, L. Peng, and C.-H. Youn, Eds., New York, NY, USA: Springer International Publishing, 2016, pp. 334–343.

[69] R. Russell, "Virtio: Towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, Jul. 2008. DOI: 10.1145/1400097.1400108. [Online]. Available: https://doi.org/10.1145/1400097.1400108.

[70] V. K. Manik and D. Arora, "Performance comparison of commercial vmm: Esxi, xen, hyper-v kvm," in *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, New York, NY, USA: IEEE, 2016, pp. 1771–1775.

[71] G. P. C. Tran, Y.-A. Chen, D.-I. Kang, J. P. Walters, and S. P. Crago, "Hypervisor performance analysis for real-time workloads," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, New York, NY, USA: IEEE, 2016, pp. 1–7. DOI: 10.1109/HPEC.2016.7761610.

[72] H. Shi, X. Li, and Y. Zhao, "Database performance comparison of xen, kvm and osv using cassandra," in *2018 2nd IEEE Advanced Information Management,Communicates,Electronic and Automation Control Conference (IMCEC)*, New York, NY, USA: IEEE, 2018, pp. 27–30. DOI: 10.1109/IMCEC.2018.8469189.

[73] T. Müller, H. Askaripoor, and A. Knoll, "Performance analysis of kvm hypervisor using a self-driving developer kit," in *IECON 2022 – 48th Annual Conference of the IEEE Industrial Electronics Society*, 2022, pp. 1–7. DOI: 10.1109/IECON49645.2022.9968908.

[74] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "Xtratum: A hypervisor for safety critical embedded systems," in *11th Real-Time Linux Workshop*, Citeseer, 2009, pp. 263–272.

[75] A. Crespo, I. Ripoll, M. Masmano, P. Arberet, and J. Metge, "Xtratum an open source hypervisor for tsp embedded systems in aerospace," *Data Systems In Aerospace DASIA, Istanbul (Turkey)*, 2009.

[76] R. Zhou, Q. Zhou, Y. Sheng, and K.-C. Li, "Xtratum/ppc: A hypervisor for partitioned system on powerpc processors," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 593–610, 2013.

[77] E. Carrascosa, J. Coronel, M. Masmano, P. Balbastre, and A. Crespo, "Xtratum hypervisor redesign for leon4 multicore processor," *SIGBED Rev.*, vol. 11, no. 2, pp. 27–31, Sep. 2014. DOI: 10.1145/2668138.2668142. [Online]. Available: https://doi.org/10.1145/2668138.2668142.

[78] J. Galizzi *et al.*, "Temporal duplex-triplex on cots processors with xtratum," *DASIA 2016-Data Systems In Aerospace*, vol. 736, p. 20, 2016.

[79] A. Platschek and G. Schiesser, "Migrating a osek run-time environment to the oversee platform," in *Proceedings of 13th Real-Time Linux Workshop*, 2011.

[80] A. Larrucea, J. Perez, I. Agirre, V. Brocal, and R. Obermaisser, "A modular safety case for an iec-61508 compliant generic hypervisor," in *2015 Euromicro Conference on Digital System Design*, 2015, pp. 571–574. DOI: 10.1109/DSD.2015.27.

[81] V. Muttillo, L. Tiberi, L. Pomante, and P. Serri, "Benchmarking analysis and characterization of hypervisors for space multicore systems," *Journal of Aerospace Information Systems*, vol. 16, no. 11, pp. 500–511, 2019.

[82] S.-w. Kim, J.-W. Choi, J.-Y. Jeong, and B.-S. Yoo, "Development of rtems smp platform based on xtratum virtualization environment for satellite flight software," *Journal of the Korean Society for Aeronautical & Space Sciences*, vol. 48, no. 6, pp. 467–478, 2020.

[83] S.-W. Kim, B.-S. Yoo, J.-Y. Jeong, and J.-W. Choi, "Overhead analysis of xtratum for space in smp envrionment," *IEMEK Journal of Embedded Systems and Applications*, vol. 15, no. 4, pp. 177–187, 2020.

[84] S. Campagna, M. Hussain, and M. Violante, "Hypervisor-based virtual hardware for fault tolerance in cots processors targeting space applications," in *2010 IEEE 25th International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2010, pp. 44–51. DOI: 10.1109/DFT.2010.12.

[85] S. Grixti, N. Sammut, M. Hernek, E. Carrascosa, M. Masmano, and A. Crespo, "Separation kernel robustness testing: The xtratum case study," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, New York, NY, USA: IEEE, 2016, pp. 524–531. DOI: 10.1109/CLUSTER.2016.91.

[86] L. Masing *et al.*, "Xandar: Exploiting the x-by-construction paradigm in model-based development of safety-critical systems," in *2022 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2022, pp. 1–5. DOI: 10.23919/DATE54114.2022.9774534.

[87] T. Poggi *et al.*, "A hypervisor architecture for low-power real-time embedded systems," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, 2018, pp. 252–259. DOI: 10.1109/DSD.2018.00054.

[88] A. Crespo, P. Balbastre, J. Simó, J. Coronel, D. Gracia Pérez, and P. Bonnot, "Hypervisor-based multicore feedback control of mixed-criticality systems," *IEEE Access*, vol. 6, pp. 50 627–50 640, 2018. DOI: 10.1109/ACCESS.2018.2869094.

[89] E. Missimer, R. West, and Y. Li, "Distributed real-time fault tolerance on a virtualized multi-core system," *OSPERT 2014*, p. 17, 2014.

[90] A. Tavares, A. Dídimo, T. Lobo, P. Cardoso, J. Cabral, and S. Montenegro, "Rodosvisor — an arinc 653 quasi-compliant hypervisor: Cpu, memory and i/o virtualization," in *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012)*, New York, NY, USA: IEEE, 2012, pp. 1–10. DOI: 10.1109/ETFA.2012.6489588.

[91] A. Carvalho *et al.*, "Functionality farming in pok/rodosvisor," in *International Journal of Computer Science and Software Engineering (IJCSSE)*, Aug. 2016, pp. 161–174.

[92] S. Pinto, A. Tavares, and S. Montenegro, "Space and time partitioning with hardware support for space applications," *Data Systems In Aerospace (DASIA), European Space Agency,(Special Publication) ESA SP*, 2016.

[93] J. Martins, J. Alves, J. Cabral, A. Tavares, and S. Pinto, "Urtzvisor: A secure and safe real-time hypervisor," *Electronics*, vol. 6, no. 4, 2017. DOI: 10.3390/electronics6040093.

[94] S. Pinto, J. Martins, J. Lopes, M. Abreu, and A. Tavares, "Secssy hypervisor: Security-safety synergy for aerospace," in *DAta Systems in Aerospace (DASIA)*, Jun. 2017.

[95] J. Winter, "Trusted computing building blocks for embedded linux-based arm trustzone platforms," in *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, ser. STC '08, Alexandria, Virginia, USA: Association for Computing Machinery, 2008, pp. 21–30. DOI: 10.1145/1456455.1456460. [Online]. Available: https://doi.org/10.1145/1456455.1456460.

[96] M. Cereia and I. Bertolotti, "Virtual machines for distributed real-time systems," *Computer Standards & Interfaces*, vol. 31, pp. 30–39, Jan. 2009. DOI: 10.1016/j.csi.2007.10.010.

[97] S. W. Kim, C. Lee, M. Jeon, H. Y. Kwon, H. W. Lee, and C. Yoo, "Secure device access for automotive software," in *2013 International Conference on Connected Vehicles and Expo (ICCVE)*, 2013, pp. 177–181. DOI: 10.1109/ICCVE.2013.6799789.

[98] G. Cicero, A. Biondi, G. Buttazzo, and A. Patel, "Reconciling security with virtualization: A dual-hypervisor design for arm trustzone," in *2018 IEEE International Conference on Industrial Technology (ICIT)*, 2018, pp. 1628–1633. DOI: 10.1109/ICIT.2018.8352425.

[99] P. Lucas, K. Chappuis, B. Boutin, J. Vetter, and D. Raho, "Vosysmonitor, a trustzone-based hypervisor for iso 26262 mixed-critical system," in *Proceedings of the 23rd Conference of Open Innovations Association FRUCT*, ser. FRUCT'23, Bologna, Italy: FRUCT Oy, 2018.

[100] D. Dasari, M. Pressler, A. Hamann, D. Ziegenbein, and P. Austin, "Applying reservation-based scheduling to a uc-based hypervisor: An industrial case study," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, San Jose, CA, USA: EDA Consortium, 2020, pp. 987–990. DOI: 10.23919/DATE48585.2020.9116385.

[101] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look mum, no vm exits! (almost)," *ArXiv*, vol. abs/1705.06932, 2017.

[102] C. Hernàndez *et al.*, "Selene: Self-monitored dependable platform for high-performance safety-critical systems," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, New York, NY, USA: IEEE, 2020, pp. 370–377. DOI: 10.1109/DSD51259.2020.00066.

[103] A. Golchin, S. Sinha, and R. West, "Boomerang: Real-time i/o meets legacy systems," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, New York, NY, USA: IEEE, 2020, pp. 390–402. DOI: 10.1109/RTAS48715.2020.00013.

[104] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems," in *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, M. Bertogna and F. Terraneo, Eds., ser. OpenAccess Series in Informatics (OASIcs), vol. 77, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020, 3:1–3:14. DOI: 10.4230/OASIcs.NG-RES.2020.3. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/11779.

[105] G. Heiser and B. Leslie, "The okl4 microvisor: Convergence point of microkernels and hypervisors," in *Proceedings of the First ACM Asia-Pacific Workshop on Workshop on Systems*, ser. APSys '10, New Delhi, India: Association for Computing Machinery, 2010, pp. 19–24. DOI: 10.1145/1851276.1851282. [Online]. Available: https://doi.org/10.1145/1851276.1851282.

[106] G. Heiser, "Hypervisors for consumer electronics," in *2009 6th IEEE Consumer Communications and Networking Conference*, New York, NY, USA: IEEE, 2009, pp. 1–5. DOI: 10.1109/CCNC.2009.4784922.

[107] E. de Matos and M. Ahvenjärvi, "Sel4 microkernel for virtualization use-cases: Potential directions towards a standard vmm," *arXiv preprint arXiv:2210.04328*, 2022.

[108] U. Steinberg and B. Kauer, "Nova: A microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10, Paris, France: Association for Computing Machinery, 2010, pp. 209–222. DOI: 10.1145/1755913.1755935. [Online]. Available: https://doi.org/10.1145/1755913.1755935.

[109] H. Li, X. Xu, J. Ren, and Y. Dong, "Acrn: A big little hypervisor for iot development," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2019, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 31–44. DOI: 10.1145/3313808.3313816. [Online]. Available: https://doi.org/10.1145/3313808.3313816.

[110] K. Lampka and A. Lackorzynski, "Using hypervisor technology for safe and secure deployment of high-performance multicore platforms in future vehicles," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, New York, NY, USA: IEEE, 2019, pp. 783–786. DOI: 10.1109/ICECS46596.2019.8964912.

[111] T. Kim, D. Kang, S. Kim, J. Shin, D. Lim, and V. Dupre, "Qplus-hyper: A hypervisor for safety-critical systems," in *The 9th International Symposium on Embedded Technology (ISET)*, New York, NY, USA: IEEE, 2014, pp. 102–103.

[112] H. Joe *et al.*, "Remote graphical processing for dual display of rtos and gpos on an embedded hypervisor," in *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, New York, NY, USA: IEEE, 2015, pp. 1–4. DOI: 10.1109/ETFA.2015.7301581.

[113] Y. Lim and H. Kim, "Cache-aware real-time virtualization for clustered multicore platforms," *IEEE Access*, vol. 7, pp. 128 628–128 640, 2019. DOI: 10.1109/ACCESS.2019.2939859.

[114] D. Reinhardt and G. Morgan, "An embedded hypervisor for safety-relevant automotive e/e-systems," in *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, New York, NY, USA: IEEE, Jun. 2014, pp. 189–198. DOI: 10.1109/SIES.2014.6871203.

[115] M. Lemerre, E. Ohayon, D. Chabrol, M. Jan, and M.-B. Jacques, "Method and tools for mixed-criticality real-time applications within pharos," *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, vol. 0, pp. 41–48, Mar. 2011. DOI: 10.1109/ISORCW.2011.15.

[116] Q. ul Ain, U. Anwar, M. A. Mehmood, and A. Waheed, "Httm-design and implementation of a type-2 hypervisor for mips64 based systems," in *Journal of Physics: Conference Series*, IOP Publishing, vol. 787, 2017, p. 012 006.

[117] Q. Ain and M. A. Mehmood, "Runtime performance evaluation and optimization of type-2 hypervisor for mips64 architecture," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 2, pp. 295–307, 2022. DOI: https://doi.org/10.1016/j.jksuci.2019.11.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157819308390.

[118] M. Project, *Minos - Flexible Virtualization Solution for Embedded System*, version 0.4, 2022. [Online]. Available: https://github.com/minosproject/minos.

[119] B. Janßen, F. Korkmaz, H. Derya, M. Hübner, M. L. Ferreira, and J. C. Ferreira, "Towards a type 0 hypervisor for dynamic reconfigurable systems," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2017, pp. 1–7. DOI: 10.1109/RECONFIG.2017.8279825.

[120] Z. Jiang, N. C. Audsley, and P. Dong, "Bluevisor: A scalable real-time hardware hypervisor for many-core embedded systems," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018, pp. 75–84. DOI: 10.1109/RTAS.2018.00013.

[121] E. Hamelin, M. Ait Hmid, A. Naji, and Y. Mouafo-Tchinda, "Selection and evaluation of an embedded hypervisor: Application to an automotive platform," in *10th European Congress of Embedded Real Time Software and Systems (ERTS 2020)*, Paris, France: Association Aéronautique et Astronautique de France, Jan. 2020.

[122] T. O. Group, "Future airborne capability environment," The Open Group, Standard, 2020.

[123] J. Tokar, "A comparison of avionics open system architectures," *ACM SIGAda Ada Letters*, vol. 36, pp. 22–26, May 2017. DOI: 10.1145/3092893.3092897.

[124] T. Gaska, "Optimizing an incremental modular open system approach (mosa) in avionics systems for balanced architecture decisions," in *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, 2012, pp. 7D1-1-7D1–19. DOI: 10.1109/DASC.2012.6382420.

[125] T. K. Ferrell and U. D. Ferrell, "Rtca do-178b/eurocae ed-12b," in *Digital Avionics Handbook*, Boca Raton, FL, USA: CRC Press, 2000, pp. 467–478.

[126] C. Holloway, "Towards understanding the do-178c / ed-12c assurance case," in *7th IET International Conference on System Safety, incorporating the Cyber Security Conference 2012*, 2012, pp. 1–6. DOI: 10.1049/cp.2012.1499.

[127] L. Rierson, *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance*. Boca Raton, FL, USA: CRC Press, 2017.

[128] F. De Florio, "Chapter 4 - airworthiness requirements," in *Airworthiness (Third Edition)*, F. De Florio, Ed., Third Edition, Oxford, England: Butterworth-Heinemann, 2016, pp. 37–83. DOI: https://doi.org/10.1016/B978-0-08-100888-1.00004-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780081008881000045.

[129] J.-P. Blanquart *et al.*, "Software safety-a journey across domains and safety standards," in *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, Paris, France: Association Aéronautique et Astronautique de France, 2018.

[130] S. Fürst *et al.*, "Autosar–a worldwide standard is on the road," in *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, vol. 62, Düsseldorf, Germany: VDI Wissensforum GmbH, 2009, p. 5.

[131] S. Fürst and M. Bechter, "Autosar for connected and autonomous vehicles: The autosar adaptive platform," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, New York, NY, USA: IEEE, 2016, pp. 215–217. DOI: 10.1109/DSN-W.2016.24.

[132] D. Reinhardt, D. Kaule, and M. Kucera, "Achieving a scalable e/e-architecture using autosar and virtualization," *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, vol. 6, May 2013. DOI: 10.4271/2013-01-1399.

[133] M. Kotur, M. Dragojević, G. Velikić, and I. Bašičević, "Digital cockpit in autosar adaptive context," in *2018 IEEE 8th International Conference on Consumer Electronics - Berlin (ICCE-Berlin)*, New York, NY, USA: IEEE, 2018, pp. 1–4. DOI: 10.1109/ICCE-Berlin.2018.8576209.

[134] S.-H. Jeon, J.-H. Cho, Y. Jung, S. Park, and T.-M. Han, "Automotive hardware development according to iso 26262," in *13th International Conference on Advanced Communication Technology (ICACT2011)*, 2011, pp. 588–592.

[135] A. Abdulkhaleq, M. Baumeister, H. Böhmert, and S. Wagner, "Missing no interaction—using stpa for identifying hazardous interactions of automated driving systems," *International Journal of Safety Science*, vol. 2, no. 01, pp. 115–24, 2018.

[136] F. Z. Rokhani *et al.*, "Asil determination for motorbike's electronics throttle control system (etcs) mulfunction," in *EPJ Web of Conferences*, vol. 162, Les Ulis, France: EDP Sciences, 2017, p. 01 066.

[137] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata studies*, vol. 34, no. 34, pp. 43–98, 1956.

[138] A. Taber and E. Normand, "Single event upset in avionics," *IEEE Transactions on Nuclear Science*, vol. 40, no. 2, pp. 120–126, 1993. DOI: 10.1109/23.212327.

[139] I. Koren and S. Y. H. Su, "Reliability analysis of n-modular redundancy systems with intermittent and permanent faults," *IEEE Transactions on Computers*, vol. 28, no. 7, pp. 514–520, 1979.

[140] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962. DOI: 10.1147/rd.62.0200.

[141] D. Czajkowski, M. Pagey, P. Samudrala, M. Goksel, and M. Viehman, "Low power, high-speed radiation hardened computer and flight experiment," in *2005 IEEE Aerospace Conference*, 2005, pp. 1–10. DOI: 10.1109/AERO.2005.1559559.

[142] X. Xu and H. H. Huang, "On soft error reliability of virtualization infrastructure," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3727–3739, 2016.

[143] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot — a technique for cheap recovery," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI'04, San Francisco, CA: USENIX Association, 2004, p. 3.

[144] K. Kourai and S. Chiba, "Fast software rejuvenation of virtual machine monitors," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 6, pp. 839–851, 2011. DOI: 10.1109/TDSC.2010.20.

[145] M. Le and Y. Tamir, "Rehype: Enabling vm survival across hypervisor failures," *SIGPLAN Not.*, vol. 46, no. 7, pp. 63–74, Mar. 2011. DOI: 10.1145/2007477.1952692. [Online]. Available: https://doi.org/10.1145/2007477.1952692.

[146] M. Le and Y. Tamir, "Resilient virtualized systems using rehype," *ArXiv*, vol. abs/2101.09282, 2021.

[147] D. Zhou and Y. Tamir, "Fast hypervisor recovery without reboot," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, pp. 115–126. DOI: 10.1109/DSN.2018.00024.

[148] C. Tan, Y. Xia, H. Chen, and B. Zang, "Tinychecker: Transparent protection of vms against hypervisor failures with nested virtualization," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, IEEE, 2012, pp. 1–6.

[149] X. Xu, R. C. Chiang, and H. H. Huang, "Xentry: Hypervisor-level soft error detection," in *2014 43rd International Conference on Parallel Processing*, 2014, pp. 341–350. DOI: 10.1109/ICPP.2014.43.

[150] X. Xu and H. H. Huang, "Dualvisor: Redundant hypervisor execution for achieving hardware error resilience in datacenters," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015, pp. 485–494. DOI: 10.1109/CCGrid.2015.30.

[151] F. Cerveira, R. Barbosa, and H. Madeira, "Mitigating virtualization failures through migration to a co-located hypervisor," *IEEE Access*, vol. 9, pp. 105 255–105 269, 2021.

[152] P. Sousa, N. F. Neves, P. Verissimo, and W. H. Sanders, "Proactive resilience revisited: The delicate balance between resisting intrusions and remaining available," in *2006 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, 2006, pp. 71–82. DOI: 10.1109/SRDS.2006.37.

[153] J. Zhu, W. Dong, Z. Jiang, X. Shi, Z. Xiao, and X. Li, "Improving the performance of hypervisor-based fault tolerance," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–10. DOI: 10.1109/IPDPS.2010.5470357.

[154] H. P. Reiser and R. Kapitza, "Hypervisor-based efficient proactive recovery," in *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, 2007, pp. 83–92. DOI: 10.1109/SRDS.2007.25.

[155] A. Oates, "7 - reliability of silicon integrated circuits," in *Reliability Characterisation of Electrical and Electronic Systems*, J. Swingler, Ed., Oxford: Woodhead Publishing, 2015, pp. 115–141. DOI: https://doi.org/10.1016/B978-1-78242-221-1.00007-1.

[156] D. Binder, E. C. Smith, and A. B. Holman, "Satellite anomalies from galactic cosmic rays," *IEEE Transactions on Nuclear Science*, vol. 22, no. 6, pp. 2675–2680, 1975. DOI: 10.1109/TNS.1975.4328188.

[157] C. S. Guenzer, E. A. Wolicki, and R. G. Allas, "Single event upset of dynamic rams by neutrons and protons," *IEEE Transactions on Nuclear Science*, vol. 26, no. 6, pp. 5048–5052, 1979. DOI: 10.1109/TNS.1979.4330270.

[158] Y. Yeh, "Triple-triple redundant 777 primary flight computer," in *1996 IEEE Aerospace Applications Conference. Proceedings*, vol. 1, 1996, 293–307 vol.1. DOI: 10.1109/AERO.1996.495891.

[159] R. Berger *et al.*, "The rad750/sup tm/-a radiation hardened powerpc/sup tm/ processor for high performance spaceborne applications," in *2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)*, vol. 5, 2001, 2263–2272 vol.5. DOI: 10.1109/AERO.2001.931184.

[160] A. Frigerio, B. Vermeulen, and K. Goossens, "Isolation of redundant and mixed-critical automotive applications: Effects on the system architecture," in *2021 IEEE 93rd Vehicular Technology Conference (VTC2021-Spring)*, 2021, pp. 1–6. DOI: 10.1109/VTC2021-Spring51267.2021.9448672.

[161] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, 1995, pp. 1–11.

[162] S. Esposito, S. Avramenko, and M. Violante, "On the consolidation of mixed criticalities applications on multicore architectures," in *2016 17th Latin-American Test Symposium (LATS)*, 2016, pp. 57–62. DOI: 10.1109/LATW.2016.7483340.

[163] H. P. Reiser, F. J. Hauck, R. Kapitza, and W. Schröder-Preikschat, "Hypervisor-based redundant execution on a single physical host," in *Proc. of the 6th European Dependable Computing Conf., Supplemental Volume-EDCC*, Citeseer, vol. 6, 2006, pp. 67–68.

[164] H. P. Reiser and R. Kapitza, "Vm-fit: Supporting intrusion tolerance with virtualisation technology," in *Proceedings of the Workshop on Recent Advances on Intrusion-Tolerant Systems*, 2007.

[165] C. M. Jeffery and R. J. O. Figueiredo, "Towards byzantine fault tolerance in many-core computing platforms," in *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, 2007, pp. 256–259. DOI: 10.1109/PRDC.2007.40.

[166] D. J. Scales, M. Nelson, and G. Venkitachalam, "The design of a practical system for fault-tolerant virtual machines," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 4, pp. 30–39, Dec. 2010. DOI: 10.1145/1899928.1899932. [Online]. Available: https://doi.org/10.1145/1899928.1899932.

[167] S. Malik and F. Huet, "Adaptive fault tolerance in real time cloud computing," in *2011 IEEE World Congress on services*, IEEE, 2011, pp. 280–287.

[168] P. Das and P. M. Khilar, "Vft: A virtualization and fault tolerance approach for cloud computing," in *2013 IEEE Conference on Information Communication Technologies*, 2013, pp. 473–478. DOI: 10.1109/CICT.2013.6558142.

[169] F. Machida, M. Kawato, and Y. Maeno, "Redundant virtual machine placement for fault-tolerant consolidated server clusters," in *2010 IEEE Network Operations and Management Symposium - NOMS 2010*, 2010, pp. 32–39. DOI: 10.1109/NOMS.2010.5488431.

[170] A. Zhou, S. Wang, C.-H. Hsu, M. H. Kim, and K.-s. Wong, "Virtual machine placement with (m, n)-fault tolerance in cloud data center," *Cluster Computing*, vol. 22, no. 5, pp. 11 619–11 631, 2019.

[171] C. Gonzalez and B. Tang, "Ft-vmp: Fault-tolerant virtual machine placement in cloud data centers," in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, 2020, pp. 1–9. DOI: 10.1109/ICCCN49398.2020.9209676.

[172] F. Lenkszus, L. Emery, R. Soliday, H. Shang, O. Singh, and M. Borland, "Integration of orbit control with real-time feedback," in *Proceedings of the 2003 Particle Accelerator Conference*, IEEE, vol. 1, 2003, pp. 283–287.

[173] L. Szerdahelyi, S. Fugger, P. Espeillac, G. Monroig, T. Pareaud, and M. Casasco, "The bepicolombo attitude and orbit control system," in *9th International ESA Conference on Guidance, Navigation and Control Systems*, Jun. 2014.

[174] A. Karimi, F. Zarafshan, and A. b Jantan, "Pav: Parallel average voting algorithm for fault-tolerant systems," *International Journal of Advanced Computer Science and Applications*, vol. 2, no. 1, 2011.

[175] G. Martin and H. Chang, "System-on-chip design," in *ASICON 2001. 2001 4th International Conference on ASIC Proceedings (Cat. No. 01TH8549)*, IEEE, 2001, pp. 12–17.

[176] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor system-on-chip (mpsoc) technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1701–1713, 2008. DOI: 10.1109/TCAD.2008.923415.

[177] *Microzed - development board based on the zynq-7000 soc*, avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/microzed/, Accessed: 2022-05-15, 2022.

[178] T. Imken, T. Randolph, M. DiNicola, and A. Nicholas, "Modeling spacecraft safe mode events," in *2018 IEEE Aerospace Conference*, IEEE, 2018, pp. 1–13.

[179] R. P. Weicker, "Dhrystone: A synthetic systems programming benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, 1984.

[180] R. York, "Benchmarking in context: Dhrystone," *ARM, March*, 2002.

[181] L. H. Crockett, R. Elliot, M. Enderwitz, and R. Stewart, *The Zynq book: embedded processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 all programmable SoC*. Glasgow, Scotland: University of Strathclyde, 2014.

[182] J. B. Blake and R. Mandel, "On-orbit observations of single event upset in harris hm-6508 1k rams," *IEEE Transactions on Nuclear Science*, vol. 33, no. 6, pp. 1616–1619, 1986. DOI: 10.1109/TNS.1986.4334651.

[183] N. Kuznetsov, "The rate of single event upsets in electronic circuits onboard spacecraft," *Cosmic Research*, vol. 43, pp. 423–431, Nov. 2005. DOI: 10.1007/s10604-005-0066-9.

[184] A. Campbell, P. McDonald, and K. Ray, "Single event upset rates in space," *IEEE Transactions on Nuclear Science*, vol. 39, no. 6, pp. 1828–1835, 1992. DOI: 10.1109/23.211373.

[185] J. Schwank *et al.*, "Effects of particle energy on proton-induced single-event latchup," *IEEE Transactions on Nuclear Science*, vol. 52, no. 6, pp. 2622–2629, 2005.

[186] R. Koga and W. A. Kolasinski, "Heavy ion induced snapback in cmos devices," *IEEE Transactions on Nuclear Science*, vol. 36, no. 6, pp. 2367–2374, 1989.

[187] J. L. Titus, "An updated perspective of single event gate rupture and single event burnout in power mosfets," *IEEE Transactions on nuclear science*, vol. 60, no. 3, pp. 1912–1928, 2013.

[188] T. Lugo, S. Lozano, J. Fernández, and J. Carretero, "A survey of techniques for reducing interference in real-time applications on multicore platforms," *IEEE Access*, vol. 10, pp. 21 853–21 882, 2022. DOI: 10.1109/ACCESS.2022.3151891.