Scuola di Dottorato ∽ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer and Systems Engineering (XXX cycle)

Mixed-Criticality Systems on Commercial-Off-the-Shelf Multi-Processor Systems-on-Chip

By

Stefano Esposito

******

Supervisor(s):
Prof. M. Violante

Politecnico di Torino
2018

Declaration

To my father, for his pride support.

To my mother, for her love.

Acknowledgements

The work in this thesis was partially founded by the ARTEMIS/ECSEL JU project in the IPP "Computing platforms for embedded systems" (AIPP5), under grant agreement no. 621429 (EMC^2 project).

I would like to thank my advisor M. Violante for his support and insights.

I would also like to thank S. Avramenko, for his valuable insights, L. Bozzoli, for her acronym-finding capabilities, and all industrial partners of the EMC^2 project, especially M. Sozzi, M. Terrone, and M. Traversone, for their contributions.

Abstract

Avionics and space industries are struggling with the adoption of technologies like multi-processor system-on-chips (MPSoCs) due to strict safety requirements. This thesis propose a new reference architecture for MPSoC-based mixed-criticality systems (MCS) - i.e., systems integrating applications with different level of criticality - which are a common use case for aforementioned industries.

This thesis proposes a system architecture capable of granting partitioning - which is, for short, the property of fault containment. It is based on the detection of spatial and temporal interference, and has been named the *online detection of interference* (ODIn) architecture.

Spatial partitioning requires that an application is not able to corrupt resources used by a different application. In the architecture proposed in this thesis, spatial partitioning is implemented using type-1 hypervisors, which allow definition of resource partitions. An application running in a partition can only access resources granted to that partition, therefore it cannot corrupt resources used by applications running in other partitions.

Temporal partitioning requires that an application is not able to unexpectedly change the execution time of other applications. In the proposed architecture, temporal partitioning has been solved using a bounded interference approach, composed of an offline analysis phase and an online safety net.

The offline phase is based on a statistical profiling of a metric sensitive to temporal interference's, performed in nominal conditions, which allows definition of a set of three thresholds:

1. the *detection threshold* T_D;
2. the *warning threshold* T_W;

3. the α *threshold.*

Two rules of detection are defined using such thresholds:

Alarm rule When the value of the metric is above T_D.

Warning rule When the value of the metric is in the warning region $[T_W; T_D]$ for more than α consecutive times.

ODIn's online safety-net exploits performance counters, available in many MP-SoC architectures; such counters are configured at bootstrap to monitor the selected metric(s), and to raise an interrupt request (IRQ) in case the metric value goes above T_D, implementing the alarm rule. The warning rule is implemented in a software detection module, which reads the value of performance counters when the monitored task yields control to the scheduler and reset them if there is no detection.

ODIn also uses two additional detection mechanisms:

1. a control flow check technique, based on compile-time defined block signatures, is implemented through a set of watchdog processors, each monitoring one partition.
2. a timeout is implemented through a system watchdog timer (SWDT), which is able to send an external signal when the timeout is violated.

The recovery actions implemented in ODIn are:

- graceful degradation, to react to IRQs of WDPs monitoring non-critical applications or to warning rule violations; it temporarily stops non-critical applications to grant resources to the critical application;
- hard recovery, to react to the SWDT, to the WDP of the critical application, or to alarm rule violations; it causes a switch to a hot stand-by spare computer.

Experimental validation of ODIn was performed on two hardware platforms: the ZedBoard - dual-core - and the Inventami board - quad-core.

A space benchmark and an avionic benchmark were implemented on both platforms, composed by different modules as showed in Table 1

Each version of the final application was evaluated through fault injection (FI) campaigns, performed using a specifically designed FI system. There were three types of FI campaigns:

1. HW FI, to emulate single event effects;
2. SW FI, to emulate bugs in non-critical applications;
3. artificial bug FI, to emulate a bug in non-critical applications introducing unexpected interference on the critical application.

Experimental results show that ODIn is resilient to all considered types of fault.

Table 1 Benchmark composition.

Benchmark	HW Platform	Modules			
		Control law	Compression	Edge detection	Sensor log
Avionic	ZedBoard	✓	✓	–	–
	Inventami	✓	✓	✓	✓
Space	ZedBoard	✓	✓	–	–
	Inventami	✓	✓	–	✓

Contents

List of Figures

List of Tables

Nomenclature

Other Symbols

ρ_i	Priority of task τ_i
R_i	Response time of task τ_i
T_i	Period of task τ_i
$\mathscr{L}$	Criticality levels
μ	Expected value of a random variable
$\Phi(x)$	CDF of the standard normal distribution (i.e., a normal distribution with null mean and unitary variance)
ψ	Size of a sample in the RICE compression algorithm
σ	Standard deviation of a random variable
σ^2	Variance of a random variable
τ_i	Task i
B_L	Length of a block in the RICE compression algorithm
C_D	Confidence level in the computation of the detection threshold
C_W	Confidence level in the computation of the warning threshold
$C_{i\ell}$	Execution time of task τ_i as estimated for criticality level ℓ
$F_X(x)$	CDF for the random variable X
$f_X(x)$	PDF for the random variable X

k	Divisor for the RICE code computation in the RICE compression algorithm
k_{opt}	Optimal value of the divisor for a given block in the RICE compression algorithm
L_i	Criticality level of task τ_i
T_D	Detection threshold
T_W	Warning threshold

Acronyms / Abbreviations

ACP	Advanced coherency port
API	Application programming interface.
APSoC	All programmable system-on-chip (the Zynq)
APU	Application processor unit
ARINC	Aeronautical radio incorporated.
ASIC	Application-specific integrated circuit.
ASIL	Automotive safety integrity level
AXI	Advanced extensible interface
BE	Best effort
CAST	Certification Authorities Software Team
CDF	Cumulative distribution function
CFC	Control flow check.
CFE	Control flow error.
CFG	Control flow graph.
COTS	Commercial-off-the-shelf.
CPU	Central processing unit

DAL	Design assurance level
DMA	Direct memory access
DP	Dynamic priority scheduling algorithms
EASA	European aviation safety agency
ECC	Error-correcting code.
EDF	Earliest deadline first
ESA	European space agency.
FAA	Federal aviation administration
FD-SOI	Full-depletion silicon-on-insulator.
FET	Field effect transistor
FF	Flip-flop
FJP	Fixed job priority scheduling algorithms
FPGA	Field-programmable gate array.
FRT	Firm real-time
FTP	Fixed task priority scheduling algorithms
GAIA	Global astrometric interferometer for astrophysics.
GPIO	General purpose input/output
GPU	Graphic processor unit
HDMI	High-definition multimedia interface
HRT	Hard real-time
I/O	Input/output
I^2C	Inter-integrated circuit
IC	integrated circuit.

IEC	International electrotechnical commission
IMA	Integrated modular avionics.
IoT	Internet-of-Things
ISO	International Standards Organization
L1C	Level-1 cache.
L2C	Level-2 cache.
MCA	Mixed-criticality application
MCS	Mixed-Criticality System
MCS	Mixed-criticality system
MMU	Memory management unit.
MOSFET	Metal-oxide-silicon field effect transistor
MPSoC	Multi-processor system-on-chip
NMOS	N-channel metal-oxide-silicon field effect transistor
OCM	On-chip memory
ODIn	Online detection of interference.
OEM	Original equipment manufacturer.
PCI-e	Peripheral component interconnect express
PDF	Probability density function
PL	Zynq Programmable Logic
PS	Zynq Processing System
Q-SPI	Quad SPI
RAM	Random-access memory.
ROM	Read-only memory.

RTCA	Radio Technical Commission for Aeronautics
RTOS	Real-time operating system.
SATA	Serial advanced technology attachment
SD	Secure digital
SDMA	Smart direct memory access
SEE	Single event effect
SEFI	Single event functional interruption
SEL	Single event latch-up
SERDES	Serialized/deserializer
SET	Single event transient
SEU	Single event upset
SIL	Safety integrity level
SoC	System-on-chip
SOI	Silicon-on-insulator
SPI	Serial peripheral interface
SRAM	Static RAM.
SRT	Soft real-time
STS	Sporadic tasks system
SWaP	Space, weight and power
TID	Total ionizing dose
TTMR	Temporal triple modular redundancy
UART	Universal asynchronous receiver transmitter
USB	Universal serial bus

VLIW	Very long instruction word
WCET	Worst-case execution time
WD	Watchdog.
WDP	Watchdog processor

Chapter 1

Introduction

1.1 Processor-based embedded systems

Processor-based systems have been used in several applications since the start of the information era. Of all such applications a special mention should be made of the embedded systems.

Electronics systems tightly integrated with a physical device, implementing the control logic of the device, can be classified as embedded systems. A typical example of embedded systems are those integrated in household appliances. Such systems have been using small and simple computers for quite some time in order to implement the control logic. Think for instance of washing machines with multiple programs and chronological programming capabilities. Such systems use simple micro-controller-based computers in order to translate user commands into actuation commands to their physical parts, for instance to the electrical motors and valves of a washing machine.

The success of embedded systems is at the origin of the explosion of *smart appliances* and recently of the current *internet-of-things* (IoT) trend. Such applications have been enabled by the use of computers with increasing computational capabilities. In particular, industries like the home appliances and the consumer electronics in general - including the very flourishing industry of handhold devices such as the smart-phones - have been taking advantage of the peculiar capabilities of a rather new set of computers designed for embedded systems. The first step towards this evolution was the integration of an entire system on a single chip, creating *system-on-chip* (SoC) devices.

The SoC concept is a very successful one, and has been since used in various instances to integrate on a single chip a set of devices which would have been integrated on a circuit board before the development of such concept, such as memory controllers, communication devices, actuator drivers , etc. . . The SoC concept actually enabled the home theater and the early smart-phone era.

However, increasing performance requirements in applications using SoC required a change in the paradigm of development of such devices. It was not possible to achieve better performance simply increasing the clock frequency as it had been done before, due to the phisical limits of the implementation technology. Therefore, chip manufacturers implemented multi-core system. Multi-core systems are characterized by the integration of more than one central processing unit (CPU) on the same chip. Software developers took advantage of such new platforms to implement new systems with an increasing number of functionalities, partially by using the parallel programming paradigm that had been developed in the years before in other contexts, such as high-performance computing. Since there are several CPUs, each able of running a separate program in cooperation with other cores, applications were segmented into parallel chunks of computation, enabling a great increase in computational throughput, i.e. of the number of computations a systems is able to perform in a unit of time.

Soon after the concept of multi-core processors was merged with that of the SoC to create the multi-processor SoC (MPSoC). An MPSoC is a SoC which integrates more than one CPU. All CPUs in a MPSoC have access to the same system interconnect and as a consequence to all the on-chip and off-chip resources. This development actually enabled the explosion of the contemporary smart-phones.

Although performance gains granted by MPSoCs are of interest for any industry, up until now their use has been greatly reduced in industry domains facing strict and long certification processes. Industries such as civil avionics, automotive, railways and signaling, are all characterized by a set of peculiar requirements that make using MPSoCs a very challenging design choice, mostly due to the added cost of certification that new applications in these domains must face.

1.1.1 The challenge of MPSoCs for avionic and space applications

Use of MPSoCs presents several challenges to designers of safety-critical embedded systems such as those found on air-crafts or spacecrafts. One of the main characteristics of such systems are the safety requirements. Such requirements impose a strict development process to achieve the maximum level of certification or qualification, as needed in some cases. In this context, MPSoCs are ill suited, since they are very complex systems, which greatly increase the complexity of the whole certification item. Moreover, MPSoCs are too complex to be developed *in house* by an avionic equipment developer, nor are MPSoCs manufacturers willing to add special provisions to their systems to meet requirements of a relatively small market, such as the avionics or the space applications.

The main issue for a system designer developing an avionic application using a MPSoC, is the partitioning problem. In this thesis, the definition of partitioning is as in [1], where it is defined as “appropriate hardware and software mechanisms to restore strong fault containment”. A precise definition of *fault* is proposed in the remaining part of this chapter; however, to understand the definition of partitioning just proposed, one can think of a fault as of the root cause of any malfunctioning. To enforce partitioning as per the definition proposed above, that each application in a system should be self-contained, and any fault affecting any given application in a system should not affect any other application in the system. This property is strictly related to the safety requirements, since the issue for avionics and space applications is the added requirement of strict safety regulations. If the system designer is able to prove partitioning, he/she could also prove safety as required by certification authorities. The partitioning problem has also been identified by the certification authorities software team (CAST), which produced the position paper CAST-32 [2]. The main recommendation to avionic OEMs in [2] is essentially to not use multi-core systems, such as MPSoCs, unless detailed information can be gathered on several details of the architecture. Such details include very sensitive information on the internal bus architecture; such information is a crucial trade secret for MPSoCs manufacturers, one they are reluctant to share with avionics OEMs. As a consequence, the choice left for OEMs is to use MPSoCs as a single-core SoC; i.e., OEMs that use MPSoCs are forced to shutdown all computational cores but one in order to avoid additional challenges - i.e., costs - during the certification process. Similar recommendations are also contained in [3], which is a similar report produced in an European context. In [3], an additional recommendation for procurement of multi-core systems is added, that recommends selecting only systems for which some level of detail on the bus subsystem's behavior is provided by, or can be obtained from the manufacturer.

This thesis deals with the challenges arising from the use of multi-core systems, more precisely MPSoCs, in avionic and space applications. The main motive for use of MPSoCs in such industry domains is the consolidation of functionalities with different levels of criticality on a single hardware, therefore reducing space, weigh, and power (SWaP) consumption of the whole system. A system with such a characteristic is define a mixed-criticality system (MCS). An application featuring at least one MCS as subsystem is a mixed-criticality application (MCA). In an MCS, the main issue is the partitioning, as defined above. To deal with the partitioning problem, it is often divided in two sub-types:

1. *Spatial partitioning*: the property of containment with respect to faults affecting resources shared by more than one application in the system. The fault does not affect timing properties of the interaction between an application and the resource, but it affects the correctness of the result. For instance, a faulty application might change the configuration of a shared peripheral in an unpredictable way.

2. *Temporal partitioning*: the property of containment with respect to faults affecting the timing of an application. The fault affects some shared resource in a way that change the timing properties of the interaction between an application and the resource but it does not affect the correctness of the result. For instance, a faulty application might abuse a shared communication channel, causing a delay in other applications.

This thesis proposes a novel reference architecture which addresses the challenges of MCS deployed on MPSoCs. This thesis describes a set of mechanisms and support hardware that can be used by designers in order to develop certifiable equipment for both avionic and space applications. It presents the key features of the architecture and a realistic avionic use-case. Although the architecture proposed in this thesis was developed with mainly the avionic and space applications in mind, the MCS problem is of interest for many other industries. Suffice to cite the EMC^2 project [4], which was promoted by a consortium of tens of companies, operating in the industries of automotive, avionic, and space. The project was focused on the identification and development of technologies for use of multi-core systems in the scope of mixed-criticality systems. This thesis work is partially in the scope of the EMC^2 project.

This thesis is organized as follows: the remainder of the present chapter introduces key concepts used in the rest of the thesis; Chapter 2 contains a survey of the relevant literature on the topic of MCA for avionics and space applications; Chapter 3 describes the proposed solution in detail; Chapter 4 describes the experimental evaluation performed on the architecture; Chapter 5 presents the conclusion of the thesis and outlines possible extensions and new applications.

1.2 Definitions and Terminology

This section provides the definition for some of the terms that are commonly used in the literature that constitutes the context of this thesis. The literature will be properly analyzed in the next chapter, however it is useful to have some terminology in order to better describe the problem tackled by the work presented in this thesis. The main focus of this thesis is on dependability and in particular safety of a set of applications defined as safety-critical. Some industry domains, characterized by strict safety requirements, have in time adopted a set of safety regulations. All such regulations share a common root in the IEC-61508 [5]. This standard introduces some core concepts which are referenced and extended in more specific standards for some domains such as RTCA DO-178C [6] for avionics and ISO 26262 [7] for automotive. In the rest of this section, the basic terms for understanding the dependability problem are defined, and then a very brief survey of the cited safety standard is provided, for the parts most relevant to the scope of the thesis. A more complete review of the standards, although it might result interesting, is out of the scope of this work.

1.2.1 Basic definitions

For the following discussion, it is useful to define some terms that are widely used in safety standards as those cited in Section 1.2. In this thesis, definitions are as in [8] and [9], and are reported here to constitute a glossary for the remainder of the thesis.

In the scope of this thesis, a **system** is any entity carrying out a specific function while interacting with other systems, which constitute its **environment**. The **behavior** of a system is represented by how the system carries out its intended function. The way this behavior is perceived by an **user** is the **service** provided by the system. An user is another system, including in this definition an human being, interacting with the system through the **system boundary**. The portion of system boundary used to deliver the service is the **service interface**. The part of the system's state that can be observed at the service interface is its **external state**, whereas the rest is the **internal state**. The portion of the service interface with which the user interfaces is the **use interface**. A system provides **correct service** when it implements its function. A **failure** is the transition or deviation from the correct service to a non-correct service. The period during which the correct service is not delivered is the **service outage**. There is a **service restoration** when after a service outage the correct service is restored. A **fault** is the "adjudged or hypothesized cause of an error" [8]. A fault is **dormant** if it has not caused an error in the system, it is **active** otherwise. An **error** is any variation in the total state of the system with respect to the expected total state in that instant in time. An error can propagate within the system until it reaches the use interface of the system. When an error reaches the interface it becomes a failure or misbehavior.

1.2.2 Dependability and its attributes

The **dependability** of a system is its ability to "deliver service that can be justifiably trusted" [8]. The dependability of a system comprises several attributes, which have been defined over several contributions in the past. The definitions used in this thesis and reported below are the ones collected in [9]. Such attributes are:

- *availability*: readiness for service or the probability that service delivered by a system is correct at any given time;

- *reliability*: continuity of a correct service or the probability that the system delivers a service correctly at a time T given that it was delivering correct service at a time $T_0 < T$;
- *integrity*: the capability of avoiding improper alterations, which can be further categorized into:
  - *system integrity*: the ability of detecting a fault and to inform the user;
  - *data integrity*: the ability of preventing and correcting errors in the system database, which may result as a consequence of an error.
- *safety*: the capability of avoiding catastrophic consequences for the environment or the user;
- *maintainability*: the capability of being repaired and evolved.

1.2.3 Threats to Dependability

Threats to dependability include all such phenomena that may affect the system reducing its dependability. A very comprehensive classification of such threats is reported in [8]. For the scope of this thesis, just a subset of such classification is needed and is here reported for a better fruition of the remainder of the thesis.

A threat to dependability acts by introducing a fault in the system. Threats to dependability can be classified based on the moment in the life cycle of the system in which they are introduced. The life cycle of the system can be subdivided in a *development* phase and in a *use* phase. In the rest of this paper, the terms use phase and *mission* of the system might be used interchangeably.

Threats introduced in the development phase consists mainly in design errors or *bugs*. Such errors can be *malicious* or *non-malicious*. The most relevant case in the scope of this thesis is the unintentional design error, which introduces an unexpected behavior of the system, which does not respect its classification in a given circumstance. Authors of [8] suggest a further classification in *deliberate* and *non-deliberate* which is of little consequence to the scope of this thesis, however it might be interesting to notice that the certification process is put in place with the main objective of avoiding deliberate or non-deliberate non-malicious development errors.

Threats introduced in the use phase consists mainly of *natural* or *human-made* threats. Human-made use phase errors can be either *malicious* or *non-malicious*, and are differentiated mainly by the intention of the human causing the error. A malicious use phase human-made error is introduced with the purpose of causing a service failure, a non-malicious use phase human-made error is introduced without the intention of causing a service failure. Such errors are not considered in the rest of this thesis, since it is supposed that human users of safety-critical systems are always highly trained in the use of the system and as such are unlikely to introduce non-malicious human-made errors during the use phase. Natural use-phase errors are those most relevant in the scope of this thesis and as such are treated in a dedicated section below.

1.2.4 Natural Threats to Dependability During Use-Phase of a System

Natural threats to dependability during the use-phase of a system are introduced by unexpected interactions between the system and its environment. In this kind of interactions, the system is subjected to random stimuli coming from the environment, which may introduce a fault in the system (see Section 1.2.3).

Faults are can be classified as follows, according to[10].

- *permanent faults*, which persist in the system for an indefinite time;
- *transient faults*, which disappear from the system after a finite amount of time.
- *intermittent faults*, which causes errors to happen in burst only under given circumstances.

Permanent Faults. A permanent fault is originated by a defect in the system. In an electronics system, permanent faults can be either introduced during manufacturing, due to the intrinsic uncertainty of some processes in the realization of integrated circuits (ICs), or can arise due to aging. The main source of aging in semiconductors are microscopic effects of current in metal conductors embedded in the IC. Such effects as the electromigration or the hill effect, can cause the entire circuit or portion of the circuit to cease function. Since these, as well as the manufacturing defects, are physical defects, they cannot be removed from the circuit and are, as a consequence, permanent faults.

Permanent faults are classified according to several models, the most widely used are the single stuck-at and the delay model.

- *single stuck-at*: this fault model refers to the situation in which the value of a logic element in the system is always providing either a logic high or a logic low voltage;
- *delay*: this fault model refers to the situation in which a defect in the system is causing a given circuit to provide its outputs later than expected. As a consequence, the results of such computations may not be correctly sampled by sequential elements in the system, thus causing an error.

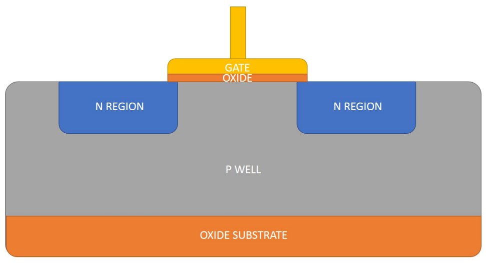
Intermittent Faults. Such faults can be modeled as permanent faults which only happen under given conditions. These faults can be difficult to distinguish from transient faults; however, some criteria are suggested in [10]:

1. intermittent faults occur repeatedly at the same location;
2. intermittent faults can be removed from the system by replacing the affected circuit, whereas transient faults are not removed by such an action;
3. intermittent faults cause bursts of errors when the fault is activated.

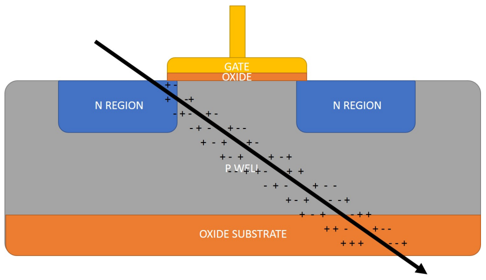
Transient Faults. A transient fault is introduced in the system by an unexpected interaction with the environment and has the characteristic of disappearing spontaneously from the system. However, such faults can cause errors and failures in the affected system. The main source of transient faults, in the scope of this thesis, are radiation effects on electronic circuits in the system. Since transients are the most relevant faults in the scope of this thesis, they are described below in more detail, together with their models.

Radiation-Induced Faults

The most relevant natural threat to dependability, in the scope of this thesis, is the radioactive environment in which the system may be operating.



(a) A typical NMOS transistor



(b) A particle strike in an NMOS transistor. The figure shows the displacement and diffusion of electron-well pairs.

Fig. 1.1 Diagrams of an NMOS transistor and of a particle striking a NMOS transistor

The radioactive environment is here defined as the capability of the environment of exchange energy with the system in the form of radiation packets or particles. Such event is very frequent and relevant for space applications, however it is becoming more relevant in more traditional applications such as avionics, high performance computing and even automotive, due to the shrinking of the technology node used in such applications.

Radiations act as unexpected stimuli to the system according to several physical mechanisms. The specifics of the effects of radiation on electronic circuits are beyond the scope of this thesis. Suffice to say that as a consequence of the interaction between radiations and the system, some unexpected behavior is observed. As already stated above, in the scope of this thesis, the system is always an electronic system. The interactions between an electronic systems and radiations can be considered as the effects of a particle strike on the matter composing the integrated circuits of the system. This matter is mostly silicon, since most of the circuitry in an electronic circuit is realized with such material. In fig. 1.1a it is depicted a typical implementation of a silicon-on-insulator (SOI) N-channel metal-oxide-silicon field effect transistor (MOSFET or simply MOS). This is the basic implementation of most modern technologies, although in more recent developments, the gate has been modified to overcome problems due to the reduced size of the channel, that is the distance between the two N regions in fig. 1.1a. The most relevant of such developments is the so-called FIN-FET, used for technology nodes of 28 nm and below. For sake of simplicity, the full extent of this technology is not described in this thesis. For the scope of this thesis it should be sufficient to mention that in this type of technology node, the channel is developed vertically above the surface of the silicon substrate and the gate envelopes the whole surface of this extrusion.

The main effect of a particle strike on a transistor as the one depicted in fig. 1.1a is the creation of a current, due to the displacement of electron-well pairs in the wake of the particle passing. When an high energy particle, like a proton or a heavy ion strikes the transistor, if it has enough energy, it can pass through the silicon lattice of the substrate, causing a parasitic current to flow through the transistor. Depending on the nature of the circuitry surrounding the transistor, this parasitic current can be seen as a fault appearing in the system. Due to the high number of variables that can change the effects of a particle strike, it is necessary to classify faults according to a set of models, which allows for an easier analysis of effects and outcomes.

Faults arising from radiation can be classified at first as effects of a single particle strike, which are the single event effects (SEE), and as effects of the total exposition to radiation, i.e. the total ionizing dose (TID). The TID effect is due to the damages introduced by several particle strikes in the system. When such damages are above a certain tolerable threshold, the system breaks down and is incapable of providing its service, unless the damaged parts are replaced by new ones. TID effects can be classified as permanent faults and are important when considering long space missions, although they are not considered in the scope of this thesis, since they are primarily of concern during components procurement.

In the scope of this thesis, SEE are much more interesting than TID. A SEE is the effect of a single particle strike, introducing a fault in the system. SEEs are further subdivided into:

- *single event upset (SEU)*: this fault model represents the effect of a single particle strike hitting a memory cell in the system and altering its content. The memory cells of an electronic system are susceptible to change in their contents due to parasitic currents induced by a particle strike.
- *single event transient (SET)*: this fault model represents the effect of a single particle strike hitting a logic gate within the system. The parasitic current resulting from the particle strike, temporarily alters the output of a logic cell. This wrong output can be sampled by multiple flip-flops (FFs) down the line, introducing multi-bit errors in the system memory.

- *single event functional interruption (SEFI)*: this fault model represents the effect of a single particle strike causing the system to enter a state from which it cannot recover without an external intervention. An example of SEFI, can be a particle striking the reset logic of a processor, causing it to enter in a permanent reset state; in this instance, the system can be recovered through a power cycle which can be initiated by a human user or by a companion system.
- *single event latch-up (SEL)*: this fault model represents the effect of a single particle strike resulting in a permanent error. In this case, the particle strike occurred in such a way to cause a high current to develop in a part of the system. Such a high current can cause a permanent damage in the system, in which case all or part of system functions are lost; if there is no permanent damage, the system can recover through a power cycle.

All SEEs can be classified as *soft errors*. As opposed to *hard errors* - which are permanent damages to the hardware of the system - soft errors are faults that can be removed from the system without replacing any parts of it. In the scope of this thesis, soft errors are most interesting than hard errors, since these are only recoverable through a recovery maintenance in which the faulty part of the system is replaced by a new fault-free one.

Although up until now only hardware faults have been considered, it is worth to mention and clarify that also software can be subject to faults. However, software faults can be introduced in the system only as design errors, meaning that they are deliberate or non-deliberate non-malicious design phase threats to dependability. However, a software fault, or bug, can be activated - excited - as a consequence of a SEE, which can put the system hardware in a state that was not foreseen during software development, and as such cause a software error. All software errors can be considered soft errors, since restoring the system to a known state usually involves also the resolution of the software error. Safety standards as those cited in Section 1.2 introduce several measures in the development process to ensure that in the system there is no software fault. Although, this kind of process is rather costly and as such is reserved only to highly critical parts of the system, i.e. those parts whose failure may result in loss of mission or in injury to humans or to the environment.

1.3 Fault Effects Analysis Techniques

Several techniques have been proposed over the years to analyze effects of faults on electronics systems. Components that have to meet quality requirements as those in automotive standards of quality - e.g., the automotive electronic council quality standards [11] - are commonly subjected to environmental conditions designed to speed-up aging processes which would naturally occur at a much slower pace. One such technique is the burn-in technique, in which a batch of newly produced ICs is inserted in an oven which reaches high temperatures - usually above 200 °C. In such an environment, phenomena like electromigration or hill effects occur much more quickly that in normal conditions, and allow to discard components subjected to the so-called infant mortality during a subsequent test phase.

The main scope of this thesis is however on transient faults; therefore, this section focuses on techniques used to introduce such faults in a working system in order to analyze their effects and to evaluate detection mechanisms. The process of introducing faults in a system is commonly referred to as fault injection [12]. Fault injection can be performed in several different ways.

1.3.1 Fault Injection by Physical Means

In this kind of fault injection, a physical mean is used to inject faults in the system. Such physical mean usually consists of a radiation beam or a laser source [13]. Experiments performed using radiation beams, or radiation experiments, are very widespread especially during qualification of components for space use. Each year, the Radiation Effects Data Workshop (REDW) provides an publication outlet for researchers around the world performing studies on radiation effects on electronics. A review of such results can provide an outline of the most used techniques.

Radiation experiments can be classified on the basis of the nature of the radiation beam. Radiation experiments are usually performed using heavy ions beams [14][15]. Results of such experiments can be used to estimate the response of the system to other types of radiations it might encounter in space[16]. Other commonly used radiation beams are protons [17][18], and neutron beams [19][20].

Data obtained through such experiments can be used to estimate dependability attributes of the system, as the figure of merit (FOM) method [16]. Another approach uses data from radiation experiments and from fault injection simulation experiments to estimate the availability of a system [21].

1.3.2 Fault Injection Simulation

Fault injection is a deliberate introduction of faults in operational systems to observe the response [12]. There are several methods to achieve such a goal that can be categorized as

1. Simulation Based
2. Software Based
3. Hybrid techniques

For the purposes of this thesis, the target system is considered to be a processor based embedded system in which a set of faults F is injected during normal operations with activation trajectories A. The behavior of the system is read at readouts R and dependability measures M are obtained through the fault injection. These sets form the so called FARM model [9]. The FARM model can be extended adding a set W containing workloads applied to the system during fault injection. The measures in M are obtained through a succession of fault injections, each called an *experiment*, forming a *campaign* [9]. Each experiment is characterized by a fault, an activation trajectory and a readout, i.e. by a triple $< f, a, r >$ with $f \in F$, $a \in A$ and $r \in R$. Measures in M are computed from readouts obtained during the campaign compared to the workloads W.

The FARM model can be applied to different techniques, being a general model for a fault injection campaign. Each technique is characterized by a set of attributes [9]:

Intrusiveness. The intrusiveness is defined by the changes that are applied to the system in order to implement the fault injection techniques. Often the memory content changes because some code is added to the system to support fault injection. Moreover, the system can show a different temporal behavior because of the fault injection. A good fault injection method tries to minimize the intrusiveness.

Speed. Duration of a fault injection campaign is given by the duration of a single experiment and the number of faults to inject, thus to reduce duration of a campaign two approaches can be applied:

1. Reduce duration of a single fault injection
2. Reduce size of the fault list, by applying some fault collapsing rule or by random sampling.

Cost. The fault injection system must have a negligible cost with respect to the cost of the system. In the cost are included all the hardware, software and the time needed to setup the system and adapt it to the target system.

Now it is in order to describe what each set in the FARM model should contain and its desirable characteristics.

Set F. This set contains the faults to be injected in a target system. To define this set a fault model must be selected, considering both the need of a fault model close to reality and feasibility of the fault injection campaign. Once the fault model has been selected, the fault list can be generated. The set F, which is practically the fault list, contains one entry for each fault, each entry containing [9]:

Fault injection time, which is the time instant when the fault is injected in the system. It can be expressed in a proper time unit, in number of instructions or in clock cycles.

Fault location, which is the part of the system affected by the fault. For example, if the selected fault model is SEU/SET, the location can be the register or the memory location to be affected by an SEU or the gate where an SET must be injected.

Fault mask, which is the way the exact bit to be changed is selected from an n-bit wide register or memory location.

Set A. This set contains the description of how the inputs should be applied to the system during the fault injection campaign. It can be very complex, especially in an embedded system featuring several inputs of different kind (digital/analog, high frequency/low frequency, ...) [9].

Set R. This set is filled by information gathered during the fault injection campaign observing the target behavior [9].

Set M. This set is filled at the end of the campaign and contains the results of the performed experiments. Faults effect are classified based on their effect on the system. Here is introduced the classification to which this thesis refers and that has been described in [9]:

- **Effect-less or silent fault.** The fault is never activated as an error and after injection stays in the system for a while as a passive fault before being removed. An example of this effect is the following: a fault is injected in a CPU register but the first operation performed on that register after the injection is a write operation, the fault is thus removed from the system and does not produce neither an error nor a misbehavior.
- **Failure.** The fault evolved as an error which was able to propagate to the system outputs.
- **Detected fault.** The fault evolved as an error, but the error was then detected by an implemented fault tolerance solution. Based on the way the fault was detected we can further classify detected faults as follows:
  - **Software-detected fault.** The module that detected the fault was implemented in the system's software which in turn signaled the presence of the error either to the user or to a fault recovery mechanism.
  - **Hardware-detected fault.** The module that detected the fault was implemented in the system's hardware. In a processor system, this can be either a special purpose peripheral or an exception mechanism implemented into the processor. An example of special purpose hardware could be a parity-checker used on some of the system's data: when a parity-violation is detected the parity checker raises an interrupt that implements the error-signaling. An example of a CPU-implemented exception is the illegal instruction trap triggered when the CPU fetches a malformed instruction from memory. This can be in turn the result of a fault affecting the instruction memory hierarchy.
  - **Timeout-detected fault.** The fault was detected by a timing violation of the outputs, which can be still correct but were provided after a

fixed deadline. This kind of faults can be detected, for instance, by a watchdog timer that expires when the system violates its timing.

1.3.3 Anatomy of a Fault Injection System

Many solutions for a fault injection system has been proposed with different approaches [22] [23] [24] [25] [26]. However, all fault injection systems share some concepts and components that are described in this section. Figure 1.2 shows how a fault injection system is organized and its components with respect to a target system.

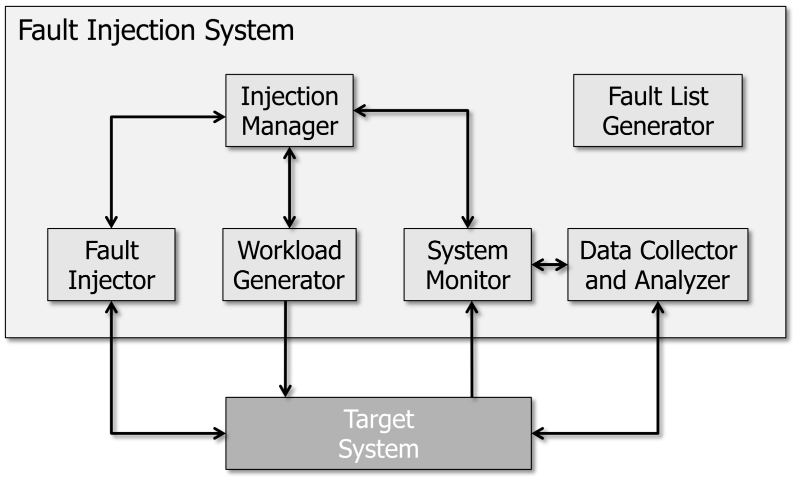


Fig. 1.2 Anatomy of a fault injection system.

Components of a Fault Injection System

This section describes the typical components of a fault injection system in terms of functionalities implemented and relations with the target system and the other components of the fault injection system.

Fault List Generator. A fault list generator is responsible for the creation and trimming of a fault list. The fault list depends on the used fault model, but with SEU/SET it can be at first very large. Since a very large fault list means a very long time needed to inject all its faults, various strategies are used to reduce its size. In [26] a fault collapsing strategy is used. The strategy is a workload-dependent fault collapsing exploiting the concepts of equivalent faults and effect-less faults. Two faults are equivalent if they propagate at the same time. A fault is effect-less if the first operation after its injection is a write operation that removes the fault. By applying this two concepts, the fault list size can be greatly reduced.

Fault Injector. The Fault injector is responsible for introducing the fault into the system. Depending on the used approach, the fault injector can be a piece of software running on a workstation computer, a piece of software running on the target system or a special purpose hardware. In simulation based solutions, the fault injector can be either a component specifically added to the model or it can be a script running inside the simulation tool in use. This component behavior is quite important since it could modify the behavior of the system, for instance by modifying its timing properties.

Workload Generator. The workload generator is responsible for generating and applying input patterns to the target system. The fault injection happens while the target system is processing the inputs generated by this component.

System Monitor. The system monitor is responsible for observing the system status and eventually triggering the data collector and analyzer component.

Data Collector and Analyzer. The Data Collector and Analyzer, fetch data from the system when triggered by the system monitor, and depending on the obtained data it decides what is the outcome of the experiment. At this end the data collector and analyzer can collect various information including the outputs, the internal state and timing information on the execution.

Injection Manager. The Injection Manager is the key point of a fault injection system. Its role is to supervise to the fault injection campaign by coordinating all other components. In particular the Injection Manager is in charge of

1. Extract one fault from the fault list
2. Instruct the Fault Injector with the information relevant to the extracted fault
3. Activate the System Monitor
4. Reset the Target System

5. Activate the Workload Generator
6. Categorize the experiment outcome based on the data collected by the Data Collector and Analyzer.

1.3.4 Fault Injection Systems

Many fault injection solutions have been developed and proposed over the years. The various fault injection systems can be categorized into three categories:

Simulation-based. To this class belong the systems that use a simulator to run a model of the target system and injecting faults as needed.

Software Implemented. This class includes systems that are implemented through software mechanisms running on the target system or on a host workstation connected to the target system.

Hardware-based. This kind of solutions includes systems implemented with an external hardware in charge of managing the faults and the injection procedure.

Hybrid Solutions. There exists solutions that can be described as hybrid between software and hardware-based systems, as the one that was used in this thesis and that will be described in following chapters.

Simulation-based Systems

This kind of systems rely on the description of the target system in some modeling language. Usually the system is described either in VHDL or Verilog and is simulated using fault-simulation tools [27]. Available logic-simulation or fault-simulation tools have usually a good built-in support for permanent fault models like stuck-at or delay, but lack a support for SEU/SET fault models, thus implementing a fault injection system requires one of the following approaches:

Modify the tool. The tool could be modified to implement SEU/SET fault models, when possible, or a different tool can be used with a built-in support for these fault models. However tools with such a built-in support are few and mostly come from universities.

Modify the model. The system model can be modified to add some support to the fault injection model. This is a rather simple solution allowing test engineers to easily implement a fault injection system and is one of the most successful approaches in this field with various proposed tools [22] [23] [24] [25]. The popularity of the solution is also due to the fact that it avoids to have to change the simulation tool already in use.

Using simulator commands. This approach exploits the scripting facilities commonly included in a simulation tool and the commands already implemented in the tool to force a value inside the model [28]. Using this approach is possible to support SEU/SET but also other fault models.

An example of the latest solution is AMATISTA [26]. The AMATISTA solution uses concepts already introduced in [28], that are

Golden Run. It is a fault free simulation of the system used to store information on the expected state and outputs. The state is saved at some given checkpoints that will be later used during fault injection

Static Fault Analysis. It is an analysis performed on the fault list to delete faults whose effect can be determined *a priori*. For instance, a fault that is injected just before a write operation on the target register. This fault can be deleted from the fault list, since it is granted to be effect-less.

Dynamic Fault Analysis. During the fault injection experiments, the status of the system is periodically compared to the state saved from the golden run to detect errors and failures.

In [26] the concept of *workload-dependent fault collapsing* is also introduced, i.e. the fault collapsing is performed based on the information available on the workload that is applied to the target system. This allows to exploit the concept of equivalent fault besides the concept of effect-less fault. Two faults are equivalent if they affect the same register and they propagate at the same instant in system execution. Of each group of equivalent faults, just one can be selected to be actually injected.

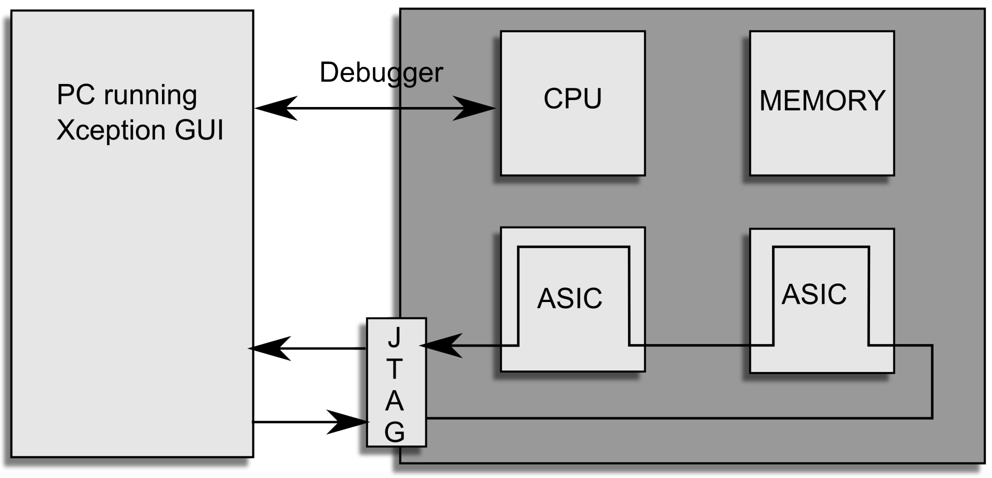


Fig. 1.3 Xception architecture.

Software-based Fault Injection

In Software-based systems, the real target system (commonly a full fledged prototype) is used to study the reaction of the system to faults. An example of such a system is Xception [29]. Xception implements a scan-based fault injection, exploiting the presence of a DFT infrastructure like a boundary scan chain, to inject the fault into the system through a JTAG interface, and a software-based fault injection to inject faults in memory and CPU registers.

Injection in memory and CPU registers is performed by a specific ISR which is activated by a software trap by any combination of the following events:

- An instruction is fetched from a specific address
- A timeout elapses
- A specific address in memory is accessed either for a read or a write operation

Injection in ASIC is performed through the JTAG interface controlled by the fault injector component of the Xception system.

Xception features a GUI that runs on a workstation. The GUI implements the functionality of the fault list generator, the injection manager and the fault injection through JTAG interface, besides the result presentation functionality. The general architecture of Xception is presented in Fig. 1.3.

1.4 Safety Standards

The basic concept to be defined for the scope of this thesis is that of *safety integrity level* (SIL). The SIL is the level of certification attributed to a given module of the system. It is defined in the IEC-61508 [5], which can be considered the basis of the other standards, which are more specific for a given industrial domain. The IEC-61508 is used as a generic standard for all such domains that have not adopted a more specific safety standard.

The IEC-61508 standard identifies 4 SIL levels, with the SIL 1 being the more relaxed level, i.e. the one with lesser safety requirements, and the SIL 4 the stricter level. The difference among the SIL levels is mostly identified by the requirements in terms of failure probability of a function certified at that level. There is no regulation mandating how this requirements should be satisfied, however to meet requirements of SIL 4, a dedicated development process should be put in place to ensure quality of both software and hardware modules and of their integration.

ISO-26262 Standard

The ISO26262 standard [7] is a safety standard dedicated to the automotive industry. Companies producing electronic equipment for automobiles adopted and enforce this standard for their products. The concept of ISO 26262 are similar to those in IEC-61508, although they are more specific for the automotive industry. ISO26262 defines the automotive safety integrity levels (ASIL); there are 4 levels identified with letters from A to D and a fifth level of *quality management* (QM). ASIL D is reserved for functionalities with a potentially catastrophic impact on the system, resulting in injury or death of occupants of the vehicle, whereas QM is for functionalities that have no impact on the safety of the occupants. To help define the level of a functionality, ISO26262 defines a set of metrics that can be either measured or defined by the experience of the designer. The metrics define characteristics of an adverse event which could result in injury to occupants of the vehicles and classify such events in terms of *severity* of the injuries resulting from the event, *exposure* and *controllability*. To each of such metrics, the designer should assign a level as defined in the ISO-26262 standard and based on such values determine the ASIL at which the affected functionality should be developed. Severity can be classified as:

S0 No injuries

S1 Light or moderate injuries

S2 Severe injuries

S3 Life-threatening or fatal injuries

Controllability is defined as the “likelihood that the driver can act to prevent the injury” and is classified in 4 levels:

C0 controllable in general;

C1 simply controllable;

C2 normally controllable;

C3 difficult to control or non-controllable.

Exposure is the likelihood of the operational conditions causing the adverse event; it is classified in 5 levels:

E0 very unlikely;

E1 very low probability;

E2 low probability;

E3 medium probability;

E4 high probability.

Once the metrics just defined have been evaluated, the ASIL level of the functionality can be determined depending on the value of each metric. To help the designer in establishing the ASIL of a functionality, the standard contains a table to cross the values of all the metrics, as reported in table 1.1. After having determined the

Table 1.1 ASIL evaluation table

Severity Class	**Exposure Class**	**Controllability Class**		
		C1	**C2**	**C3**
S1	**E1**	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	**E1**	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	**E1**	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

ASIL level, the designer is able to define the development process to use with the considered equipment or *item*. Development process for a high ASIL item are much more expensive and demanding than those for a QM or low ASIL item. Such process may require use of a specific subset of the programming language used in software development - e.g. MISRA-C - or a set of test procedures and coverage targets for both hardware and software faults. The target for an ASIL D item, is to have a very low probability of failure - similarly to the SIL 4 of IEC-61508 - in order to avoid serious injury or loss of life for occupants of the vehicle.

Avionic standards

The civil avionic industry adopted a set of two standards, the RTCA DO-178C [6] and the RTCS DO-254 [30], one focused on the software development and one for the hardware development process. The key concepts used in these standards are similar to those found in the IEC-61508 and the ISO-26262 standards. The basic concept of safety integrity level, is called design assurance level (DAL) in these standards. DALs are classified from E - roughly equivalent to the QM of ISO-26262 - to A - roughly equivalent to SIL-4 and ASIL D.

Table 1.2 DALs are defined based on the hazard analysis and are associated to a maximum probability of occurrence per flight hour.

Hazard Classification	Design Assurance Level	Probability
Catastrophic	A	1×10^{-9}
Hazardous	B	1×10^{-7}
Major	C	1×10^{-5}
Minor	D	–
No Effect	E	–

The main focus of these standards is on the safety of the entire aircraft operation, since it is conceivably the only real assurance on the safety of passengers and crew. The DAL is assigned to a given function of the analyzed system based on the impact of a failure of such system on the overall operation of the aircraft. The lightest DAL are assigned to functionalities that, if missing or damaged, pose no threat to the flight and operations of the aircraft, including the maintainment of acceptable environmental parameters in the cabin. Highest DAL are assigned to functions whose failure could pose a serious threat to the operations of the aircraft, including the possibility of a crash.

Avionics is one of the most strict industry concerning safety of the systems. Players in the avionic industry must face a certification process involving public agencies such as the federal aviation administration (FAA) in the United States and the European aviation safety agency (EASA) in the European Union. The certification process for a new avionic equipment includes the necessity for the developer, which proposes the new equipment for certification, to argue about the safety of the proposed equipment.

Contrary to the ISO-26262, avionic standards do not include ways of assign a DAL to a component, but define each DAL level according to safety requirements. To meet such safety requirements, the developer should use other standards dedicated to such analysis. Assessment of the DAL to be assigned to a new equipment is performed according to the ARP4761 [31] standard, which describes a process to asses safety of airborne equipment. Avionic software is often analyzed according to the IEEE Std 1228-1994 [32], which describes the contents of a software safety plan. The plan, proposed by the developer, is analyzed by the certification authority to whom the developer submits the design as part of the certification process.

Table 1.2 reports the definition of each DAL, based on the hazard analysis performed on the airborne equipment as resulting from the analysis done as prescribed in ARP4761 or other equivalent standards.

European Cooperation for Space Standardization Standards and Handbooks

The European cooperation for space standardization (ECSS) is an European organization with the goal of releasing and maintaining a set of standards for development of space applications. The ECSS has released several standards covering all aspects of developing a space application, including mechanical and pyrotechnical parts. The standards that are relevant for this thesis, are those relating to the safety of a system. More specifically the ECSS-Q-ST-40C standard [33] defines the criticality levels in a similar way as in the DO-178C and DO-254 standards described above. Such criticality definitions are system-level and are applied to a function of the space system. They can be applied to both hardware and software development using other ECSS standards, such as the ECSS-Q-ST-80C standard [34], which defines processes for the software dependability, and its supporting handbooks, such as the ECSS-Q-HB-80-30A, which defines how dependability and safety requirements specified in ECSS-Q-ST-80C should be met by the software.

The ECSS standards share several concepts with the standards already described, although they specify an entire product management and safety management process which is dedicated to the specifics of a space application. For instance, the ECSS-Q-ST-40-02C standard [35] is completely dedicated to the hazard analysis of a space product, with specificity to the hazard sources and consequences that are characteristics of a space applications and are not applicable, without some modification, to other types of application.

Another set of standards is applied to electronic and electrical equipment, as the ECSS-Q-ST-60C [36] and connected handbooks and standards, which are relative to techniques for development of safe and reliable hardware for spacecrafts.

1.5 Dependable Embedded Systems Design for Safety-Critical Applications

The complexity of embedded system increases with the requirements of their application. In this thesis, the focus is on safety-critical applications, with a special focus on the avionics and the space applications. In such applications, the requirements of safety and fault containment are particularly strong due to the nature of the application. In civil avionics, safety requirements are strictly enforced by certification authorities, upholding several standards such as those briefly described in Section 1.4. In space applications, safety requirements are not so strictly enforced, but they are a significant part of any contract between a space agency and the original equipment manufacturers (OEM). Such requirements can vary widely according to the application, especially due to the changes in the environment to which the equipment is exposed during the mission. Moreover, the focus in space application is on the continued operations of the equipment, to avoid outage of service or irrecoverable errors in a piece of equipment which cannot be easily maintained by human operators. In civil avionics, the focus is instead on the safety of the vehicle and on its capability of staying airborne even in the worst-case scenario, at least for as long as it is needed to safely reach ground.

The difference in the focus of these kind of application is evident in the lack of a universally adopted reference standard for space applications, whereas in the civil avionic industry there are several standard to support the design of such complex system, all universally adopted and accepted by certification authorities. However, both space and civil avionics share some common issues. In both space and civil avionics applications, there is a concern on the space that the equipment would take on the vehicle, on the power consumption during the operational phase, especially in terms of heat dissipation, and on the weight the equipment adds to the final product. Such concerns push designers towards an increase in integration for their applications. In the past, avionics were dominated by the *federated* architecture model [37] were each application was deployed to its dedicated equipment. All equipment acted to carry on the general function of the on-board avionics, managing the aircraft according to specification.

At the beginning of the 1990s, the focus moved towards a new system paradigm, the *integrated modular avionic* (IMA) architecture [38]. In the IMA architecture, different applications can share a single piece of hardware, provided that no interference is possible in any way. To help developers implement an IMA-based application, the standard *ARINC-653* [39] was developed and proposed. The ARINC-653 describes, among other things, an application programming interface (API) which all compliant real-time operating systems (RTOSes) should provide. Moreover, a scheduling scheme is proposed that grants absence of interference among multiple applications using the same hardware. The scheduling is based on a windowed time-line scheduling, in which each application has its dedicated window during which it can use the hardware. Tasks belonging to a given application can only run during the application's window and are preempted at the end of the window. Even though IMA greatly increased the integration for avionic applications with respect to the previously used federated architecture, the number and complexity of applications to be integrated on a single aircraft or satellite is ever growing and is pushing designer towards new solutions.

An interesting technical innovation could be represented by adoption of MPSoCs for avionic applications.

Nowadays, MPSoCs have reached a considerable complexity, offering a set of interesting features and performance figures for embedded designer. Most of the contemporary devices which are protagonists of the IoT explosion and most of the smart-phones, which are using MPSoCs as centerpiece of their architecture. Of course, avionics and space applications could greatly benefit from the adoption of such architectures. However, designers have been refrained from developing new equipment based on MPSoCs, due to the requirements of certification and qualification processes.

1.5.1 Fault Detection, Isolation, and Recovery

With the term *fault detection, isolation, and recovery* (FDIR), it is usually indicated a set of design techniques used to detect effects of a threat to dependability and to react by restoring correct functionality of the system. A large number of techniques has been developed with such intent, and a complete survey of all such techniques is out of the scope of this thesis. The interested reader may refer to [9] for a more exhaustive review of the most relevant techniques and principles. However, some of the concepts that are relevant to the topic of this thesis are outlined in this section;

more specific FDIR techniques developed with specific applications in mind are discussed in the next chapter.

Fault detection refers to set of techniques used to detect the occurrence of a fault in the system. Such techniques are often able to detect an error, rather than an actual fault, before its effect propagates to the service interface of the system. Fault isolation refers to a set of techniques used to define the actual location of the fault, to obtain a better result from the next set of techniques, i.e., fault recovery techniques. Fault recovery techniques are focused on canceling effects of a fault, allowing the system to continue working as expected. They operate by either removing the fault, e.g., replacing the faulty component, or by going around the faulty component, changing the behavior of the system in a way that keeps intact the functionalities while avoiding to use the faulty component.

A very successful FDIR technique, which has been applied to several applications in several domains, is the hardware redundancy. Hardware redundancy requires to add spare hardware to the system in order to grant correct functionality even if some of the redundant hardware is faulty. A common hardware redundancy scheme is the *triple modular redundancy* (TMR), depicted in Fig. 1.4.

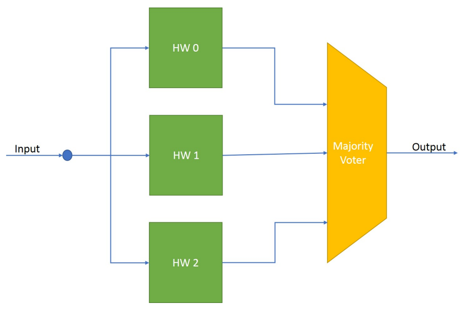


Fig. 1.4 Triple modular redundancy.

In TMR, there are three copies of a given hardware subsystem. All three copies implement the same functionality and receive the same input. The three replicas provide their outputs to a majority voter, which is an additional module that compares the three outputs according to the simple algorithm in Algorithm 1.1. This redundancy scheme is very successful, due to its efficacy against both permanent and transient faults. However, TMR is not able to detect or correct faults in more than one module; this limitation is still acceptable in most applications, since the occurrence of concurrent multiple faults is quite unlikely.

Another successful FDIR technique is the *stand-by spare* technique. In this technique, a whole subsystem is instantiated twice or more. At any given time, only one system (master) is providing its output to the service interface, whereas the other replicas (slaves) acts as stand-by spares. When a fault is detected by any fault detection technique, one of the stand-by replicas is activated and the faulty system is recovered as appropriate for the type of fault affecting it. There are two types of stand-by spare:

1. *Hot stand-by spare*: in this version, slaves run the same workload as the master, so that whenever a fault is detected by the master, one slave is immediately selected as the new master and can provide correct output to the service interface with very little delay.

2. *Cold stand-by spare*: in this version, slaves are in a reduced power state. When a fault is selected in the master, one of the slave is selected as the new master and brought to a full-power state. Before the output can be provided to the service interface by the new master, it must be configured, therefore this version introduces some reconfiguration delay.

Since the cost of both TMR and stand-by spare solutions is quite high, requiring additional hardware in the system, several software solutions were also developed for FDIR. In software-implemented fault tolerance (SIFT), the software running in a system is modified to implement detection techniques. SIFT techniques can be classified according to the type of error each technique is able to detect.

Data hardening techniques are focused on detecting errors in data used by the system. Such errors can be result of a fault affecting either memory or execution units.

Algorithm 1.1 Majority voter algorithm

```
1: procedure MajorityVote( $O_1$, $O_2$, $O_3$ )
2:     if $O_1 = O_2$ then
3:         return $O_1$
4:     else if $O_2 = O_3$ then
5:         return $O_2$
6:     else
7:         return $O_3$
8:     end if
9: end procedure
```

Algorithm 1.2 Data duplication example

```
1: $a \leftarrow b + c$
```

```
1: if $b_1 \neq b_2$ then
2:     Error
3: end if

4: if $c_1 \neq c_2$ then
5:     Error
6: end if

7: $a_1 \leftarrow b_1 + c_1$
8: $a_2 \leftarrow b_2 + c_2$
```

Most techniques in this category resort to some code modification as in Algorithm 1.2. Several techniques have been developed to reduce the overhead introduced by such techniques, but a complete survey is out of the scope of this thesis. The interested reader may refer to [9].

Other SIFT techniques are focused on detection of control flow errors (CFEs). CFEs represents deviation of the control flow of a program with respect to the intended control flow graph (CFG). The CFG is a connected graph in which each node $n \in \mathcal{V}$ is a basic block (BB) and each edge $e \in \varepsilon$ is a jump. In compiler theory, a BB is a partition of the program with only one entry point and only one exit point in which only the first instruction can be target of a jump and only the last instruction can be a jump.

Several control flow check (CFC) techniques have been proposed [9], all of them have in common the fundamental idea of identifying the correct path of execution and detecting deviations from it by performing some computation to check the actual progression of the program. A common idea is that of computing a signature which is updated and checked at each BB. At each point in the CFG, the value of such a signature is unique and known, therefore any change in the signature is resulting from an error in the control flow of the program.

The idea of controlling the execution flow of a program is also at the basis of the development of the watchdog (WD) [40]. A WD is a timer which should be reset at predefined moments in the program execution. A WD is able to detect all conditions that prevent the correct reset of the timer, therefore any CFE causing a deviation from the expected path, including infinite loops, can be detected by means of WDs. More complex WDs can be used to analyze bus transactions and detect erroneous patterns, or to implement other CFC techniques with an hardware support which is able to alleviate the performance overhead of a software implementation. Such solutions are often referred to as hybrid solutions.

Chapter 2

Mixed-Criticality Systems

This thesis deals with two industries with very strict safety requirements and with unique characteristics which separates them from most other industry domains. Avionics and space, often referred to as aerospace, is a niche industry with respect to the production volumes, especially when compared with other industries with unique safety requirements such as automotive or railway signaling. However, aerospace has a technical content very distinct from any other industry and as such requires a special treatment during design of electronics system.

Section 1.4 briefly describes two of the most relevant standards in safety critical applications - i.e. the ISO 26262 for automotive and the RTCA standards (DO-254 and DO-178C) for avionics. The main difference that can be extracted from that description of the standards is that in the avionic ones, the focus is on proof of safety, whereas in the automotive standard the focus is on the process that grants reasonable levels of safety. This is mainly due to the higher complexity of avionics systems in civil avionics with respect to the automotive, and to the absolute dependence of the flight on the correct functioning of the electronics, whereas in automotive applications the electronics, although important, are not as crucial to the correct functioning of the vehicle. To put it in ISO 26262 terms, a misbehavior in an automotive equipment is more controllable than a misbehavior in an avionic equipment.

The difference is even more extreme when considering space applications. In this case, the safety of the application is also impaired by the radioactive environment, which has been coped with in different ways in the past, most noticeably using specially devised hardware.

This thesis proposes a solution to the problem of using new hardware for this set of special applications. The main characteristics of such hardware is that it has not been designed for the needs of such applications. Therefore, some design choices have impaired its use in the avionics and space applications.

In this chapter, a literature survey is proposed. The survey describes how the key aspects of the central problem in this thesis have been framed and solved in the past, so that the reader can have a sufficient understanding of the context to better understand the proposal of the thesis.

This chapter is organized as follows: Section 2.1 describes the mixed-criticality problem and scheduling solutions applied to it; Section 2.2 describes the peculiar solutions used in the space industry to cope with radiation induced soft errors.

2.1 Mixed-Criticality Systems Model

The term mixed criticality applies to systems in which two or more applications, classified at different levels of criticality, share hardware resources, including - but not limited to - the CPU. The most common case, and the most relevant for the scope of this thesis, is the scenario in which a safety-critical application shares hardware with one or more non-safety-critical applications. An example of such scenario could be the current trend in automotive of integrating infotainment functionalities with critical information such as speed of the vehicle, motor's rounds per minute, etc...

Using mixed-criticality systems (MCS) is the most interesting solution to problems concerning space, weight and power (SWaP) requirements of a system. As such it is a very interesting solution for avionics and space industries, where space is at a premium and operational cost is directly linked to weight and power of equipment.

The first impulse towards MCS for avionics was generated by the introduction of the IMA. Considering an IMA-based system implemented using single-core processors, mixed-criticality becomes a scheduling problem. Through scheduling alone, it should be possible to grant safety of the application, by granting absence of interference between safety- and non-safety-critical applications. The seminal paper for scheduling of mixed-criticality applications was published by Vestal in 2007 [41]. In this paper, Vestal proposed the problem formulation that was used in subsequent developments.

Although Vestal contribution and subsequent developments have provided a significant impulse to research in the field of MCS, it is worth to notice that Vestal and the research based on Vestal's model used some assumptions that do not have any reference to the safety standards used in the industrial domains interested by the MCS problem, as pointed out in [42]. In the scope of this thesis, Vestal model has been used as a starting point; however, the assumptions challenged in [42] are not used in this work, and the approach described in Chapter 3 is not dependent on the validity of such assumptions.

Before describing Vestal's work, a brief description of basic scheduling concepts is in order.

2.1.1 Scheduling Concepts

For the scope of this analysis, a system is considered as the union of its tasks. A *task* is a running program that satisfies at least one of the functional system requirements. A task can be executed more than once during the system life, for instance as response to external stimuli. Each execution of a task is a *job*. In real time systems, of which mixed-criticality systems are a special case, it is common to refer to the sporadic task model for schedulability analysis. In such model, a tasks systems is indicated as τ and is a set of tasks τ_i with $i \in \mathbb{N}^+$. Each task τ_i is characterized by the following characteristics:

- A worst case execution time (WCET) estimation C_i.
- A period T_i, which is the time between two consecutive jobs of a task.
- A deadline D_i, which is commonly defined as a relative time after which a job should be completed.

Using this model, a sporadic tasks system (STS) can be treated mathematically, for instance by employing the queue theory. A more detailed description of such approaches is out of the scope of this thesis; however, some results are interesting for the discussion.

A STS is *feasible* if there exists at least one scheduling in which all tasks always meet their deadlines. A STS is *schedulable* according to a given *scheduling algorithm* if it is feasible using that algorithm.

Several algorithms have been proposed in the literature, here we refer to a broad classification which separates algorithms in three classes:

- *Fixed task priority* (FTP): such algorithms assign a priority to each task. This priority never changes during system execution.
- *Fixed job priority* (FJP): such algorithms assign a priority to each job upon release. The priority of a job does not change during its execution, however a job might be preempted if an higher priority job is released. Earliest deadline first (EDF) is an example of FJP algorithm. In EDF the maximum priority is assigned to the job with the closest deadline. It has been proven that EDF is optimal for STSes - i.e., it is always possible to schedule a feasible STS using EDF.
- *Dynamic priority* (DP): such algorithms do not assign a fixed priority, which can be always changed according to the current state of the system.

From these definitions, it follows that DP algorithms are a generalization of FJP algorithms, which are in turn a generalization of FTP algorithms.

This thesis is mostly focused on a particular set of systems, having some real-time requirements. An application has real-time requirements if it is required to complete its workload in a predefined time, that is the deadline already defined above. The strictness of such real-time requirements depends mostly on the type of application. Usually, real-time requirements are classified in three classes, plus a fourth class for non-real-time applications:

1. *Hard real-time* (HRT): this is an absolute requirement that each job finish within its deadline. This type of requirement is used when a missed deadline can have catastrophic consequences. For instance, it is quite common in control applications, in which a missed deadline might cause the controlled system to diverge, causing a loss of functionality which may result in loss-of-mission or loss-of-life/injuries to human users.
2. *Firm real-time* (FRT): this is a requirement that each job finish within its deadline, otherwise results of its computation are considered useless or misleading. This type of requirement is often used in communication applications that can

tolerate loss of a few frames. Late frames are dropped rather than processed late.

3. *Soft real-time* (SRT): this is a requirement that each job finish within its deadline or soon-after. It is often used in streaming applications and voice-over-ip (VoIP) applications, in which a late frame might cause a degradation of quality but a still understandable communication, provided that it is not too late.

4. *Best effort* (BE): this class includes all applications without real-time requirements.

2.1.2 Mixed criticality Scheduling

An extension to the task model described in Section 2.1.1 is proposed in [41]. Here, the author considers a FTP scheduling algorithm to be extended with the notion of criticality. The original proposal was extended to FJP and DP algorithms in [43], which also proposes a more formal analysis of the algorithm proposed in [41]. In [41], the task model is extended to add a criticality level L_i for a task τ_i. Such level is a confidence level for the WCET estimation of the given task. Moreover, each task is assigned a set of WCET estimations C_il where $l \in \mathscr{L} = \{A,B,C,D,E\}$. Such estimations are performed according to the state-of-the-art techniques for WCET estimation, which are out of the scope of this thesis and are not further discussed. It should be assumed that$D_i \leq T_i$ and that $C_{iA} \geq C_{iB} \geq C_{iC} \geq C_{iD} \geq C_{iE} \forall i$

The main target of this analysis is to prove that the task τ_i does not miss its deadline with a level of confidence L_i. This can be achieved modifying the preemptive fixed priority (PFP) scheduling. In PFP, which belongs to the class of FTP algorithms, each task is assigned a priority level ρ_i. At any given time, the processor is executing the highest priority task among those dispatched but not completed. If $\rho_i < \rho_j$, task τ_i has an higher priority than task τ_j. The analysis can be performed by using 2.1 to compute the response time of task τ_i knowing the execution time, and the period or interarrival time of all tasks with an higher priority.

$$R_i = \sum_{j:\rho_j \leq \rho_i} \left\lceil \frac{R_i}{T_j} \right\rceil C_{jL_i} \tag{2.1}$$

In 2.1, the computation time of all tasks τ_j with $\rho_j < \rho_i$ at level L_i is used to computed the response time of task τ_i. The equation should be computed recursively by first assigning $R_i = C_{iL_i}$ and then repeatedly computing the right hand side to update R_i, stopping either when there are no tasks left in the set of tasks with highest priority of when $R_i > D_i$, in which case the system is not schedulable with the given priority assignment. A priority assignment algorithm based on period transformation is proposed in [41], implementing a FTP scheduling algorithm which is proved to be optimal for MCSTS in the FTP class.

Starting from the model proposed in [41], authors of [43] provide a more formal analysis of scheduling approaches for MCSTSes on uniprocessor systems. They also propose several methods to prove feasibility of a MCSTS.

It should be noted that this model is a generalization of the previously existing model for traditional sporadic tasks systems (STS) on uniprocessors as described in 2.1.1, since that model can be obtained assuming $L_i = L_j \forall i, j$. In [43], one of the key concepts is the *corresponding traditional STS*. Given a MCSTS, its corresponding traditional STS is $\bigcup_{\tau_i \in \tau} \{C_i(L_i), D_i, T_i\}$.

By applying the task model described above and considering existing scheduling algorithms for traditional sporadic task systems, authors of [43] enunciate a series of theorems on the feasibility and schedulability analysis of MCSTS. For instance, since for any MCSTS there exists a corresponding traditional STS, any MCSTS is feasible if the corresponding traditional STS is feasible. Moreover, they prove that EDF is non-optimal for MCSTSes and that there exists MCSTSes which are schedulable with an FTP algorithm but are non-schedulable with EDF, and viceversa. Furthermore, there exist MCSTSes which can be scheduled by a DP algorithm which are not schedulable by any FTP or FJP algorithm.

In the new algorithm proposed in [43], more than one task can be assigned a given priority, whereas in the *vanilla* Vestal algorithm, each task had a unique priority. The assignment proceeds by simulating the SAS of the currently assigned priorities. During this simulation, tasks in τ_{hi} are assumed to have a higher priority than tasks in τ_{curr}, but tasks in τ_{hi} do not have a specific priority level yet. Each time a deadline miss is detected during such simulations, the task that generated the deadline is promoted to a higher priority. These simulations are repeated for each criticality level in the system, using the WCET estimate corresponding to the current criticality level at each step. The algorithm terminates either if the set τ_{hi} is empty at the end of all simulations, meaning that all tasks have been assigned a priority level, or if the set τ_{curr} is empty, meaning that all tasks should be promoted to a higher priority, therefore the system is not schedulable.

Authors of [43], proved that the proposed priority assignment algorithm dominates both EDF and Vestal algorithm - i.e., all algorithms that are schedulable by EDF or Vestal algorithm are also schedulable by Algorithm **??**. However, the algorithm is not optimal in the FJP class.

A first attempt of porting results of [41] and [43] to multi-core systems was proposed in [44]. This paper describes a first attempt at what can be classified as a time-division approach to avoid interference among tasks of different criticality levels, while keeping an acceptable utilization of the processors focused around the implementation of a module for a real-time version of the Linux kernel.

The proposal in [44] was extended in [45], where it was applied to the use-case of an avionic system. In this proposal, the system is composed of applications in 5 criticality levels, corresponding to those defined in the RTCA DO-178B standard, which was the version of the standard more recently replaced by the DO-178C discussed in Section 1.4. Since the WCET analysis for tasks of high-criticality is pessimistic, it is often the case that high-criticality tasks do not completely consume their time budget. To increase utilization, authors of [45] propose to use the slack generated by high criticality tasks to schedule lower criticality tasks. In this approach, each time a task is released, it receives a time budget. Tasks that complete before spending all the assigned budget are defined as *ghost* tasks. A scheduler that selects a ghost task is suspended until the time budget of the ghost task is depleted or an higher priority task at the same criticality level is released. By doing so, execution of lower criticality tasks is allowed despite being still in the time allotted to a high criticality task.

2.1.3 Mixed Criticality Scheduling and Resource Sharing

When more than one task share a single hardware platform, they share some resources. The most basic shared resource, and also the most problematic from a real-time scheduling perspective, is the system interconnect, by which all peripherals and shared levels of the memory hierarchy are accessed by all CPUs. Several proposals where made for scheduling MCAs taking into account the interference produced by resource sharing.

The problem of the interference among tasks of different criticality levels was tackled in [46], where an online monitoring of the execution time was proposed. The methodology requires a WCET analysis to be performed on each task. The WCET estimations are then used to configure a dedicated hardware, which at any time knows what task is running on the system and what is its WCET. If a low criticality task violates its WCET it is interrupted to allow execution of high criticality tasks.

In [47], the main idea is that at any given time, all cores run programs at the same criticality level, mixing this with a cyclic executive approach. This solution is well suited for certification in avionic applications, although it imposes the non-trivial requirement of granting that at any given time all programs running in the system belong to the same criticality level. The task allocation issue is dealt with in [48], where the scheduling proposed in [47] was paired with the issue of task mapping to obtain a complete mixed-criticality system (MCS) methodology. This proposal still requires that at any given time, all cores are running programs at the same criticality level. Other solutions also introduced the concepts of fault-tolerance in the scheduling algorithms used in MCS [49]

One core issue in MCAs deployed on MPSoCs is the temporal interference among tasks running on the different cores; several solutions have been proposed in the years to deal with this problem. Overall, all solutions can be classified as in [50]:

- *Time multiplexing*: solutions in this class use an offline analysis approach to schedule the system taking into account the interference produces by concurrent execution.

- *Bounded interference*: solutions in this class are focused on finding or imposing a bound to the interference that tasks can exert on each other.

Time Multiplexing Solutions

Most time multiplexing solutions require an offline analysis of the tasks in order to properly schedule the system. Many approaches split tasks in phases during which a task can access shared resources - access phases - and phases during which access to shared resources is forbidden - local phases. Access phases are never

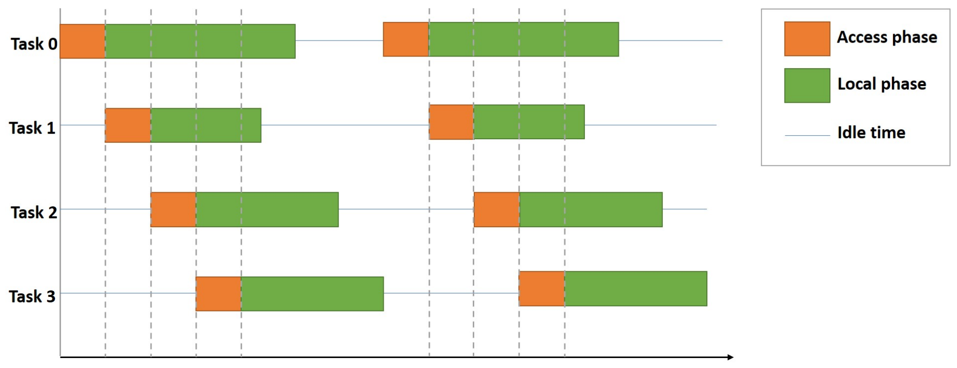


Fig. 2.1 Example of a four tasks system scheduled on a quad-core processor with time multiplexing approach.

scheduled in parallel on different cores, i.e. while one core is running a phase containing accesses to shared resources, the other cores must be running phases that only use local resources [51][52]. In [53], access phases have the additional requirement that no code external to the application should be executed, therefore, system calls or interrupt service routine cannot be executed during access phases. All time multiplexing approaches share the common flaw of under utilization of the hardware platform, due to the introduction of idle time during access phases, as depicted in Fig. 2.1. Fig 2.1 represents the time diagram of the execution of four tasks on four cores scheduled with a time multiplexing approach. In the depicted scenario, memory is the shared resource. In Fig. 2.1, the two basic phases, access and local, are considered. It is evident from this time diagram, that such a scheduling introduces a non-negligible amount of idle time.

Bounded Interference Solutions

Bounded interference solutions include methods to evaluate a maximum interference among tasks of a MCS on a multi-core system. In most cases, such interference are evaluate in a worst-case scenario, therefore bounded interference solutions can introduce idle periods as well. Such idle periods can be used in a slack-scheduling scheme to increase system utilization, reaching utilization rates much higher than in time-multiplexing solutions. The notion of interference can also be used to improve WCET estimation, as proposed in [54]. The method proposed is named *interference sensitive worst-case execution time* (isWCET). In isWCET, the access latency to a shared resource is computed considering the number of concurrent accesses that

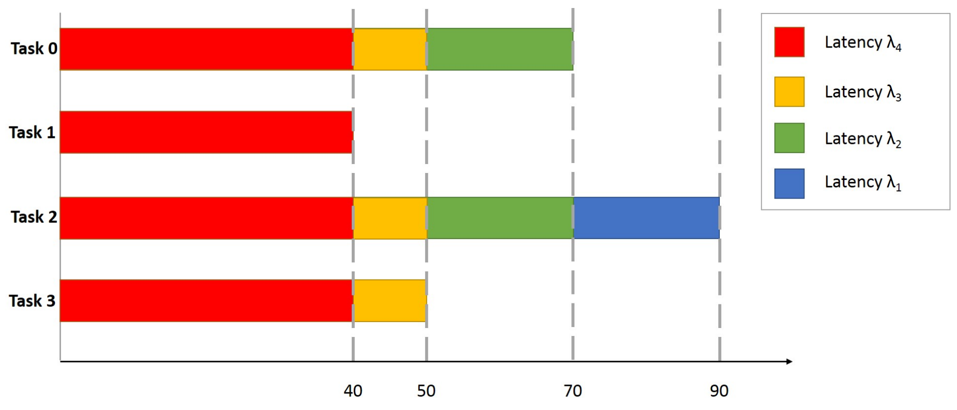


Fig. 2.2 Example of application of isWCET to four tasks on a quad-core system

can occur during execution of the system. An example of application of isWCET to a quad-core system is depicted in Fig. 2.2. The key concept is that the number of accesses to a shared resource by each task is known at analysis time. The requests are considered as starting all in parallel at time T_0. In a set of tasks as in Tab. 2.1, the first 40 accesses by all tasks should be considered as having the maximum latency λ_4, as experienced with four concurrent accesses to the shared resource. After this number of accesses, task τ_1 has performed all the necessary accesses, therefore, the next 10 accesses are considered as having latency λ_3. After these first 50 accesses, task τ_3 has completed its accesses as well, therefore the next 20 accesses of tasks τ_0 and τ_2 are accounted for with a latency λ_2. Finally the last 20 accesses for task τ_2 are considered to have the minimum latency λ_1.

The isWCET approach is a bounded interference approach, and can be combined to a slack-scheduling solution as proposed in [55]. The computational resources that are not used when scheduling a system using isWCET, can be reallocated to other tasks. Performance counters can be used to implement a safety-net that counts the number of accesses to a shared resource by each task. When a task completes the scheduled accesses, the slack time can be used for additional computation. The same safety-net can also be used to detect errors: if a task exceeds the scheduled number of accesses to a shared resource, all future attempts to access that shared resource by that task are blocked, pending intervention of a recovery mechanism.

Table 2.1 Task system used as example of application for isWCET

Task	Accesses to shared resource
τ_0	70
τ_1	40
τ_2	90
τ_3	50

The solution in [56] can be classified as a bounded interference as well. This solution relies on an off-line computation of the interference bounds and on an online mechanism for enforcing such bounds. The run-time bound-enforcing mechanism is based on a server mechanism which is in charge of providing the budgeted accesses to a shared resource for each core in the system. This solution requires one server per shared resource; moreover, it requires that at least one core in the system is used to run such servers instead of functional workload.

Another approach to the solution of the shared resource problem has been proposed in [57] and [58]. In this solutions, authors propose the implementation of monitoring approaches based on modeling the real time behavior of the system and ensuring that the actual behavior respects the specification as the model does by observing and enforcing given properties of the system, such as task activations. The implementation is described as software modules, one for each property to be monitored, integrated in the operatying system. The work proposed in this thesis, also applies the monitoring approach, but the proposed approach is more generic that the one just described, and does not require a new implementation for any given property, as described in more detail in the next chapter.

2.2 Commercial-Off-The-Shelf Components in Space Applications

In space applications, radiations have traditionally been a great concern for designers. Most of the research in fault-tolerance cited in Section 1.5.1 was promoted by the space industry's interest.

In the past, space applications would be developed using specially designed hardware. Such hardware would employ special packages - e.g., ceramic packages - and other design precaution to improve resistance to radiation. A complete survey of such designs is out of the scope of this thesis. According to the level of precautions taken during design of such hardware, two types of space-graded hardware are recognized in this thesis and in much of the relevant literature:

1. *Radiation hardened* (rad-hard): such hardware is realized in such a way that radiations have little effect, if at all. Rad-Hard hardware would employ realization technologies proved to be more resistant to TID than other technologies - e.g., full-depletion silicon-on-insulator (FD-SOI) - together with design patterns such as TMR.
2. *Radiation tolerant* (rad-tolerant): such hardware uses technologies that are more resistant to radiation effect than other counterparts. For instance, rad-tolerant field-programmable gate arrays (FPGA) employs read-only memories (ROM) or flash memories for their configuration memory as opposed to the static random access memory (SRAM) often used by commercial-grade FPGAs.

However, such hardware has a longer development and qualification cycle with respect to commercial-grade hardware; moreover, solutions like TMR increase the silicon area required to implement the hardware. As a consequence, rad-hard or rad-tolerant hardware is often well behind the technology cutting-edge in terms of performance granted to system designers. When paired with increasing performance requirements of new space applications, and the everlasting quest for reduced SWaP consumption by equipment, such considerations pushed designers towards different solutions.

The logical step to increase performance of space equipment is using newer hardware. However, such hardware would not be space-graded. Therefore, a space system using such hardware must take more precautions in order to be allowed on an actual spacecraft.

An early example of a space computer based on commercial-off-the-shelf (COTS) components, was the Maxwell SCS750 computer [59]. The SCS750 is based on a TMR implementation. The CPU is a single-core 800 MHz PowerPC 750FX processor. The SCS750 uses three instances of the CPU, each running the same program,

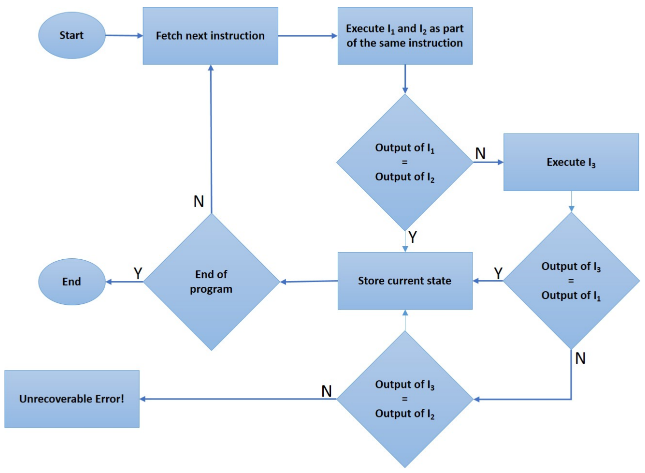

Start
Fetch next instruction
Execute I_1 and I_2 as part of the same instruction
Output of I_1 = Output of I_2
N
Execute I_3
Y
N
End of program
Y
End
Store current state
Y
Output of I_3 = Output of I_1
N
Unrecoverable Error!
N
Output of I_3 = Output of I_2

Fig. 2.3 Temporal TMR as implemented by the Proton-100k pre-processor.

receiving the same inputs. Each replica is connected to the same companion chip. The companion chip should be a rad-tolerant component, implemented either on an FPGA or an application-specific integrated circuit (ASIC). However implemented, the companion chip acts as a memory controller with error correcting code (ECC) and as a voter for the TMR. The output of the processors is processed by the voter before being forwarded to the memory controller. This architecture has been employed, among others, in cited European space agency (ESA) global astrometric interferometer for astrophysics (GAIA) mission, which is currently in orbit and fully operational. Nonetheless, the SCS750 architecture has some issues that make it obsolete in the MPSoC era. First of all, it still relies on single-core processors without an on-chip memory controller. This kind of processor architecture is almost completely out-of-market, mostly due to the advent of SoC which often include a memory controller to improve both performance and power consumption. When using such SoC architecture, it is impossible to interpose a companion chip between the processor and the memory controller; this makes the SCS750 architecture incompatible with MPSoCs.

Another example of space computer using COTS is the Proton-100k computer [60]. The Proton-100k uses a temporal TMR (TTMR) approach to modify software running on a very long instruction work (VLIW) processor. In this scenario, each instruction can be duplicated, and the two copies run in parallel on the multiple execution units in the processor, implementing the same workflow as in Fig 2.3. To help the designer, the software transformation can be executed by a dedicated pre-processor. The unrecoverable error state in Fig. 2.3 can only be reached if two separate SEE cause two faults, one in the first two executions and one in the third execution. Such event is extremely unlikely and can be neglected. However, should this state be reached, a flag can be raised to signal to error to the user. The Proton-100k also employs a companion chip, the H-Core. The H-Core is used to detect SEFIs and therefore should be implemented on a rad-tolerant chip.

In the same period, the duplex multiplexed in time (DMT) architecture [61] was proposed. DMT requires that the program is partitioned into three phases: input, processing, and commang generation or output.

The input and the processing phase are executed twice. Results of both repetitions are compared before continuing execution. The final execution flow is: input phase 1, input phase 2, input comparison, processing phase 1, processing phase 2, processing comparison, commands generation or output.

Execution of the duplicated phases is not physically isolated, therefore they can suffer common mode errors as a consequence of data corruption. To avoid such occurrence, each execution has its own dataset, which is separated and stored at different memory locations. Some of the fault-detection and tolerance functionalities are delegated to an external companion chip, which should be implemented on a rad-hard component. The companion chip implements ECC, memory segmentation, and safe context storage. The safe context is used to recover functionality of the system in case of a fault. When a fault is detected the last safe context is restored and execution resumes from there. The *dual duplex tolerant to transients* (DT2) [61] architecture uses a duplicated processor/memory subsystem to implement the same concept of the DMT architecture. It also requires additional functionalities in the companion chip to synchronize the two replicas. A more recent instance of space computer based on COTS components has been presented in [62], representing some of the results obtained during ESA's *HiRel* program. The architectural block diagram is represented in Fig. 2.4 The prototypical computer for this architecture is based on a single-core powerPC (PPC). A memory controller and high-speed communication controllers (spacewire and high-speed serial link) are implemented on a COTS FPGA, whereas a second FPGA implements low-speed communications and house-keeping functionalities. The system implements a software transformation similar to that in DMT, but it also adds memory protection configuration at the boundaries between phases to enforce isolation of execution. Moreover, it can be configured to use either

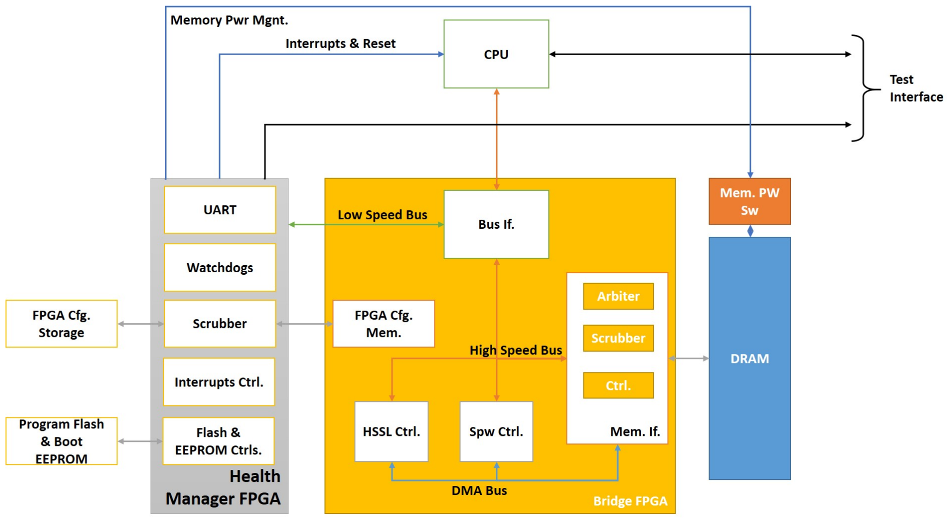


Fig. 2.4 Architectural block diagram of the ESA HiRel Computer

a backward recovery similar to that in DMT or a *forward recovery*, implementing a TTMR scheme. To improve fault-detection, the system employs a WDP and a WDT. The WDP is used to check compile-time signatures of blocks, and triggers a recovery action when the signatures are not received in time or are not received in the expected order. The WDT is a windowed WDT, therefore it is able to detect two conditions:

1. the execution is taking too long;
2. the execution ended too soon.

When one of the two conditions is detected, a recovery action is triggered. The proposed system architecture was evaluated through fault injection experiments; results showed a high reliability, fulfilling ESA requirements for next generation satellites.

Chapter 3

The ODIn Architecture

3.1 Safety Guidelines for Multi-Core Use in Avionic Applications

The main contribution in this thesis is the development of a novel architecture for implementation of MCSes on MPSoCs. The need of such a reference architecture is evident when considering the complexity of such systems and the necessity of having a system able to satisfy safety requirements mandated by relevant standards as a starting point to reach a certification by competent authorities. Reports of studies funded by EASA [3] and FAA [63], arrive at the conclusion that no COTS multi-core system is suitable for avionics applications and that certification should be performed on a case-by-case basis for each new product. Such position arises from the complexity added by multi-core system architectures to the WCET estimation time. The latest CAST position paper on the topic of use of multi-core processors in airborne systems [64] identifies a set of guidelines to be followed by an OEM in designing its equipment. Such guidelines are organized in 5 objectives:

1. *Software planning*: the target multi-core system should be appropriately modeled and the software architecture should be detailed in terms of its dynamic behavior. Moreover, partitioning should be proved to be robust and all tools and methods used during software development should be carefully described. Such descriptions should include all software running on the equipment, including operating systems and hypervisors.

2. *Resource planning*: this objective deals with the stable configuration of the system - i.e., the state reached at the end of the boot process. All details of such state must be provided to obtain certification, including the clock frequency of all active cores, the cache configuration, memory allocation, memory interface configuration, etc . . . As part of this description, the OEM should also detail what are the shared resources in the system and how usage of resources is protected against excesses. Finally, the OEM should prove that once reached such configuration is stable and protected against illegal modifications, such as those effected by an error in the system.

3. *Interference channels*: since there are shared resources in the platform, there is a coupling among different applications running on the hardware. The OEM should identify all possible interference source - or channels - and should detail how the identified interference is accounted for during system design and how it is dealt with at run-time, to avoid unexpected interference as consequence of errors in the system.

4. *Software verification*: due to resource sharing and interference, software verification should be performed in a single step, without any incremental step allowed. To avoid such occurrence, which would significantly reduce development speed and increment verification costs, the OEM should prove that the platform provides strong partitioning, therefore all applications in the system can be verified in isolation.

5. *Error detection and mitigation*: due to the sharing of resources and to the concurrent execution, errors in the software execution, such as memory violations, can happen at run-time. Such occurrences should be detected and mitigated. The OEM should detail how such detection and mitigation occur.

3.1.1 Partitioning and Safety Objectives

It is possible to map all the aforementioned objectives in the broad categories of spatial and temporal partitioning. By doing so, it is possible to satisfy all objectives by solving these two relatively independent problems . The concepts of spatial and temporal partitioning have been described in Section 1.1.1, but are reported and extended here for a better fruition.

Spatial Partitioning

Spatial partitioning is the property of containment with respect to faults affecting resources shared by more than one application in the system. The fault does not affect timing properties of the interaction between an application and the resource, but it affects the correctness of the result. For instance, a faulty application might change the configuration of a shared peripheral in an unpredictable way.

Using hardware modules available in MPSoC architectures - e.g. the memory management unit (MMU) - this kind of partitioning can be implemented through a type-1 hypervisor. A type-1 hypervisor is a virtualization software that does not rely on the services of an underlying host operating system. The type-1 hypervisor implements a partitioning mechanism for virtual machines that can be used to enforce spatial partitioning.

By granting this kind of partitioning objectives 1 and 2 are satisfied, whereas objectives 3, 4, and 5 are partially satisfied.

Temporal Partitioning

Temporal partitioning is the property of containment with respect to faults affecting the timing of an application. The fault affects some shared resource in a way that change the timing properties of the interaction between an application and the resource but it does not affect the correctness of the result. For instance, a faulty application might abuse a shared communication channel, causing a delay in other applications.

When implemented on MPSoCs, applications share resources which are tightly integrated with the processing cores. Therefore, evaluation of interference impact on the applications execution time is very difficult without very detailed information on the MPSoC. This kind of information is often unavailable to users of the MPSoC, as it constitutes an important trade secret for MPSoC manufacturers. Therefore, it is almost impossible to prove safety of a system using MPSoC, which is the main reason why this technology is still seldom used in mission- or safety-critical systems.

By granting this kind of partitioning together with spatial partitioning, all objectives are satisfied.

This chapter presents the *online detection of interference* (ODIn) architecture, which enforces both spatial and temporal partitioning on target MPSoCs. ODIn is intended as a reference architecture for avionic applications that can argue all safety objectives are met by the implementation of the proposed architecture.

ODIn is a composition of two main solutions: one is mainly concerned with spatial partitioning, whereas the other is mainly concerned with temporal isolation. Both solutions are described separately before describing their integration.

3.2 Spatial Partitioning in ODIn

Spatial partitioning in the ODIn architecture is based on the use of a type-1 hypervisor to implement partitioning of the applications running in the system, as proposed in [62]. In this solution, each *resource partition* is actually a virtual machine managed by the type-1 hypervisor. The type-1 hypervisor is responsible for the scheduling of the partitions according to a cyclic executive approach compliant with the ARINC-653 suggestions in terms of partition scheduling. Each resource partition can be statically assigned to a subset of the cores available on the system.

The type-1 hypervisor enforces partitioning using the MMU available in the MPSoC architecture. By a proper configuration of the MMU, the type-1 hypervisor is able to statically assign a different set of resources to each partition. Therefore, the type-1 hypervisor is able to define a set of resource partitions R_p and to map each resource to a subset of R_p. The definition of resource partitions and the mapping of resources to resource partitions is performed by the system integrator, according to the nature of the application and of the hardware. In most cases, to satisfy safety requirements the system integrator may prefer to assign control of a resource to a partition and only to such partition. In this case, resource sharing is still possible by adopting inter-partition communication (IPC) channels. Moreover, any information shared among different partitions should be exchanged through one of such IPC channels. This would include computation results of an application running in a given partition which are needed by a different application running in a different partition. The solution in [65] had a dual-core architecture as target, however it can be extended to any number of cores. The main requirement is that the system architecture is still a bus-based architecture, since the network-on-chip (NoC) interconnect requires a different analysis which is out of the scope of this thesis.

The spatial partitioning in the ODIn architecture can be used at different granularity levels, from finest to coarsest:

1. *Each task is isolated in its own partition.* This kind of granularity requires a great number of partitions, which may even go beyond the support of the type-1 hypervisor. Moreover, this kind of granularity would also require a noticeable number of IPC channels, introducing a performance overhead.

2. *Each application is isolated in its own partition.* This kind of granularity is the most suitable for integrating a limited number of applications, like could be the case in most mission-critical systems. This is also the case considered in [65][66]. The need for IPC channels in this case is limited.

3. *Applications at the same criticality level share the same resource partition.* This kind of granularity is perhaps the easiest to use for porting existing IMA-based systems to the proposed architecture; it also reduces the need of IPC channels to a bare minimum.

The ODIn architecture requires instantiation of a special system partition which is executed at the end of the bootstrap phase and is responsible for hardware configuration necessary for granting temporal partitioning (see Section 3.3). The software running in this partition has rank of system software and should be certified at the highest level of certification in the system. This software is also responsible for implementation of some of the recovery actions implemented in the ODIn architecture. In this role, the software in the system partition is responsible to manage the pagefault exceptions raised by the MMU when a partition tries to perform and illegal access to a resource. In this context, an access is illegal if the resource does not belong to the same resource partition as the software attempting to perform the access.

The recovery actions implemented by the system software are better detailed in Section 3.3.6, together with the trigger conditions of each action.

3.3 Temporal Partitioning in ODIn

The temporal partitioning implemented in the ODIn architecture is quite complex and is based on a method for online detection of interference between tasks running on an MPSoC. This method is independent on the scheduling approach adopted in the system development; furthermore, this method does not require any knowledge of the access latency to shared resources, nor a special WCET estimation technique.

The method described in this section, should be classified among bounded interference approaches as described in Section 2.1.3. The method is composed of two main phases: an offline phase and an online phase. During the offline phase a bound to the interference is computed for each shared resource and each task. The online phase implements detection and recovery mechanisms for unexpected interferences. The offline phase can be considered an extension of the profiling and WCET estimation phases of the design; it can use the same data as well to compute a set of three thresholds which are used during the online phase. The online phase should be considered a safety-net as in [50], since it monitors the usage of a shared resource by a given task. To perform such monitoring, performance counters are used during the online phase.

The analysis of the offline phase can be performed at different granularity levels. The online phase can work with different granularity levels as well, but the same granularity should be used for both phases. Two granularity levels are identified:

1. *Core granularity*: it is the coarsest level. At this granularity level, the unit of monitoring is the core rather than the task. This requires that each task is statically assigned to one core and that the accesses of all such tasks to the shared resource under scrutiny are summed to provide the total number of accesses to that resource by the monitored core.

2. *Task granularity*: it requires to analyze each task separately and that the performance counters used during the online phase are part of the task context managed by the underlying operating system. This means that performance counters values should be saved and restored at each context-switch.

This section is organized in subsections that describe each step of the process to apply the proposed method. Each subsection describe the theory of the method and then proposes a practical example. By collating practical examples in all subsections, the reader should be able to obtain a complete example for application of this method.

Algorithm 3.1 Stressor application

```
1: procedure STRESSOR(L1_line_size, data)
2:     v ← data/L1_line_size
3:     while 1 do
4:         for i = 0 → $v$ do
5:             for j = 0 → length(data) step L1_line_size do
6:                 data[$i+j$] ← data[$i+j$]
7:             end for
8:         end for
9:     end while
10: end procedure
```

3.3.1 Benchmark Application

The whole practical example is based on a synthetic benchmark application. The synthetic benchmark was created to provide a realistic example considering the memory as a shared resource in conditions that mimic a real use case. The memory hierarchy is a very critical shared resource, since it is shared among all applications in the system.

With this objective in mind, two memory bounded applications were implemented as components of the benchmark:

Stressor: a synthetic memory bounded program performing consecutive memory accesses, of both read and write type. Each access is designed to force a cache line eviction, therefore canceling the masking effect the level 1 (L1) cache might have with respect to the rest of the memory hierarchy. This program is described in Algorithm 3.1.

Bubble sort: performs sorting of an integer array [67], bigger than the L1 data cache, again to avoid the masking effect of that level of cache. This program is described in Algorithm 3.2.

Nominal interference - i.e., expected interference due to nominal system behavior - is represented by bubble sort tasks. The stressor task plays the role of the victim task. The aggressor task is implemented by a slight modification of the stressor task. Whereas the nominal stressor task waits at the end of execution until it is released again - i.e., it behaves as a periodical task in a real-time system - the aggressor task does not wait at the end of execution. This behavior emulates a bug that causes an

Algorithm 3.2 Bubble Sort

```
procedure BubbleSort(unsorted)
    sorted ← {}

    copy(sorted, unsorted)
    repeat
        swapped ← 0
        for i = 0 → length(sorted) do
            for j = i → length(sorted) do
                if sorted[i] > sorted[j] then
                    swap(sorted[i], sorted[j])
                    swapped ← 1
                end if
            end for
        end for
    until not swapped
end procedure
```

application to enter an infinite loop. The design of the stressor application, which causes a burst of memory access at each step, stresses the memory hierarchy and enhances observability of interference.

Both the stressor and bubble sort were selected to be representative of an avionic workload. Algorithm 3.1 emulates a control application, reading data from sensors, performing some computation, and writing back results of such computation to actuators. Algorithm 3.2 represents a memory bounded application, such as a vision algorithm or a logging application.

The benchmark application is to be deployed on a MPSoC. In this example, the target MPSoC features an ARM Cortex-A9 processor in two variants: a dual-core and a quad-core processor. Details of how the benchmark application is deployed on an actual target system are provided in Chapter 4. However, it is useful to define the target architecture for this example in order to provide a guidance in the metric selection.

3.3.2 Metric Definition

In the scope of this theses, the interference is any phenomenon that is able to induce a misbehavior in a monitored task. If not corrected, such misbehavior would cause a system failure (see Section 1.2). The main focus in this section is on temporal interference, i.e., interference that can cause an excessive execution time, with respect to the estimated WCET.

An *interference metric* - also simply *metric* - is a quantity that is measurable at run-time and is sensible to interference, i.e., its measurement changes when the monitored task is affected by an interference. Interference metrics can be aggregated using any convenient mathematical application. The results of such composition is a *compound metric*, which is made of *component metrics*.

A compound metric is well-defined if it satisfies all requirements of an interference metric as stated above: it must be measurable at run-time, and it must be sensible to interference. A compound metric is measurable at run-time if and only if all its component metrics are also measurable at run-time. A compound metric is sensible to interference if at least one of its component metrics is sensible to interference. Different component metrics might be sensible to different sources of interference.

3.3.3 Metric Selection

Since a metric is defined as a run-time-measurable quantity, it is necessary to identify a suitable measuring tool for a candidate metric before being able to use it. A measuring tool can be either an ad-hoc piece of hardware added to the system with the sole purpose of measuring the interesting quantity, or it can be some hardware component already in the system for other purposes. An example of such hardware are the performance counters. Performance counter are available on most MPSoCs to help designers during profiling activities.

Performance counter have been designed to be used during profiling and debug; nonetheless, they can be used as a safety-net in a real-time context. Each architecture has different performance counters which are able to measure different quantities for performance evaluation [68][69]. However, there are a few categories of performance counters that are generally available on any architecture:

Cycle counters: these are incremented at each processor clock cycle. Some can be configured to count a clock divided by a configurable amount n.

Cache hit/miss counters: these are incremented at each cache hit or cache miss. Often, there are separate counters for the instruction and data caches.

Data read/writes: these are incremented at each read/write operation in data memory. Here, data memory includes data caches and main memory.

Branch prediction miss: these are incremented each time the branch prediction unit performed a wrong prediction. In many cases, there are separate counters for different mis-prediction - i.e., one counters counts each time a branch that was predicted as not-taken is taken and a different counter counts each time a branch that was predicted as taken is not taken.

Exception taken/return: these are two different counters. The exception taken counter is incremented each time an exception handler or an interrupt service routine are executed. The exception return is incremented each time an exception handler or interrupt service routine is completed.

Any quantity measured by a performance counter can serve as an interference metric, provided they also satisfy the requirement of sensitivity to the interference. The selection of the metric should be performed by the designer also taking into account the nature of the application, in order to select the most relevant metrics.

When an interference between two tasks occur, each task can be either a victim, or an offender. The offender task is the task that is generating an interference, e.g., by using a shared resource in an improper way. The victim task is the task that is suffering effects of the interference without being responsible for its generation. Metric selection should be performed considering the difference between an offender and a victim task as well. For instance, an offender task can be detected by observing that it is performing more accesses to the shared resource than it is expected; on the other hand, a victim task can be identified by observing that the access latency to a shared resource is longer than expected.

Practical Example

Here it is described how a metric selection can be performed starting from knowledge of the application, and how the choice can be validated by checking that the selected metric satisfies the definition of interference metric.

First of all, the objective of the method application has to be defined in order to select a proper metric. To define an objective, the designer should consider the tasks one at a time and define what are the objectives for the monitor applied to each task. Such objectives can be:

1. to identify the monitored task as an offender, or
2. to identify the monitored task as a victim.

In this example, we chose a monitored task and define the objective of identifying the monitored task as a victim.

The first step in the application of the proposed method, is the identification of a metric. The definition of metric provided in Section 3.3.2 indicates that the metric should be a quantity to be measured at run time and that is sensitive to interference affecting the monitored task. To identify a task as a victim, it is necessary to detect interference generated by other tasks in the system. In this example, this occurrence is due to other tasks in the system abusing of their access to the memory hierarchy and to the interconnect; this abuse causes a delay in the execution of the monitored task due to increase in its access latency to the shared memory.

A metric suitable to identify this case, would be a metric measuring directly the access latency of a memory operation. However, such metric is not available in most architectures. The target architecture does not provide such a direct metric. Analyzing the available performance counters and knowing the characteristics of the benchmark application, the data cache-dependent stall cycles (DCSC) is a good candidate to detect the interesting interference in a victim task. DCSC is defined as the number of clock cycles during which the processor is stalled waiting for data to be provided by the memory subsystem.

DCSC is surely measurable at run-time, therefore it satisfies the first requirements of the definition in Section 3.3.2. However, it is still necessary to prove that it is sensitive to interference. This goal can be achieved by using an experimental approach. This approach requires that the DCSC metric is measured during the profiling phase considering two scenarios:

1. while the monitored task is running alone;
2. while the monitored task is running concurrently with other tasks in the system.

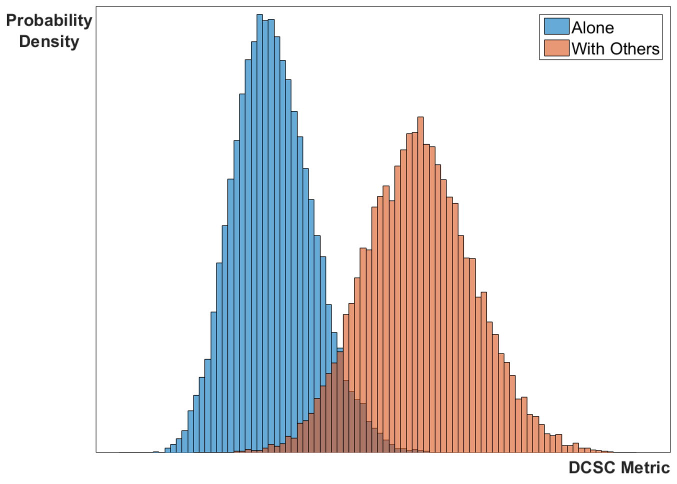


Fig. 3.1 Comparison of measurements performed with the monitoring task running alone, or with other tasks in the system. Each dataset consists of 15000 measures taken on as many executions. In the *alone* scenario, the monitored task was running alone in the system. In the *with others* scenario it was running with one bubblesort task on each of the other cores in the system.

If the comparison is executed *ceteris paribus*, the difference in the measurements is due to the interference among tasks, therefore the selected metric is sensitive to interference and satisfies the metric definition in Section 3.3.2. Results of such comparison on the target system with the benchmark application are presented in Fig. 3.1. A further description of such results is proposed in the next chapter.

3.3.4 Metric Characterization

Interference metrics are often non-deterministic, due to several factors, including - among others:

- expected interference from other tasks,
- non-deterministic execution order in superscalar architectures,
- non-deterministic memory access sequences due to bus arbitration and/or complex memory controllers implementing,
- architecture-dependent factors.

Therefore, a statistical characterization of the selected metric should be performed in order to apply the proposed method. More precisely, the probability density function (PDF) of the selected metric should be obtained.

In most cases, the PDF should be computed starting from profiling data, since it is not known a priori. Therefore, profiling data should be fitted to a known PDF. This step should be performed, in practice, by using one of the statistical analysis software tools available on the market. Such tools make available complex statistical analysis and can be scripted to extract the best-fitting PDF for the available data.

Practical Example

The metric selected as explained in Section 3.3.3 is here characterized using the theory described above.

The first step in metric characterization is gathering a sufficient number of measures. This number depends on the goodness-of-fit test selected for testing the data against a set of known PDF or to a non-parametric distribution [70]. In this example, the selected goodness-of-fit test is the Anderson-Darling test [71] and its k-sample, non-parametric version [72]. Such tests are supported by software tools available on the market. In this example, the used software is Mathworks' Matlab. The software is used to perform the statistical analysis described here.

The Anderson-Darling test requires a little as 8 samples to provide a significant result. Since the data to be used in this phase is gathered during profiling phase, it is safe to assume that available data satisfies this cardinality requirement. Data gathered in this example was fitted to a normal distribution as in Fig. 3.2. The fitting for both the normal and the non-parametric distribution showed in the figure was performed using an Anderson-Darling test with a confidence level of 0.001. The result showed that the data fitted a normal distribution, therefore the analysis can go on to the next step.

However, should the data not fit a normal distribution, the designer can use any other known distribution, using a non-parametric distribution as a last resort [70][73]. In Fig. 3.2 it is showed how a non-parametric distribution with a normal kernel fits the data using an Anderson-Darling goodness-of-fit test with confidence level of 0.001.

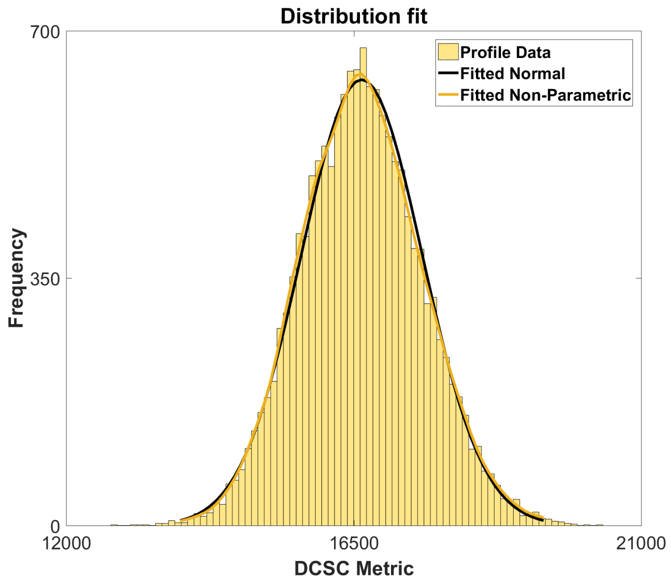


Fig. 3.2 Data was fitted to a normal distribution and to a non-parametric distribution using Matlab's statistical toolbox. Both distributions where test using an Anderson-Darling goodness-of-fit test. The confidence level in both tests was 0.001.

In the rest of this example, the fitting normal distribution will be used for any further analysis.

3.3.5 Thresholds Determination

The offline phase has the main goal of identifying a set of thresholds to be used at run-time to detect a faulty situation. First, two thresholds are defined:

1. *Detection Threshold T_D*: a fault is detected in the system if any run-time measure of the metric is above this threshold. Depending on the observed metric, this means that the observed task is either an offender or a victim.
2. *Warning Threshold T_W*: a measure above the warning threshold but below the detection threshold can be the result of a fault active in the system or can be due to a fringe case in the expected interference. Further observation would be needed in this case.

Both thresholds are defined starting from confidence levels C_D and C_W respectively. Confidence levels should respect the only requirement that $C_D < C_W$, therefore, they can be arbitrarily set by system designers. However, the designer should keep in mind that detection accuracy and performance overhead are directly linked to the selected confidence level. Moreover, as can be expected, accuracy and performance overhead are inversely proportional. Too strict confidence levels increase the detection rate, but also significantly increase the performance overhead, due to the occurrence of false positives.

If X is a random variable describing the selected metric, then T_W and T_D are defined as:

$$\begin{aligned} P(X \geq T_D) &\leq C_D \\ P(X \geq T_W) &\leq C_W \end{aligned} \tag{3.1}$$

Since it is required that $C_D < C_W$, it follows that $T_W < T_D$. The cumulative distribution function (CDF) of a random variable X is defined as a function $F_X(x) = P(X \leq x)$, therefore definitions in 3.1 can be restated in terms of CDF:

$$\begin{aligned} F_X(T_D) &\geq (1 - C_D) \\ F_X(T_W) &\geq (1 - C_W) \end{aligned} \tag{3.2}$$

From the relation between PDF and CDF - reported in 3.3 as a reminder - of a random variable, the computation of the thresholds can be computed numerically solving 3.4 and 3.5 respectively for the detection and warning threshold.

$$F_X(x) = \int_{-\infty}^{x} f_X(\lambda)d\lambda \tag{3.3}$$

$$F_X(x) = \int_{-\infty}^{x} f_X(\lambda)d\lambda = 1 - C_D \tag{3.4}$$

$$F_X(x) = \int_{-\infty}^{x} f_X(\lambda)d\lambda = 1 - C_W \tag{3.5}$$

The same software tool used during the analysis in Section 3.3.4 can be used to solve 3.4 and 3.5.

The warning range is defined to detect situations that may be consequence of a fault being active in the system. However, the probability $P(T_W \leq X \leq T_D) =$

$C_W - C_D$ is still high enough that $[T_W, T_D]$ is high, therefore this range contains values that can still be just consequence of the expected interference. Nonetheless, if the metric value falls in this range for too many consecutive measurements, an faulty situation can be inferred and detected. Therefore, this range is a *warning range*. To discriminate between a tolerable condition and an active fault in the system bringing the value of the metric in this region, a third threshold is defined. This α threshold is defined as:

$$[P(T_W \leq X \leq T_D)]^{\alpha} = (C_W - C_D)^{\alpha} = C_{\delta}^{\alpha} \tag{3.6}$$

From the definition, follows that α can be computed as:

$$\alpha = \left\lceil \frac{ln(1 - C_G)}{ln\, C_{\delta}} \right\rceil \tag{3.7}$$

with C_G a confidence level for α. From equation 3.7, follows that values of C_G close to 1 result in high values for α. The α threshold represents the maximum number of consecutive measure that can follow in the warning region without a fault being active in the system.

Most metrics gathered during profiling activities, would be described by a normal distribution. Using properties of the normal distribution, thresholds definitions can be simplified as in the following equations.

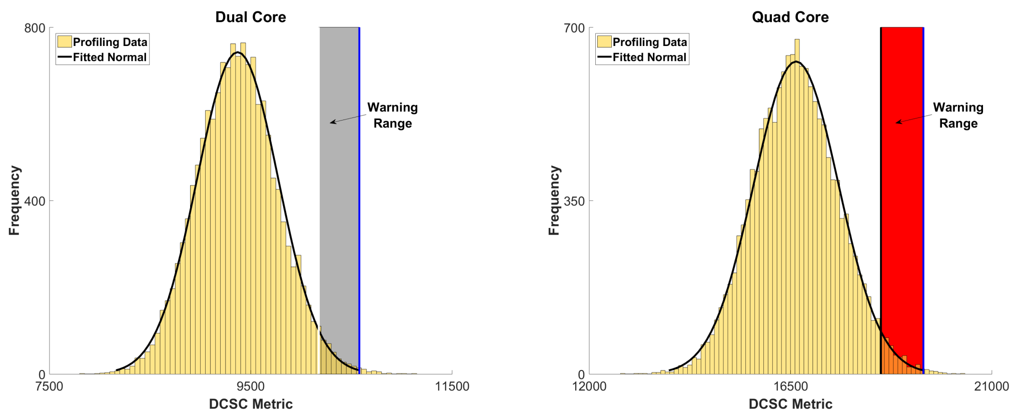
$$T_D = \mu + 3\sigma \tag{3.8}$$

$$T_W = \mu + 2\sigma \tag{3.9}$$

$$\alpha = \left\lceil \frac{ln(1 - C_G)}{ln(\Phi(3) - \Phi(2))} \right\rceil \tag{3.10}$$

Practical Example

The first step in threshold computation is to identify adequate confidence levels. This selection should be performed by the designer according to relevant experience and



(a) Thresholds evaluation on a dual-core MPSoC (b) Thresholds evaluation on a quad-core MPSoC

Fig. 3.3 Data gathered during the profiling phase, with fitter normal distribution, T_W (warning) threshold, and T_D (detection) threshold. The warning range is delimited by the T_W threshold on the left and the T_D threshold on the right.

sensibility. A good rule of thumb is to start with a confidence level $C_D = 0.05$ and adjust according to expected false positive rates. This confidence level is directly linked to detection rate and false positives. A small confidence level is linked to less false positives; however, a small confidence level is also linked to a lower detection rate. A good trade off to solve this conflict could be to select a small C_D and a large C_W, which results in a large warning region. However, such a choice, while keeping a good detection rate and a low false positive rate, would also introduce an error latency due to the fact that an error detected through the α threshold - i.e., an error that causes the metric value to be measured in the warning region for more than α consecutive times - could be detected earlier if the detection threshold is computed with a larger C_D. All such considerations are dependent on the application at hand, therefore, there is no general rule and the designer should select the best trade-off for the considered application. The third threshold, C_G) is, by definition, the probability that α consecutive measurements find a value of the metric within the warning range without any unexpected interference. In this example, a value of C_G close to 1 has been selected.

After definition of the confidence levels, it is possible to compute T_D, T_W, and α according to equations 3.4, 3.5, and 3.7 using the PDF identified in the previous step. If the identified PDF is that of a normal distribution, then equations 3.8, 3.9, and 3.10 could be used instead. In this example, the DCSC metric fits well a normal distribution, therefore the latter equations have been used. Visualization of the computed thresholds is depicted in Fig. 3.3 for two use cases: a dual-core MPSoC and a quad-core MPSoC. In both cases, the considered application is the benchmark described in Section 3.3.1.

3.3.6 Online Phase: Detection and Recovery

With the previous step, the offline phase is completed. The online phase configuration is available as a set of thresholds. What remains to define is the behavior of the online phase. This phase is completely implemented in the system software and is composed of two modules:

1. *Detection*: to monitor appropriate performance counters and signal when a violation of a detection rule occurs.
2. *Recovery*: respond to detection by actuating a proper recovery action.

If a task granularity is desired for the application at hand, the two modules just described should be integrated in the operating system and executed with the scheduler.

The detection module detects an interference and triggers the recovery module; detection occurs according to two rules:

1. *Alarm rule*: interference is detected through a measure of the metric above threshold T_D.
2. *Warning rule*: interference is detected through α consecutive measurements of the metric withing the warning region defined by $[T_W, T_D]$.

The behavior of the detection module is described in Algorithm 3.3.

The recovery module contains recovery actions. The designer can define any recovery action that is suitable for the application at hand. However, two types of recovery actions can be identified:

1. *Graceful degradation*: scheduling is temporarily or permanently altered to reduce interference. For instance, offending non-critical tasks can be delayed or entirely stopped to reduce interference on a victim critical task.

Algorithm 3.3 Detection module

```
Require: static variable alphacounter
 1: for all ConfiguredMetrics as metric do
 2:     if metric.is_composite then
 3:         values = {}

 4:         for all metric.counters as counter do
 5:             value = READ(metric.counter)
 6:             APPEND(value, values)
 7:         end for

 8:         m = COMPOSEMETRIC(metric, values)
 9:     else
10:         m = READ(metric.counter)
11:     end if

12:     if m $\geq T_D$ then
13:         RECOVERY(ALARM_RULE)
14:     else if m $\geq T_W$ then
15:         if alphacounter $\geq \alpha$ then
16:             RECOVERY(WARNING_RULE)
17:         else
18:             alphacounter++
19:         end if
20:     end if
21: end for
```

2. *Hard recovery*: the system cannot recover in time to grant functionality of safety-critical tasks, therefore a switch to an hot stand-by spare is performed to avoid failure.

The used recovery can be selected according to the triggering detection rule. If recovery is triggered by the warning rule, graceful degradation could be the most suitable action, whereas if recovery is triggered by the alarm rule, the hard recovery should be used. However, any final decision is for the designer to make, since all considerations are strongly application-dependent and no general rule can be formulated.

3.4 Hardware Support for ODIn

The ODIn architecture implements both spatial and temporal partitioning as described in Section 3.2 and 3.3. To be able to implement both partitioning, the ODIn architecture uses some supporting hardware. At least some of such hardware is often already implemented in the target MPSoC. This section contains a description of all the supporting hardware needed by the ODIn architecture; the designer should map the provided specification to available hardware on the system and add companion chips as needed. In most cases, at least one companion chip would be needed to implement the required support hardware. Such companion chip can be implemented either as an ASIC or, more likely, as an FPGA, and it would contain all the hardware that cannot be mapped on some portion of the target MPSoC.

3.4.1 Watchdog Processor

The watchdog processor (WDP) is a second line detection mechanism for temporal interference and also detects other types of CFE. Some faults that introduce temporal interference might not be detected by monitoring the selected metric. Although in some cases, this would indicate a wrong metric selection, in most cases it is due to the fact that coverage of the selected metric is not complete.

THE CFC implemented by the WDP is based on signatures defined at compile time. The first step in the implementation of such technique is to partition the monitored software module into blocks. In the ODIn architecture, the monitored software module nature depends on the granularity of the implementation. For instance, if the architecture is implemented with a task-level implementation, each task in the system should be monitored by a separate WDP. However, there is no impediment to implementing a different granularity in the CFC. Therefore, depending on the chosen granularity, the monitored software module is:

1. a task if the CFC is implemented at task granularity;
2. an application if the CFC is implemented at application granularity;
3. a partition if the CFC is implemented at partition granularity.

However selected, the software module should be partitioned into block. A block is here defined as a portion of software with one entry and one exit point. Differently to BBs, a block does not pose any constraint on internal jumps or even function calls. This definition allows for another degree of freedom to the designer, which can chose different sizes of blocks. The finest granularity in partitioning is using BBs as blocks for the CFC; the coarsest granularity is to consider the whole module as a block. In between such extremes, the software module can be partitioned in any number of blocks, as long as each block as exactly one entry point and one exit point. However, when partitioning the monitored module into blocks, the designer should consider that larger blocks mean longer error latency.

Once the software module has been partitioned into blocks, an arbitrary signature is assigned to each block. Signatures should be unique system-wide to avoid cross-errors, i.e., situations in which spatial partitioning fails and the WDP is re-armed by a different software module than the expected one. If signatures are unique, the WDP is able to detect this condition as an error. Signatures are constant at compile time, and are configured in the WDP at bootstrap; this design choice allows re-use of the same WDP design for different applications, therefore, the design should be certified only once.

During bootstrap, system software is in charge of configuring all WDPs in the system with the correct signature sequence. For each block, the WDP stores a signature and a WCET. Once configuration is complete, the WDP is enabled and starts expecting to receive the correct signature sequence within the expected timeouts. The WDP implements a cyclic behavior, i.e., it goes back to the first signature in the sequence after all signatures have been received.

The WDP is able to detect the following errors:

- *Wrong or illegal signature*: the WDP has received a signature which is not in the expected sequence (illegal) or it is in the expected sequence but is received at the wrong time (wrong signature). This condition can be an error of a CFE causing the software to skip a block or to execute the same block twice (wrong signature), or a data error in the interaction between the processor subsystem and the WDP.
- *Timeout*: the WDP does not receive the signature for the current block within the expected time.
- *Illegal interaction*: the software tries an unexpected interaction with the WDP, e.g. it tries to enable it again or to change the configuration. This can be a

consequence of a CFE causing the software to run the configuration phase again, or a data error in the interaction between the processor subsystem and the WDP.

When it detects an error, the WDP raises an interrupt request (IRQ) to the processor subsystem. The processor reacts by executing the proper interrupt service routine (ISR), which is part of the system software running in the system partition. The main limitation of the WDP, is that it relies on software to implement a recovery action, therefore it is unable to recover an error if the processor cannot execute software due to a SEFI or a permanent error.

The specifications of the WDP are quite peculiar, and it is rather difficult to find an MPSoC which already implements some hardware that satisfies all such requirements. Therefore, it is most likely that the WDP should be implemented on a companion chip.

3.4.2 Hardware Majority Voter

The hardware majority voter - or simply voter - implements the voter of a TMR scheme (see Section 1.5.1). It is an optional component of the ODIn architecture, only needed in some configuration (see Section 3.5). The voter is used when a given software module is triplicated and executed in parallel. Outputs of the three replicas are all written on the voter interface. As soon as all three outputs have been written, the voter performs a majority vote (Algorithm 1.1) and presents the selected output to the use interface.

To the best of our knowledge, no COTS MPSoC implements a voter, despite TMR being a quite common redundancy solution. However, several intellectual property (IP) providers have their version of a voter, which can be used when implementing ODIn on a new equipment.

3.4.3 System Watchdog Timer

Combining WDPs and Voter, CFEs and SDCs can be detected and managed by the ODIn architecture. However, some error conditions cannot be detected by none of the described detection mechanisms, including the spatial partitioning and temporal partitioning approaches described earlier in this chapter. Such conditions include all faults that make the system unable to run software or react to IRQs. To detect such errors, an additional hardware is required. Such hardware is here named the system watchdog timer (SWDT). The SWDT is a WDT that is configured and armed at bootstrap by the system software. One of the applications running in the system should be in charge of re-arming the SWDT within a given timeout. The most reasonable choice, is to give to the highest criticality application the job of re-arming the SWDT, and to use its WCET as a timeout.

When triggered - i.e., when it is not re-armed withing the configured timeout - the SWDT should be able to send an external signal to activate an hot stand-by spare. This is because the SWDT is only triggered when all other detection systems have failed and an error has caused the highest criticality application to overrun its WCET. Therefore, an unaffected hot stand-by spare has the output ready for the critical application and should be used as master from the detection instant on, until the next re-configuration is needed or until the next maintenance.

Most MPSoCs include some kind of WDT, however, not all WDT are able to send an external signal, therefore if no such WDT is available in the target MPSoC, the SWDT should be implemented on the companion chip.

3.4.4 Metric Monitor

A metric monitor is an hardware component which is able to measure an interference metric as define in Section 3.3.2. As discussed in Section 3.3, the best candidates to fulfill this role are the performance counters usually available in most MPSoC architecture. However, there exist the possibility that the application peculiarity requires monitoring of a metric which is not readily available in the available performance counters, or that in the particular target MPSoC there are no performance counters available through a software interface - they could be only accessible through a debug interface. In such instances, the designer should implement his/her own version of the performance counters in order to use the full ODIn architecture.

The main requirement of a performance counter, is that it should measure the selected metric in a completely transparent way, i.e., it should not introduce any interference in the system execution, besides configuration and periodic reading. Configuration is executed at bootstrap by the system software, whereas the reading operation is implemented either in system software - if the temporal partitioning is implemented in the system scheduler - or by the application software.

3.5 ODIn Configurations: Avionics and Space

The ODIn architecture has as main goal that of enabling the use of MPSoCs for MCS in industrial domains characterized by strict certification requirements and safety regulations. Of such industrial domains, the most peculiar ones are surely the avionic and space industry.

As described in Section 1.4, avionic OEMs have a particularly strict and complex certification process to face when proposing new equipment to be used on board of aircrafts, especially for civil avionics - military standards, though demanding, do not pose the same stress on safety that the civil standards do.

Although the space industry does not have the same strict safety requirements than the civil avionic, it still has some peculiar qualification processes for new equipment. Such processes are not as much standardized as the civil avionic process, and are specified on a case-by-case basis in a contract between the OEM and its client - which is often a public entity such as ESA or the national aviation and space agency (NASA). However, space applications share most of the concerns of an avionic application, although their requirements in terms of availability are usually softer than those for the avionic counterparts. Moreover, space applications have a unique concern with the space radiation environment. In this environment, electronic equipment is subject to particle strikes and as a consequence to SEEs. Therefore, OEM must take into account radiation effects in their design process, and contractors will often pose high dependability requirements - mostly on subsystems identified as mission critical.

In this section, two configurations of the ODIn architecture are described. One is more suitable for avionic applications, whereas the other is thought for the space industry domain.

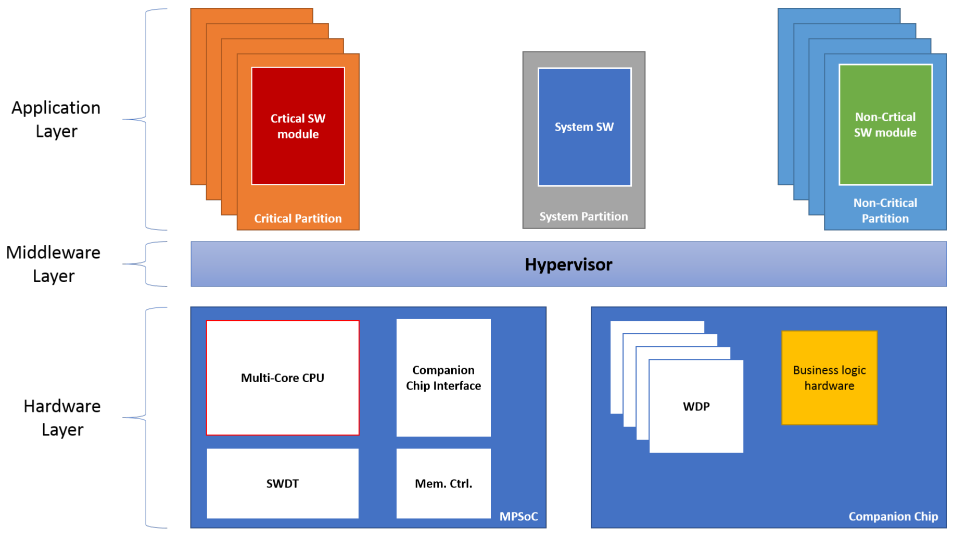


Fig. 3.4 Block diagram for the ODIn-A configuration.

3.5.1 Avionic Platform

ODIn for avionics (ODIn-A) can be described as a layered architecture containing most of the hardware and software modules described in this chapter. The layers of this architecture are the following.

Hardware layer: this is the lowest layer and contains both the target MPSoC and the companion chip, if needed. The WDP, the SWDT, the metric monitors, all belong to this level. Specifications described in Section 3.4, except for the voter, should all be mapped to this layer.

Middleware layer: this layer sits on top of the hardware and contains the system software. In this layer there are all the software means of partitioning, including:

- the type-1 hypervisor,
- the ISRs for all detection mechanisms described in this chapter,
- the temporal partitioning mechanism described in Section 3.3.

Application layer: this layer contains the application software. This is the business logic of the system and should satisfy all functional requirements.

As it is common in layered architectures, each layer provides services to the layer above. The hardware layer provides the interfaces for the middleware layer, which are used to configure the support hardware and to receive detection signals. The middleware layer provides mostly hardware-access functionalities to the application layer, including the spatial partitioning and the temporal partitioning.

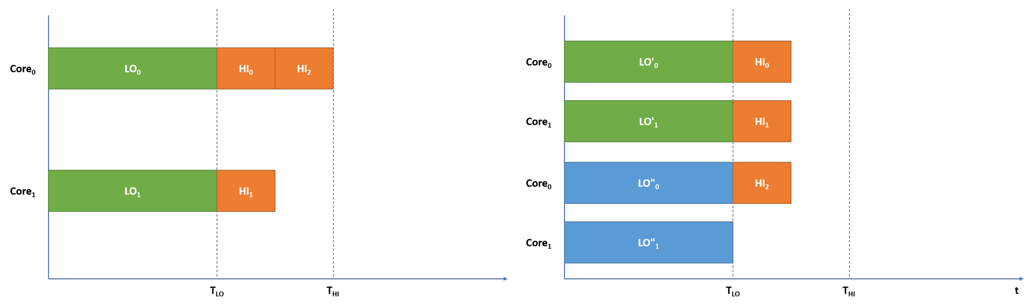
A block diagram of this configuration is provided in Fig. 3.4.

3.5.2 Space Platform

ODIn for Space (ODIn-S) is very similar to ODIn-A as described in Section 3.5.1. The main difference is that in this platform, the voter should be implemented in the hardware layer. Another difference is in the application layer.

In the avionic configuration, the application layer contains all the applications that the system needs to satisfy functional requirements, without any particular mandate on how such applications should be partitioned, which is left for the designer to decide. In the space platform, on the other hand, some special provisions should be take to account for the higher probability of soft errors due to SEEs.

Such provisions requires the classification of the applications to be integrated on the system in two criticality levels, HI and LO. Applications classified as HI are deployed with a TMR scheme. The voter is needed to support this configuration. In most cases, high criticality applications in the space domain are control applications, designed to keep the spacecraft in proper working order and to keep its designed position in orbit or its course in space. Applications classified as LO are configured to



(a) Scheduling on a dual-core MPSoC (b) Scheduling on a quad-core MPSoC

Fig. 3.5 Example of scheduling for space platforms on a dual-core and on a quad-core MPSoC. T_{LO} is the period of the LO applications, T_{HI} is the period of the HI application. On the dual-core, there are one LO application and one HI application. On the quad-core there are two LO applications and one HI application.

use a TTMR scheme. The TTMR scheme also requires a voter, however, the designer could chose to use a software voter for LO applications instead of an hardware one.

Another step in the space platform configuration is the definition of the scheduling of the system. Assuming that both LO and HI criticality applications are periodic and that their periods are harmonic, and assuming that the WCET estimation has already been performed, the parallelism provided by the MPSoC should be exploited to implement the TTMR and the TMR. Fig. 3.5 represents an example of how the scheduling of a space platform can be defined. The example refers to a dual-core MPSoC (Fig. 3.5a) and to a quad-core MPSoC (Fig. 3.5b). The scheduling should also take into account the necessity of executing the LO application for a third time in case an error affects one of the first two executions. Of course, the designer may opt for discarding the results of the affected execution and go on with normal computation, if the application can tolerate the loss of the results of the LO application. On the other hand, if this is not tolerable, the next period of the LO application should be devoted to the third execution and to the comparison, if the voter is implemented in software. Another interesting observation, is that on a dual-core platform, to implement the TMR for the HI application, the designer would need a processor that is twice faster as strictly needed to be able to perform the three execution as depicted in the scheduling. This requirement is not present in the quad-core version, where the three execution can happen all in parallel.

3.6 Chapter Summary

In this chapter the core proposal of this thesis has been presented. The issue that the thesis has been developed to solve, and which were introduced in Chapter 1, have been described in higher detail at the beginning of this chapter. Then the description of how the proposed ODIn architecture implements the two key requirements of space and temporal partitioning has been proposed.

Space partitioning is implemented by means of appropriate configuration of the MMU available on the target MPSoC. The MMU is managed by the type-1 hypervisor which is a key component of the ODIn architecture. The type-1 hypervisor is in charge of implementing the spatial-partitioning, as configured by the system designer; it is also in charge of providing support to application software in using the underlying hardware, and to support the temporal partitioning solution. Therefore, the type-1 hypervisor acts as a middleware in the ODIn architecture.

Temporal partitioning is a complex problem which has been the key focus of research in the MCS topic. In this thesis, the temporal partitioning is implemented through a bounded interference approach. An offline phase uses profiling data of the application at hand to compute such boundaries and then an online module uses performance counters or similar hardware to monitor a suitable metric to ensure that the boundaries of the interference computed in the offline phase are not overflown. The approach is a passive-reactive approach, i.e., it does not enforce the boundaries, but reacts to violations of such boundaries by implementing appropriate recovery actions.

The recovery actions proposed in the scope of the ODIn architecture are of two kinds:

Graceful degradation: this recovery action is used mainly when a fault is detected in a low criticality application or when a tolerable interference violation is detected, as better described in Section 3.3.6. It consists in the temporary or permanent stop of one or more low criticality applications in order to grant a larger resource budget to critical applications.

Hard recovery: this recovery action is used when a fault is detected in a high criticality application or when a non-tolerable interference violation is detected, as better described in Section 3.3.6. It consists in the activation of a hot-standby spare computer to grant that the critical applications provide their output on time.

The support hardware for implementing the ODIn architecture is described in Section 3.4 and consists of mainly four IP cores:

1. a WDP, which is in charge of detecting some spatial and temporal violations;
2. a voter, which is used especially in space applications to implement a TMR scheme for critical applications;
3. a SWDT, which is a watchdog timer that can trigger hard recovery. SWDT reacts to SEFIs or other conditions in which the processors are not able to run software anymore, therefore no other recovery action is possible.

4. a set of metric monitors, used to measure the metrics defined for the temporal partitioning approach. Such monitors can usually be mapped to performance counters readily available in the target MPSoC.

Finally, the two main configurations of the ODIn architecture are presented. ODIn-A is a configuration that mainly targets avionic applications, whereas the ODIn-S configuration targets space applications.

Chapter 4

Experimental Evaluation

This chapter describes the evaluation of the ODIn architecture, which has been performed to validate with empirical data the implementation of the partitioning strategies described in Chapter 3.

4.1 Target Hardware

To validate ODIn, the architecture has been implemented on two systems based on a MPSoC and featuring an FPGA, used to implement functionalities deployed on the companion chip. The selection of the boards has been made with two main criteria:

- the target system should be a complex COTS MPSoC available for real applications but not qualified nor certified for avionic or space applications;
- the two target systems should be rather similar MPSoCs, in order to cross-validate results on the two different platforms.

Two similar ARM systems were finally selected for validation purposes. Both systems are based on the Cortex-A9 processor architecture, which is a rather complex processor architecture featuring most of the key features of modern SoC. The two system selected were a dual-core Cortex-A9 and a quad-core Cortex-A9. The dual-core system is the Xilinx Zynq all-programmable SoC (APSoC), described in Section 4.1.1; the quad-core systems is the NXP i.MX6Q, described in Section 4.1.2.

Both systems were part of a development board. The Zynq was deployed on a ZedBoard development board; the i.MX6Q was deployed on an Inventami development board, featuring a companion FPGA.

4.1.1 ZedBoard

The ZedBoard is a development board commonly used to prototype systems based on the Zynq APSoC. Detailed features of the board are not of much interest for the ODIn evaluation described in this chapter; the interested reader may refer to [74]. Suffice to say that the ZedBoard, besides providing a working system based on the Zynq system,t does also provide the necessary infrastructure for connecting the Zynq to a debugging system; moreover, the ZedBoard features all the necessary support hardware to load a FPGA configuration and a program on the system.

The ZedBoard features a Xilinx Zynq APSoC [75][76]. The Zynq is a system that features on the same chip a MPSoC based on a Cortex-A9 dual-core processor and a FPGA fabric.

The Cortex-A9 subsystem is labeled processing system (PS) in the Zynq jargon, as opposed to the programmable logic (PL) which indicates the FPGA fabric.

The PS is a self-contained MPSoC, which features an application processing unit (APU) - i.e., a Cortex-A9 dual-core SoC - and several peripheral that are commonly used in system development. Such peripherals include:

- several I/O interfaces - universal asynchronous receiver transmitter (UART), serial peripheral interface (SPI), inter-integrated circuit (I^2C, universal serial bus (USB), secure digital (SD) interface, general purpose I/O (GPIO), gigabit Ethernet (GigE), controller area network (CAN) - all connected to a multiplexed I/O (MIO) and an extended multiplexed I/O (EMIO) controller for selecting appropriate output pin configuration;
- a CoreSight™ debugging component
- a central interconnect between the APU and the I/O peripherals described above. The central interconnect also has interfaces towards low-speed memory interfaces - NOR/NAND flash, Q-SPI - and towards the PL through a set

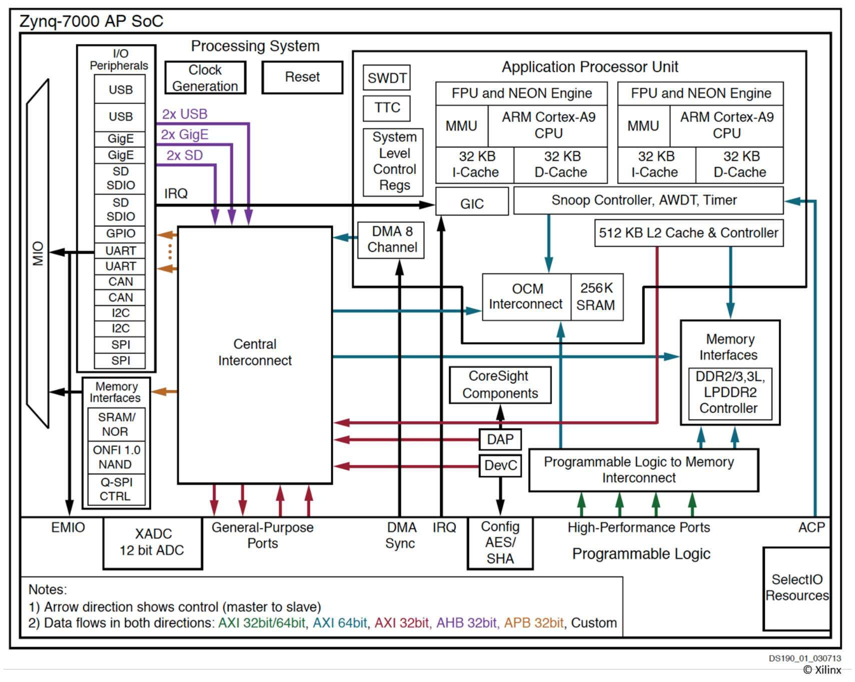


Fig. 4.1 Block diagram of the Xilinx Zynq-7000 APSoC.

of two general purpose advanced extensible interface (GP AXI) ports, each including a master and a slave interface;

- a set of four high-performance ports for direct access to the central memory and to the on-chip memory (OCM) from the PL. These ports are used for high-speed communication between PS and PL, although the data coherency is not guaranteed in this channel;
- an advanced coherency port (ACP), implementing high-speed communication with data coherency between APU and PL;
- a direct memory access (DMA) interface, allowing the PL to use the DMA controller available in the APU.

All such devices are configurable by software through a dedicated set of configuration registers. Details of such interfaces are out of the scope of this thesis; the interested reader may refer to [76].

The PL on the Zynq is connected to the PS through the AXI interfaces listed above. Moreover, the PL can also access a set of pins to communicate outside the

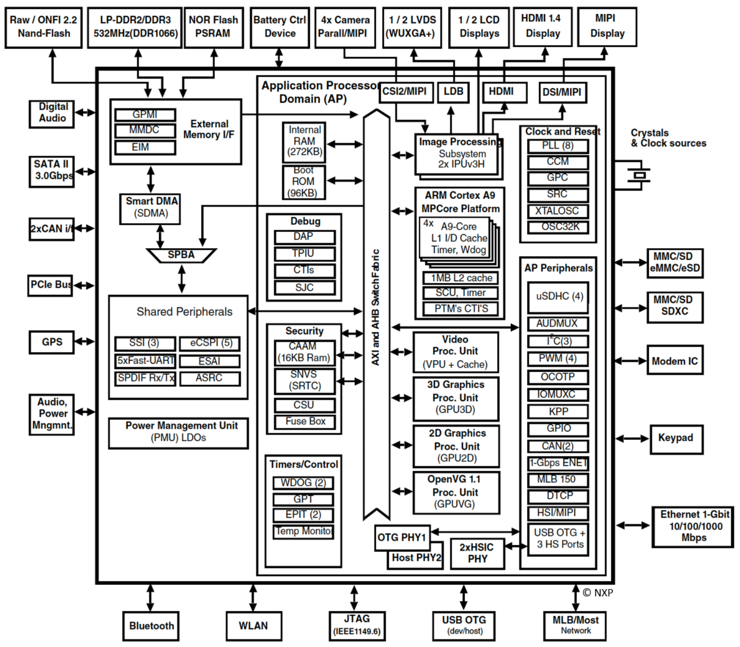


Fig. 4.2 Simplified block diagram of the i.MX6Quad MPSoC

Zynq subsystem. For instance, the PL could receive and process sensor data directly, without any involvement of the PS. This approach, commonly named *hardware acceleration* is quite common in system design, FPGAs have been used for similar purposes on all sorts of systems, including space and avionic applications. The main advantage of the Zynq, in this context, is that the PL is tightly linked to the PS, therefore it suffers less communication latency than an external FPGA.

The Zynq represents a good choice for ODIn evaluation, since it satisfy the few hardware requirements of ODIn: it is a MPSoC and it features MMUs. The integration of the FPGA with the MPSoC, makes it also a convenient choice for fast prototyping of the ODIn architecture and early evaluation of the solution.

4.1.2 Inventami Board

The Inventami Board is a rather new development board proposed by SanitasEG [77], to which the author of this thesis had an early access. It is a development board based on a NXP i.MX6Q MPSoC and on an FPGA, connected to the MPSoC through a peripheral component interconnect express (PCI-e) interface.

The i.MX6Q MPSoC is a complex MPSoC based on a quad-core Cortex-A9 [78]. It features advanced peripherals including:

- a 2D and a 3D graphics processor unit (GPU) to be used in multimedia applications;
- a high-definition multimedia interface (HDMI) controller;
- a PCI-e controller
- a serial advanced technology attachment (SATA) controller;
- a smart DMA (SDMA) controller.

Moreover, the i.mx6Q MPSoC also includes most of the peripherals already listed for the Zynq; the main difference between Zynq and i.MX6Q, for the scope of this thesis, is of course the lack of an FPGA. The solution for this issue is the placement on the same board of an FPGA, as on the Inventami board.

The FPGA on the Inventami board is a Lattice®ECP5UM85 with a serialized/deserializer (SERDES) component.

4.2 Benchmark Applications

The evaluation of the ODIn architecture has been performed on both the target hardware described above for an avionic use case and for a space use case. The benchmarks used on both targets are described in this section. Both benchmarks have been implemented in two versions:

1. a dual-core version for the Zynq;
2. a quad-core version for the i.MX6Quad.

The avionic benchmark was developed in the framework of a collaboration with Leonardo S.p.A. It is based on requirements that Leonardo has indicated for an on-board equipment to be used on civil helicopters designed for rescue duties.

The space benchmark has been developed in the scope of this thesis and is based on actual software modules that can be commonly found in payload computers, especially in scientific missions. By payload computer, it is here intended any computer that is on board of a spacecraft and is not involved in maintaining the spacecraft on its correct course and with the expected asset. An example of payload computer can be found in [62] and [79].

Both benchmarks share some key component algorithms that are briefly described in the following subsection, before describing the benchmarks and their implementation on both the dual-core and the quad-core platforms.

4.2.1 The *RICE* compression algorithm

Both benchmarks use a compression algorithm which is based on the RICE encoding. The RICE encoding belongs to the family of the Golomb codes [80]. These are arithmetic encodings based on the division operation. In general, a Golomb code encodes a number by computing its quotient and remainder by an integer k. The quotient is encoded in unary, whereas the remainder is encoded in binary; the two parts are separated by a spacer bit.

In RICE encoding, k is assumed to be a power of 2, which simplifies the implementation since the division by a power of 2 can be simply implemented through a shift operation.

The RICE compression algorithm used in the benchmarks described in this section, is an entropy encoder which uses the RICE coding to encode sample differences. It is a loss-less compression algorithm, since the original signal can be completely reconstructed. To improve compression ratio, the algorithm includes some adaptiveness.

The original data O to compress is composed of samples p of size ψ. O is subdivided into blocks of fixed length B_L. Differences d_i in a given block l are computed between each sample O_{i+lB_L-1} and its predecessor O_{i-1+lB_L-1} - the predecessor of the first sample in a block is the latest sample of the previous block O_{lB_L-1}. Differences are then mapped on the positive integer m_i by the simple law

$$m_i = \begin{cases} 2d_i \text{ if } d_i \geq 0 \\ 2\left|d_i\right| - 1 \text{ if } d_i < 0 \end{cases} \tag{4.1}$$

After these operations, each sample m_i is encoded according to the RICE encoding.

Algorithm 4.1 RICE compression algorithm

```
Require: $O$, $B_L$, $\psi$
Ensure: $C$
 1: for all Blocks of length $B_L$ in $O$ do
 2:     Compute sample-to-sample entropy

 3:     if entropy is null then
 4:         Count the block as a block of zeros
 5:     else
 6:         Encode blocks of zeros and reset counter of blocks of zeros.

 7:         Map entropy to positive integers according to (4.1)
 8:         Find $k_{opt}$

 9:         for all samples $m_p$ in the current block do
10:             if $k_{opt} < log_2(\psi)$ then
11:                 RICE encode sample $m_p$
12:             else
13:                 Encode sample $m_p$ as is
14:             end if
15:         end for
16:     end if
17: end for
```

For each block of samples, the optimal value of the divider 2^k is identified by analyzing m_i samples and performing a *dry-run* to compute the size of the resulting compressed block for each value of k such that $2^k \leq \psi$ where ψ is the size of a sample in bits. If the optimal k is actually such that $2^k = S$, then no compression is performed and the block is transmitted as is. If the current block entropy is null - i.e., all samples in the block are equal to each other and to the last sample of the previous block - the current block is not encoded, instead it is counted in a special counter of blocks of zeros. As soon as a block with non-null entropy is found, all blocks of zeros counted up until now are encoded as in (4.2). All regular blocks are instead encoded as in (4.3).

$$\underbrace{\overbrace{000}^{log_2(\psi)\text{ bits}}}_{\text{header}} \quad \underbrace{\overbrace{10\ldots01}^{\psi\text{ bits}}}_{\text{zeros counter}} \tag{4.2}$$

$$\overbrace{\underbrace{0\ldots1}_{k}}^{log_2(\psi)\text{ bits}} \quad \overbrace{\underbrace{1\ldots1}_{\text{quotient}}}^{\text{variable length}} \quad \overbrace{\underbrace{0}_{\text{separator}}}^{\text{1 bit}}\overbrace{\underbrace{10\ldots01}_{\text{remainder}}}^{k\text{ bits}} \tag{4.3}$$

At the end of the algorithm, the compressed data C is ready to be transmitted. The RICE compression algorithm is described in Algorithm 4.1.

4.2.2 The *Sobel* edge detection algorithm

Modern aircrafts, especially UAVs, often includes some vision algorithms, as do most space applications with a scientific mission. A common component of such vision algorithms is an edge-detection module. This section describes one of the most widely used two-dimensional edge detection algorithms, which is also used as a component of the benchmarks used in this thesis. The filter that is at the core of this algorithm was first described in [81], although it was first proposed in [82]. The theory behind the definition of this filter is out of the scope of this thesis. Suffice it to say that the algorithm is based on computation of the image gradients in both directions; the points where the gradient is higher are likely edges of the objects in the image.

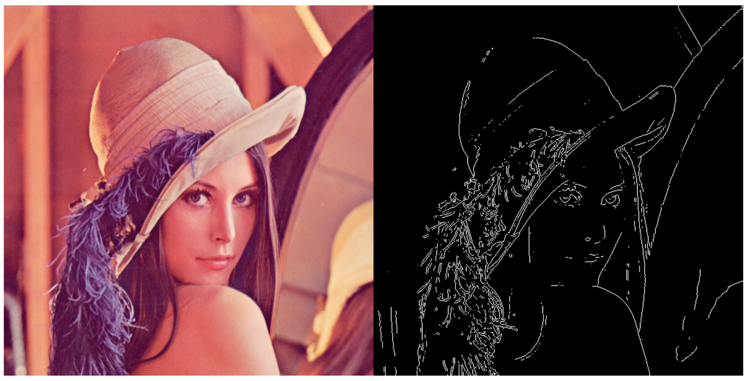


Fig. 4.3 Application of the Sobel edge detection on a standard image. On the left the original image, on the right results of the Sobel algorithm

Sobel algorithm results can be displayed as a gray-scale image in which the edges are visible as white lines. An example of application of the Sobel algorithm on a standard image is provided in Fig. 4.3.

4.2.3 The Avionic Benchmark Application

The basic components of the avionic benchmark - that are shared by both the dual-core and the quad-core version - are a control law module and a sensor log module.

The *control law module* implements a part of an asset control system for an unmanned aerial vehicle (UAV). It reads sensors inputs, perform some computation and produces some commands for the actuators. It receives 12 bytes of data, representing the three asset angles: yaw, pitch, and roll. It processes the sensors data to produce two 4 bytes command words for two actuators.

The *vision module*, which receives a frame from an on-board camera to compress it using the RICE algorithm (see Section 4.2.1) to allow real-time transmission to a ground station.

The quad-core benchmark also includes two additional modules:

1. the *sensor log module* receives the same inputs as the control low data, and stores it in a special memory area to be dumped on the ground by maintenance personnel;
2. the *identification module* is a pre-processing module for a target identification system. This module performs an edge detection using the Sobel filter (see Section 4.2.2), whose results should be then used by other modules within the target identification and tracking subsystem.

The control law module is a mission-critical application. It could also be considered safety-critical if the UAV application could cause passer-by injuries. The control law module is directly involved in the airworthiness of the vehicle; all other modules described above are classified at a lower criticality than the control law module, with the sensor log module classified as a best effort - i.e., non-critical - module.

The avionic use case requires the configuration of one partition for each of the described modules, besides the system partition required by the ODIn architecture.

Moreover, the control law module must be analyzed in order to apply the detection interference method described in Section 3.3. In the scope of this thesis, the control law module has been considered a victim of non-critical tasks affected by bugs. The monitored metric is the DCSC as in the example in Section 3.3.

4.2.4 The Space Benchmark Application

The space benchmark is based on common requirements for payload computers. Two component applications are shared by both the dual-core and the quad-core implementations:

1. the *control module* implements a simple control application which reads sensors data, performs some computations, and outputs actuators commands. This is a mission-critical application, since such kind of control application might be in control scientific instrumentation crucial for the payload computer mission and which cannot be misdirected.
2. the *data processing module* implements a data processing algorithm composed of an edge detection and of a compression. The edge detection is implemented through the Sobel algorithm (see Section 4.2.2), whereas the compression is performed using the RICE algorithm (see Section 4.2.1).

In the quad-core implementation, a third module is added to take advantage of the additional computation capabilities. The additional module is a sensor logger, similar to the one used in the avionic benchmark (Section 4.2.3), which is a non-critical module.

The space benchmark was deployed on an ODIn-S platform. In the dual-core scenario, the data processing module D was considered a non-critical application and deployed in a TTMR scheme, whereas the control module was considered a critical application and deployed in a TMR scheme. As in the avionic benchmark, the control module was analyzed for the application of the online detection methodology described in Section 3.3.

4.3 Experimental Setup

The experimental evaluation of the ODIn platform in the two described use cases was performed by means of fault injection campaigns. The fault injection is a common methodology for evaluating response and performances of fault injection/fault recovery strategies. In the scope of this thesis, the fault injection was used with a double purpose:

1. prove the dependability of the ODIn architecture in an avionic use case, where the main threat to dependability is represented by low-criticality applications interfering with high-criticality applications sharing the same hardware;
2. prove the dependability of the ODIn architecture in a space use case, where the main threat to dependability is represented by radiation effects and where the same issue of interference by low-criticality applications against high-criticality applications also applies.

The experiments where performed through a custom fault injection system developed in the scope of this thesis, which was designed to inject the following kind of faults:

1. bit-flips in configuration registers of the target MPSoC and in architectural registers of the CPU, to emulate SEEs.
2. bit-flips in the code memory area to emulate bugs in the non-critical applications [83].
3. excitation of an artificial bug in non-critical applications to induce temporal interference.

In the rest of this section, the fault injection system is described with references to relevant literature and the execution of a fault injection campaign is detailed for better evaluation of the results presented in following sections, then the type-1 hypervisor used in the experiments is described.

4.3.1 The Fault Injection System

The fault injection system implemented in the scope of this thesis is a custom fault injection which is composed of two actors:

1. a host computer;
2. a hardware debugger interface.

As described above, a fault injection system is composed of

- a *fault list generator*;
- an *injection manager*;
- a *fault injector*;
- a *workload generator*;
- a *system monitor*;
- a *data collector and analyzer*.

Fault List Generator

This component is in charge of generating a fault list to be used during fault injection campaigns. The fault injection system implemented in the scope of this thesis is able to inject different kind of faults, however, each campaign should be homogeneous in the type of fault injected - i.e., a campaign should only inject SEEs and let the software fault injection to a different campaign. The fault injector is able to generate all kinds of fault and provides different lists, one for each kind of fault.

In the case of the SEEs injection and of the software fault injection, the whole space of faults is quite big, therefore a sampling is needed to obtain a feasible campaign. In both cases the fault space is a two dimensional space in which one dimension is the time - with t_0 being the beginning of the execution of the workload and with a granularity equal to the clock cycle tick - and the other dimension is the composed of the bits that can be flipped by any given fault. In the case of SEE injection, such bits are all the bits in the configuration registers of the MPSoC and

Algorithm 4.2 Fault List Generator pseudo-code.

```
Require: Targets, MaxTime, Cardinality
Ensure: FaultList
1: $TargetsN \leftarrow length(Targets)$
2: for $i = 0 \rightarrow Cardinality$ do
3:     $Target \leftarrow Targets[\textsc{RandomInt}(0, TargetsN)]$
4:     $MaxBit \leftarrow sizeof(Target)$
5:     $Bit \leftarrow \textsc{RandomInt}(0, MaxBit)$
6:     $Time \leftarrow \textsc{RandomInt}(0, MaxTime)$
7:     $FaultList[i] = concatenate(Register, Bit, Time)$
8: end for
```

all the bits in CPU registers; in the case of software fault injection, such bits are all the bits composing the code memory area.

A first step to reduce the size of the fault list is to only inject faults in relevant phases of the workload. The workload always includes the bootstrap phase of the system, which is assumed to be immune to SEEs due to its limited run-time with respect to the main application. The time during which the bootstrap is executed can be ignored in the fault list generation. A second step is to exclude all those bits that are known to have no effect on the application - such as configuration bits of unused peripherals in the system - or whose effect is known a priori - e.g., modifying the configuration of the memory controller brings the system to a halt since it becomes unable to access the central memory.

To further reduce the fault list size, a random sampling is introduced. The final size of the fault list is selected through a set of fault injection campaigns with increasing fault lists cardinality. The final cardinality is selected when the collected statistics reach a statistical convergence, in a way similar to the Montecarlo approach. The algorithm implemented by the fault list generator is provided in Algorithm 4.2.

Injection Manager

This component coordinates the components directly communicating with the target system and selects the next fault to inject. The injection manager of the implemented fault injection system reads a fault from the fault list, passes it to the fault injector and starts the workload generator. It also awaits results from the data collector and analyzer before going on with the next fault injection.

This component generates a sets of commands for the hardware debug system, which implements the fault injection system's components that are more strictly coupled with the target system. The injection manager algorithm is described in Algorithm 4.3.

Algorithm 4.3 Injection Manager pseudo-code.

```
Require: FaultList, Cardinality
Ensure: Fault Campaing Results
1: for $i = 0 \rightarrow Cardinality$ do
2:     $Fault \leftarrow FaultList[i]$
3:     Reset Target
4:     Load benchmark on target and start its execution
5:     Wait for bootstrap completion
6:     Instruct fault injector with $Fault$ data
7:     Wait for data collector and analyzer results
8: end for
```

Workload Generator

This component creates a suitable activation path for the system, to make fault effects observable by the fault injection system. This component loads the software and FPGA configuration on the target system and starts the execution of the applications.

Data Collector and Analyzer

This component collects and analyzes results from the target and classifies fault effects as described in Section 4.3.2.

System Monitor

This component monitors the target system during execution.

Fault Injector

This component injects a fault for each fault injection experiment. This component is implemented on the hardware debug system, which receives the fault information

Algorithm 4.4 Data collector and analyzer pseudocode

```
Require: Application Results location and length, WD error register address
Ensure: Fault classification
 1: if if an exception has been triggered then
 2:     Classify as Exception
 3:     return
 4: end if
 5: if WD error register is not null then
 6:     Classify as CFE detected
 7:     return
 8: end if
 9: Dump memory region containing results
10: if the dump is different from the golden results then
11:     Classify as Failure
12: end if
```

from the injection manager. The hardware debug system uses the debug interface on the target system to stop the execution, inject the fault, and resume execution.

4.3.2 Description of a Fault Injection Campaign

A fault injection campaign is a collection of fault injection experiments. Each fault injection experiments injects a single fault from the fault list in the target system and observe the outcomes to classify the fault.

The first step in a fault injection campaign is the generation of the fault list and the production of a golden result. The golden result is obtained by running the workload once without injecting any fault. After this initialization phase, the proper campaign starts by executing one experiment for each line in the fault list.

A fault injection experiment's first step is to reset the target system to ensure a clear state and avoid fault accumulation. After the system reset, the system configuration - composed of the software and of the FPGA configuration - is loaded on the system and the system is released. At this point the system executes the bootstrap phase, at the end of which the injection manager triggers the fault injector. The fault injector stops the target system and injects the fault by modifying the target bit or by activating the excitation condition of the artificial bug. After the injection, the fault injector releases the system which resumes its workload. The execution may end as expected or it may be stopped due to an exception or timeout being triggered in the system. Another stop condition is the activation of the hard recovery rule described in Section 3.3.6. At the end of the execution, however reached, the data collector and analyzer is triggered. This component reads the state of the system and the output data and classifies the outcome of the experiment according to the following classifications. A different classification has been used for hardware fault injection campaigns and for software fault injection campaigns. In hardware fault injection campaigns, faults were classified as:

Silent or No Effect (NE) faults that did not produce any observable misbehavior after their injection. Faults affecting a register just before it is overwritten with a correct value, and faults affecting unused parts of the system are examples of this class of faults.

Control Flow Error (CFE) faults detected by either a hardware exception (fetch at illegal address, fetch outside the code memory area resulting in an illegal opcode exception), or by the WDPs (illegal signature, timeout).

System timeout (TO) faults detected by the SWDT.

Silent data corruption of failures (F) faults that cause a wrong output without any detection mechanism issuing a warning to the user.

Faults in the software fault injection campaigns were classified as:

Silent or No Effect (NE) did not produce any observable misbehavior.

Non-critical error (NCE) faults causing an error in a non-critical application.

Critical error (CE) faults causing an error in a critical application

Illegal opcode (IC) faults affecting the opcode of an instruction making it an illegal opcode - i.e., an opcode not defined in the instruction set architecture (ISA). Such faults are detected by hardware exception.

Silent data corruption of Failure (F) undetected faults that causes a wrong output from the critical application.

The artificial bug injection campaigns were evaluated in terms of activation of one of the two detection rules described in Section 3.3.

4.3.3 The Type-1 Hypervisor

Although the approach described in Chapter 3 is independent on the actual hypervisor selected in the specific application, some requirements are implicitly imposed on the selection of such component. A first requirement is that the selected type-1 hypervisor supports a scheduling which is compatible with the recommendations and guidelines currently used in the industrial domain of reference. A second requirement is that the selected type-1 hypervisor implements resource partitioning in a transparent way from the application software point-of-view. For instance, resource partitioning might be implemented by the type-1 hypervisor by means of MMU configuration. The space use case, described below, adds the further requirement of inter-partition communication (IPC) channels in order to implement the TTMR approaches. In this case, IPC channels are used to communicate results of the first two execution to the partition implementing the voter.

In the scope of the experimental evaluation described in this chapter, Sysgo's PikeOS has been used a type-1 hypervisor. PikeOS was born as a RTOS, and grew to become a type-1 hypervisor focused on embedded systems. It is a commercial product, available for use in actual applications, which was available to the author through the EMC^2 project.

In the following sections, the actual implementation of the evaluation systems is described in more detail. Whenever refferring to a type-1 hypervisor in such descriptions, the reader should assume that PikeOS was used.

4.4 Evaluating ODIn for an Avionic Application

As described in Section 3.5.1, the ODIn architecture has been used to implement an avionic application on two target platforms, described in Section 4.1. In this section, the two target platforms are described in their final configuration, then the performed experiments are described and their results are discussed.

4.4.1 Dual-Core Platform

The dual-core platform used for the experiments is described in detail in Section 4.1.1. The avionic benchmark deployed on this system is composed of two applications:

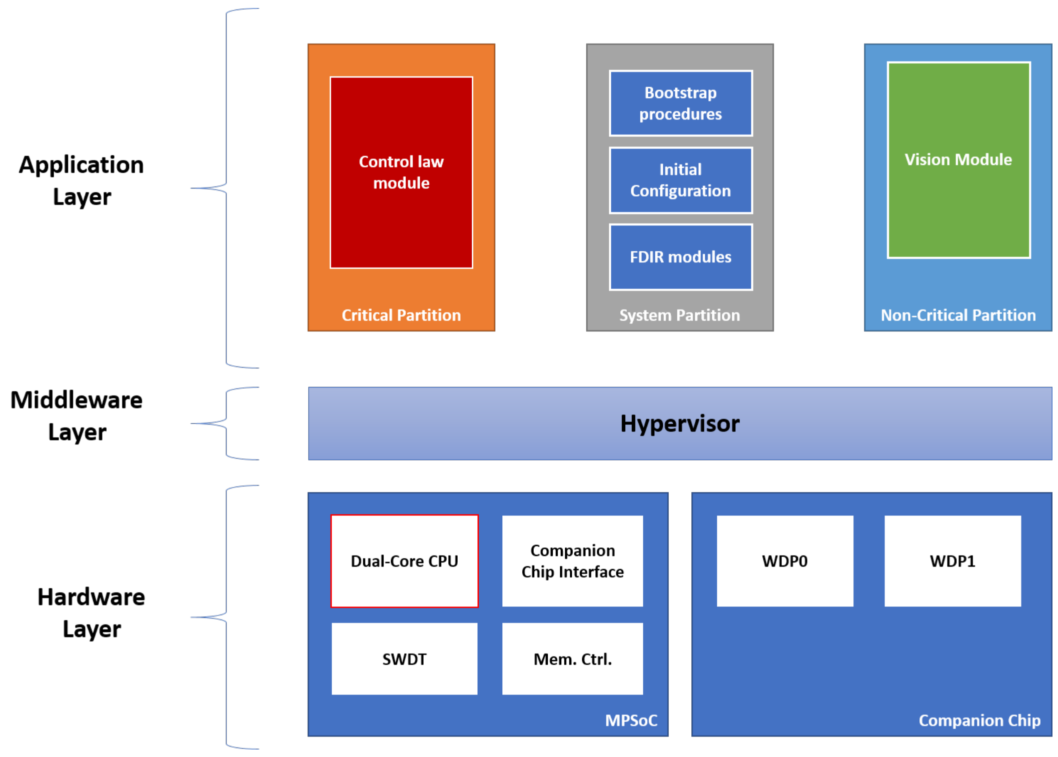


Fig. 4.4 System-level block diagram for avionic dual core implementation.

1. the critical *control law* module,
2. the non-critical *vision* module.

The vision module, in this implementation, performs a RICE compression of a fixed frame. The frame is read from a memory location, which represents the destination buffer of a camera. The purpose of this application is to reduce the communication bandwidth necessary for real-time communication with a ground station in an UAV application. The control law module, in this implementation, controls the three fundamental angles in the longitudinal flight phase. Such angles are yaw, pitch, and roll. Together, they control the asset of the plane in the tridimensional space. The control law implemented is intended to keep the angles against noise introduced by air turbulence and to respond to inputs from the pilot. Both sensors and pilot inputs are received by the application through a proprietary serial protocol. Commands are intended to direct the mechanical actuators and are sent to the actuator drivers through the same proprietary serial protocol.

The ODIn configuration on this platform is as follows:

Software layer. In this implementation, the ODIn architecture requires three partitions:

1. critical partition to contain the critical application
2. non-critical partition to contain the non-critical application
3. system partition to contain the bootstrap procedures, the initial system configuration, and the system recovery actions.

The configuration of the software layer is performed by the system integrator at the end of the development of the application software and hardware components.

Middleware layer. In this implementation, the middleware layer - i.e., the type-1 hypervisor - is configured to enforce containment of the three partitions described in the software layer. Such configuration includes the static mapping of each application to a subset of the available processing cores. More precisely:

- the critical partition is statically mapped to core 0,
- the non-critical partition is statically mapped to core 1,
- the system partition is allowed to execute on the first available core.

The middleware layer is also configured to implement the scheduling scheme needed by the application software. The scheduling scheme is the same for both critical and non-critical software. Both partitions are scheduled concurrently on the same core with a period of 20 ms. Any available slack is used to run the system partition. After system initialization, the system partition only runs in an asynchronous way to respond to IRQs and exceptions.

Hardware layer. In this implementation, the hardware layer is composed of:

- two WDPs and one SWDT. Both are IPs described in Section 3.4
- relevant performance counters.

In particular, the performance counters have been configured to observe the DCSC metric described in Section 3.3 for the control-law module. The performance counters in this implementation are up-counters that are able to issue an IRQ when reaching their maximum value. They are configured so that the maximum value is reached after the T_D events are counter - in this implementation, after the core running the critical application is stalled for more than T_D clock cycles waiting for data to be available in the local cache. The performance counters are also read by a module in the critical application (which should be integrated in the type-1 hypervisor in a real-life application) just before the application yields to the system scheduler. If the value read by this module is above T_W, a software counter is incremented. When such software counter reaches the α threshold, a graceful degradation recovery action is implemented by the system partition, which is in turn invoked by the critical partition through a software trap.

4.4.2 Quad-Core Platform

The quad-core platform used for the experiments is described in detail in Section 4.1.2. The avionic benchmark deployed on this system is composed of four applications:

1. the critical *control law* module,
2. non-critical *vision* module,
3. non-critical *identification* module,
4. non-critical *sensor log* module.

The vision module, in this implementation, performs a RICE compression of a fixed frame. The frame is read from a memory location, which represents the destination buffer of a camera. The purpose of this application is to reduce the communication bandwidth necessary for real-time communication with a ground station in an UAV application.

The identification module, in this implementation, applies a Sobel filter to the same frame used by the vision module, to identify and track a target.

The control law module, in this implementation, controls the three fundamental angles in the longitudinal flight phase. Such angles are yaw, pitch, and roll. Together, they control the asset of the plane in the tridimensional space. The control law implemented is intended to keep the angles against noise introduced by air turbulence and to respond to inputs from the pilot. Both sensors and pilot inputs are read from a buffer region, which is written by a serial communication controller implementing a proprietary protocol. Commands are intended to direct the mechanical actuators and are sent to the actuator drivers through the same proprietary serial protocol.

The sensor log module reads the same inputs as the control law module and stores them in a dedicated permanent memory, to be used by ground personnel.

The ODIn configuration on this platform is as follows:

Software layer. In this implementation, the ODIn architecture requires five partitions:

1. critical partition to contain the critical application,
2. non-critical partition to contain the vision module application,
3. non-critical partition to contain the identification module application,
4. non-critical partition to contain the sensor logging application,
5. system partition to contain the bootstrap procedures, the initial system configuration, and the system recovery actions.

The configuration of the software layer is performed by the system integrator at the end of the development of the application software and hardware components.

Middleware layer. In this implementation, the middleware layer - i.e., the type-1 hypervisor - is configured to enforce containment of the three partitions described in the software layer. Such configuration includes the static mapping of each application to a subset of the available processing cores. More precisely:

- the critical partition is statically mapped to core 0,
- each non-critical partition is statically mapped to one of the remaining cores,
- the system partition is allowed to execute on the first available core, except core 0. This reduces the expected interference allowing for a less pessimistic WCET estimation for the critical application.

The middleware layer is also configured to implement the scheduling scheme needed by the application software. The scheduling scheme is the same for both critical and non-critical software. All partitions are scheduled concurrently on the same core with a period of 20 ms. Any available slack on cores $1,2,3$ is used to run the system partition. After system initialization, the system partition only runs in an asynchronous way to respond to IRQs and exceptions.

Hardware layer. In this implementation, the hardware layer is composed of:

- four WDPs and one SWDT. Both are IPs described in Section 3.4. Each WDP is in charge of controlling a single partition. Each WDP issues a specific IRQ, allowing for a limited diagnosis. Such diagnosis allows execution of a specific recovery action depending on which WDP has been triggered.
- relevant performance counters.

Performance counters have been configured to observe the DCSC metric described in Section 3.3 for the control-law module. The performance counters in this implementation are up-counters that are able to issue an IRQ when reaching their maximum value. They are configured so that the maximum value is reached after the T_D events are counter - in this implementation, after the core running the critical application is stalled for more than T_D clock cycles waiting for data to be available in the local cache. The performance counters are also read by a module in the critical application (which should be integrated in the type-1 hypervisor in a real-life application) just before the application yields to the system scheduler. If the value read by this module is above T_W, a software counter is incremented. When such software counter reaches the α threshold, a graceful degradation recovery action is implemented by the system partition, which is in turn invoked by the critical partition through a software trap.

4.4.3 Experiments and Results

Hardware and Software fault injection campaigns

Similar sets of experiments were performed on both the dual-core and the quad-core implementation.

Table 4.1 Hardware fault injection results on the dual-core platform for the avionic use-case. For acronyms see Section 4.3.2

Target	NE	CFE	TO	F	Inj.
CPU RF	87.75 %	5.70 %	6.55 %	0	2000
Config.	96.65 %	0	3.35 %	0	4000
Total	93.68 %	1.73 %	4.33 %	0	6000

Table 4.2 Software fault injection results on the dual-core platform for the avionic use-case. For acronyms see Section 4.3.2

NE	CE	NCE	IO	F	Inj.
88.80 %	0	11.20 %	<0.01 %	0	10000

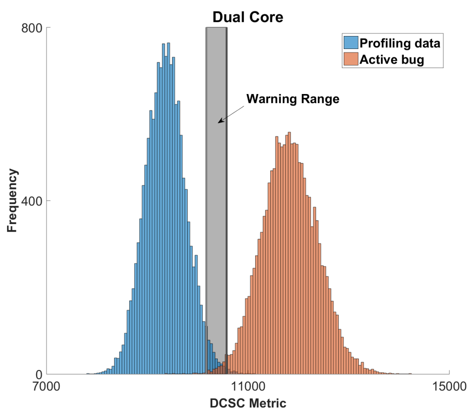
The first set of experiments was a fault injection campaign injecting hardware faults on all cores and on MPSoC configuration registers. A second set was a different fault injection campaign injecting software faults on all applications.

Results of such campaigns are very similar for both quad-core and dual-core implementations. Table 4.1 and Table 4.2 show results for hardware and software fault injection campaigns, respectively, on the dual-core platform.

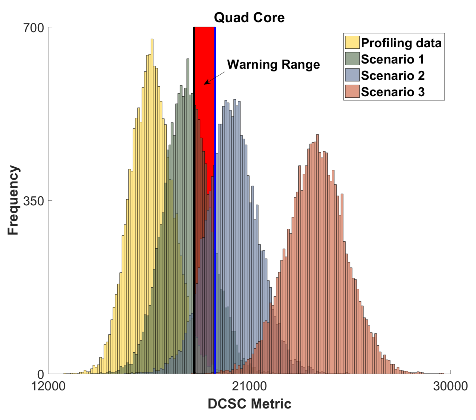
Such results show that the implemented partitioning mechanisms are able to grant a total coverage on the type of faults injected. The number of injected faults has been selected as described in Section 4.3.1 by observing that statistical convergence of the fault classification. Results show that the implementation of the ODIn architecture makes the application immune to all injected faults, since there are no failures.

Artificial bug injection campaigns

The evaluation of the temporal partitioning solution described in Section 3.3, required the injection of an artificial bug in non-critical applications in order to introduce unexpected interference for the critical application. The shared resource considered in these experiments is the share memory hierarchy, starting from the level-2 cache (L2C) up to the central memory. The artificial bug was designed so that the affected non-critical application would perform continuous accesses to the memory subsystem, without respecting its scheduling, and causing a line eviction on the private



(a) Dual-core platform



(b) Quad-core platform

Fig. 4.5 Results of the application of the temporal partitioning on both the dual-core and the quad-core platforms

level-1 cache (L1C) at each access. The line eviction generates additional traffic on the shared bus, since it requires the evicted line to be written to memory - according to the selected cache coherency protocol - and the new line to be read from memory. The method described in Section 3.3 was applied to both platforms, according to the description provided in said section. The bug was injected a total of 15 000 times at different time instants during the execution of the critical application. Results are represented in Fig. 4.5. The figure shows how the distribution of the observed DCSC metric shifts to the right when there are active bugs in the system.

Table 4.3 reports a summary of the detections and which rule detected the fault in each scenario on both platforms. To each rule corresponds the activation of the corresponding recovery policy as described in Section 3.3.6. The summary provided in the table, shows that the quad-core platform is able to better tolerate a single bug with respect to the dual-core platform. This is due to the different design choices performed for the system bus on the two MPSoCs. The bus used on the quad-core platform has a higher bandwidth with respect to the dual-core platform, therefore the interference introduced by a single buggy task is tolerated. However, the tolerance rapidly decrease with the number of affected tasks, as showed by the subsequent lines in the table. Moreover, when a single task is affected by the bugs, activations of the warning rule are more frequent than activations of the alarm rule. Since the recovery action executed in response to the warning rule introduces a lower performance overhead than the recovery action for the alarm rule, this means an overall lower

Table 4.3 Temporal partitioning detection rules activations. The *affected tasks* column refers to the number of non-critical tasks affected by the artificial bug at the same time.

Platform	Affected tasks	Warning	Alarm	NE
Dual-core	1	2 (0.01 %)	14859 (99.06 %)	139 (0.93 %)
Quad-core	1	1114 (7.43 %)	1668 (11.12 %)	12218 (81.45 %)
	2	528 (3.52 %)	11348 (75.65 %)	3124 (20.83 %)
	3	0	14490 (99.93 %)	10 (0.07 %)

performance overhead of the temporal partitioning on the quad-core platform. In such experiments, the occurrence of a bug in the critical application has not been considered for two reasons:

1. the critical application is developed and tested according to the requirements of the highest safety level of the reference standard, therefore bugs should be proven non-existent;
2. even if a bug is present in the critical application, it would be detected by either the WDP or the SWDT, triggering a recovery action.

Moreover, the mitigating effects that quality-of-service (QoS) policies might have on this kind of bug are not considered. The reason is that QoS policies implemented on most COTS MPSoCs are designed for the consumer electronic markets; therefore, they are not able to guarantee hard real-time constraints and cannot be used in the type of applications considered in the scope of this thesis.

4.5 Evaluating ODIn for a Space Application

4.5.1 Dual-Core Platform

The dual-core platform used for the experiments is described in detail in Section 4.1.1. The space benchmark deployed on this system is composed of two applications:

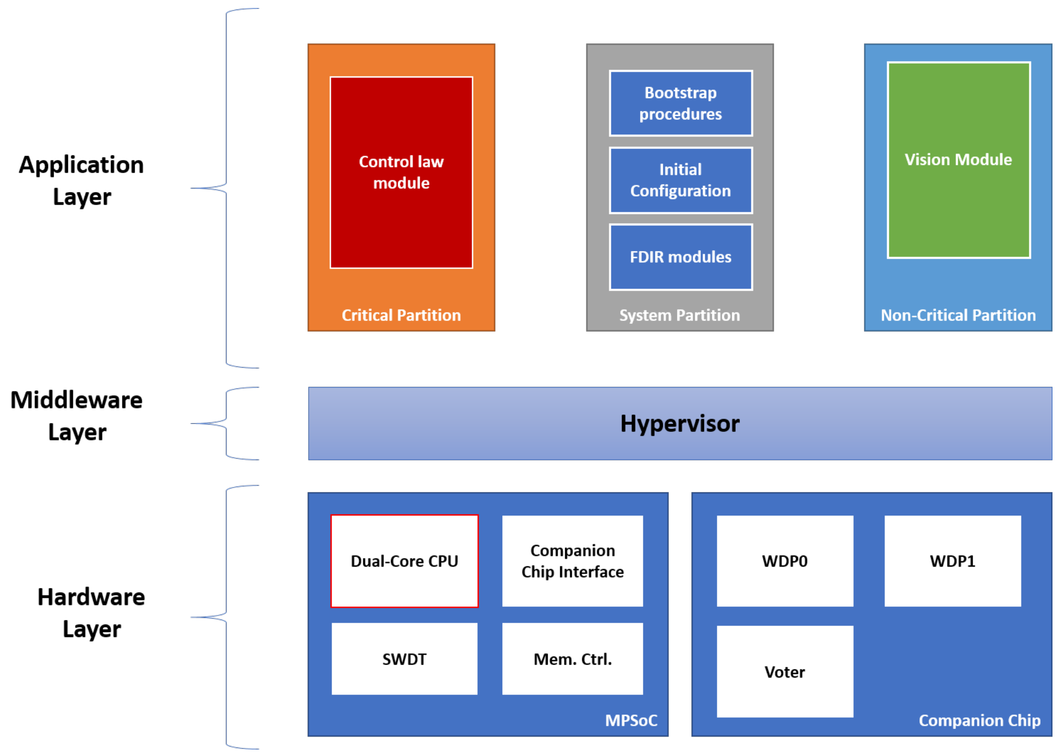


Fig. 4.6 System-level block diagram for space dual core implementation.

1. the critical *control law* module,
2. the non-critical *vision* module.

The vision module, in this implementation, performs a RICE compression of a fixed frame. The frame is read from a memory location, which represents the destination buffer of a camera. The purpose of this application is to reduce the communication bandwidth necessary for real-time communication with a ground station, which is a common requirement in space applications.

The control law module implements a simple controller designed to keep the main sensor camera pointed in a fixed direction during the orbit. The module receives sensors data on a serial interface and produces command to a set of actuators through a similar serial interface. It can also accept an override coming from a different module (not implemented in the benchmark), which receives commands from the ground station to direct the camera in a new direction.

The ODIn-S configuration on this platform is as follows:

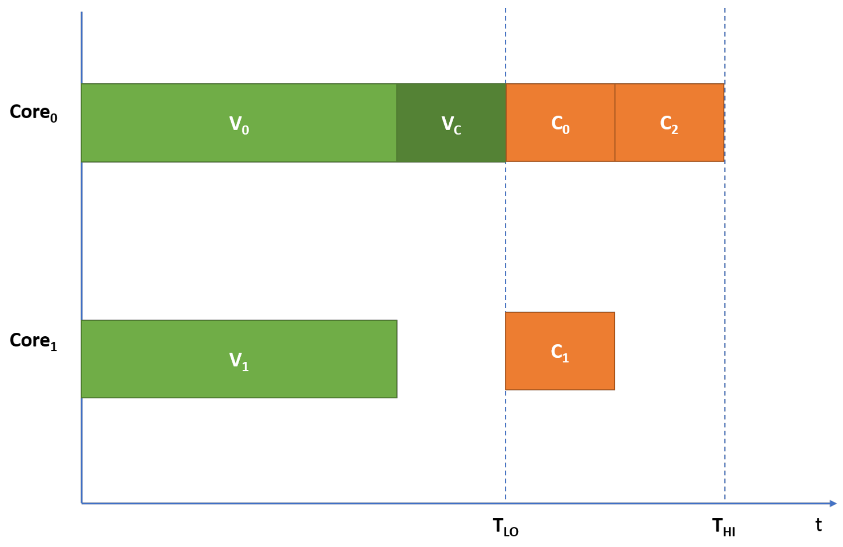


Fig. 4.7 Nominal system scheduling for the space application on the dual-core platform.

Software layer. In this implementation, the ODIn architecture requires implementation of a TTMR for the vision module and a TMR for the control-law module. Therefore the task set on the system is the following:

- V_0, V_1, V_2 are the three replicas of the vision module;
- V_C implements the comparator and the software majority voter used in the TTMR for the vision module;
- C_0, C_1, C_2 are the three replicas of the control-law module.

. The partitions required in this implementation are eight:

- three non-critical partitions to contain the three replicas of the vision module;
- an additional non-critical partition to contain the V_C task;
- three critical partition to contain the three replicas of the control-law module.
- a system partition to contain the bootstrap software and the recovery actions implementations.

The configuration of the software layer is performed by the system integrator at the end of the development of the application software and hardware components.

Middleware layer. In this implementation, the middleware layer - i.e., the type-1 hypervisor - is configured to enforce containment of the partitions described in the software layer. Such configuration includes the static mapping of each application to a subset of the available processing cores. More precisely:

- V_0 is mapped to core 0;
- V_1 is mapped to core 1;
- V_2 is mapped to core 0;
- V_C is mapped to core 0;
- C_0 is mapped to core 0;
- C_1 is mapped to core 1;
- C_2 is mapped to core 0;
- the system partition is allowed to execute on the first available core.

The middleware layer is also configured to implement the scheduling scheme needed by the application software. The scheduling is reported in Fig. 4.7.

Hardware layer. In this implementation, the hardware layer is composed of:

- six WDPs - controlling all partitions except the V_C partition and the system partition - and one SWDT. Both are IPs described in Section 3.4;
- a hardware majority voter, used by the TMR implementation;
- relevant performance counters.

In particular, the performance counters have been configured to observe the DCSC metric described in Section 3.3 for the control-law module. The performance counters in this implementation are up-counters that are able to issue an IRQ when reaching their maximum value. They are configured so that the maximum value is reached after the T_D events are counter - in this implementation, after the core running the critical application is stalled for more than T_D clock cycles waiting for data to be available in the local cache. The performance counters are also read by a module in the critical application (which should be integrated in the type-1 hypervisor in a real-life application) just before the application yields to the system scheduler. If the value read by this module is above T_W, a software counter is incremented. When such software counter reaches the α threshold, a graceful degradation recovery action is implemented by the system partition, which is in turn invoked by the critical partition through a software trap.

4.5.2 Quad-Core Platform

The quad-core platform used for the experiments is described in detail in Section 4.1.2. The space benchmark deployed on this system is composed of three applications:

1. the critical *control law* module;
2. the non-critical *vision* module;
3. the non-critical *sensor log* module;

The vision module, in this implementation, performs a RICE compression of a fixed frame. The frame is read from a memory location, which represents the destination buffer of a camera. The purpose of this application is to reduce the communication bandwidth necessary for real-time communication with a ground station, which is a common requirement in space applications.

The control law module implements a simple controller designed to keep the main sensor camera pointed in a fixed direction during the orbit. The module receives sensors data on memory-mapped buffer and produces command to a set of actuators through a serial interface. It can also accept an override coming from a different module (not implemented in the benchmark), which receives commands from the ground station to direct the camera in a new direction.

The sensor log module reads the same sensor data as the control module and stores them on a special permanent memory. Such memory is periodically read and sent to ground station for diagnostic purposes.

The ODIn-S configuration on this platform is as follows:

Software layer. In this implementation, the ODIn architecture requires implementation of a TTMR for the vision module, a different TTMR for the sensor log, and a TMR for the control-law module. Therefore the task set on the system is the following:

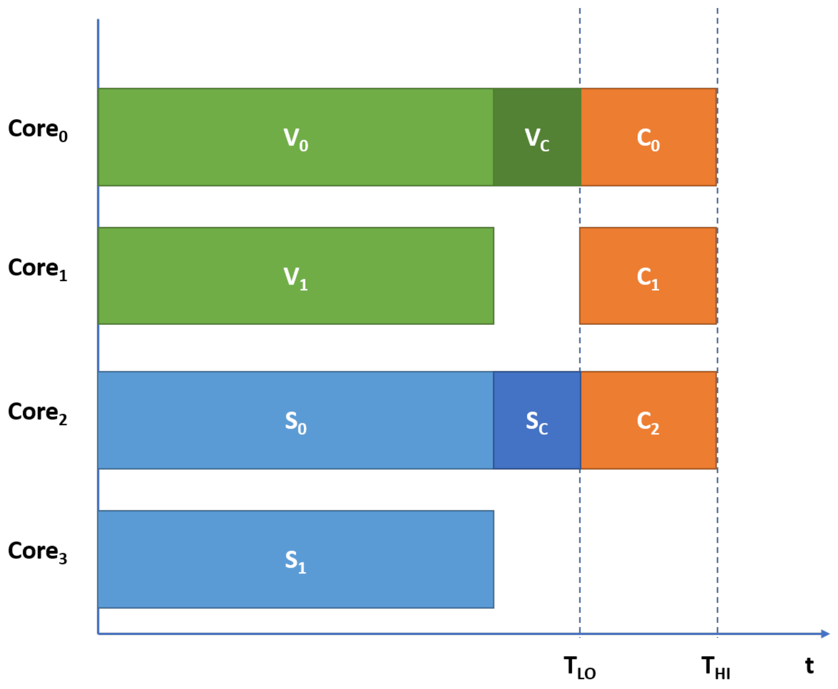


Fig. 4.8 Nominal system scheduling for the space application on the quad-core platform.

- V_0, V_1, V_2 are the three replicas of the vision module;
- V_C implements the comparator and the software majority voter used in the TTMR for the vision module;
- S_0, S_1, S_2 are the three replicas of the sensor log module;
- S_C implements the comparator and the software majority voter used in the TTMR for the sensor log module.
- C_0, C_1, C_2 are the three replicas of the control-law module.

. The partitions required in this implementation are twelve:

- three non-critical partitions to contain the three replicas of the vision module;
- an additional non-critical partition to contain the V_C task;
- other three non-critical partitions to contain the three replicas of the sensor log module;
- yet another non-critical partition to contain the S_C task;
- three critical partition to contain the three replicas of the control-law module.

- a system partition to contain the bootstrap software and the recovery actions implementations.

The configuration of the software layer is performed by the system integrator at the end of the development of the application software and hardware components.

Middleware layer. In this implementation, the middleware layer - i.e., the type-1 hypervisor - is configured to enforce containment of the partitions described in the software layer. Such configuration includes the static mapping of each application to a subset of the available processing cores. More precisely:

- V_0 is mapped to core 0;
- V_1 is mapped to core 1;
- V_2 is mapped to core 0;
- V_C is mapped to core 0;
- S_0 is mapped to core 2;
- S_1 is mapped to core 3;
- S_2 is mapped to core 2;
- S_C is mapped to core 2;
- C_0 is mapped to core 0;
- C_1 is mapped to core 1;
- C_2 is mapped to core 2;
- the system partition is allowed to execute on the first available core, with an implicit preference for core 3, since it is the less utilized core.

The middleware layer is also configured to implement the scheduling scheme needed by the application software. The scheduling is reported in Fig. 4.8.

Hardware layer. In this implementation, the hardware layer is composed of:

- nine WDPs - controlling all partitions except the V_C, S_C, and system partitions - and one SWDT. Both are IPs described in Section 3.4;
- a hardware majority voter, used by the TMR implementation;
- relevant performance counters.

Table 4.4 Hardware fault injection campaigns results on the quad-core platform for the space use-case.

Target	NE	CFE	TO	F	Inj.
CPU RF	97.30 %	1.12 %	1.58 %	0	10 000
Config.	97.13 %	1.83 %	1.04 %	0	10 000
Total	97.215 %	1.475 %	1.31 %	0	20 000

In particular, the performance counters have been configured to observe the DCSC metric described in Section 3.3 for the control-law module. The performance counters in this implementation are up-counters that are able to issue an IRQ when reaching their maximum value. They are configured so that the maximum value is reached after the T_D events are counter - in this implementation, after the core running the critical application is stalled for more than T_D clock cycles waiting for data to be available in the local cache. The performance counters are also read by a module in the critical application (which should be integrated in the type-1 hypervisor in a real-life application) just before the application yields to the system scheduler. If the value read by this module is above T_W, a software counter is incremented. When such software counter reaches the α threshold, a graceful degradation recovery action is implemented by the system partition, which is in turn invoked by the critical partition through a software trap.

4.5.3 Experiments and Results

Campaigns performed to evaluate ODIn-S are similar to those performed to evaluate ODIn-A, described in Section 4.4, although the software fault injection was skipped in this case, since TMR and TTMR, associated with software n-versioning [9], are able to completely mask effects of such faults. Results of the hardware fault injection campaigns performed on the space use-case are reported in Table 4.4.

The evaluation of the temporal partitioning scheme has not been repeated for this use-case, since there is no conceptual difference between its implementation in this use-case and the implementation in the avionic use case, therefore results would be qualitatively equivalent.

4.6 Overhead Evaluation

Experiments performed on both platforms allow to evaluate a performance overhead of the ODIn architecture with respect to a *vanilla* system. The performance overhead has been evaluated in terms of area overhead and performance overhead.

4.6.1 Area Overhead

The area overhead is mainly due to the additional code and hardware components to be added to the system in order to implement the ODIn architecture. The main contribution to the code size is the type-1 hypervisor, since it is a rather complex software module, comparable in all respects to an operating system. A separate analysis should be performed for any given type-1 hypervisor, therefore it has not been considered in this overhead analysis.

A different contribution to the code size is the software used to configure WDPs and SWDT at bootstrap and for their usage. The actual overhead of this modules depends on the system configuration - more specifically on the partitioning in blocks of each software modules - however it can be estimated as two memory writes for each block signature to be configured in each WDP and to one memory write to configure the SWDT. The usage code consists of one write at the end of each block to send the corresponding signature to relevant WDP, and one write to re-arm the SWDT. Also the voter requires a single memory write per task. Considering a 32bit ISA, this overhead can be estimated as:

- 12 bytes per signature per WDP;
- 8 bytes for the SWDT;
- 4 bytes for the Voter.

A further contribution to code memory size is the code used to detect temporal interference. This code is described in Algorithm 3.3. Its size depends on the compiler and on the target ISA. In the implementation used in this thesis, targeting a 32bit ARM processor, the size of this module was of 92 bytes.

Finally, the contribution of the ISRs used to respond to exceptions and recovery requests should be considered. ISRs used to respond to IRQs coming from WDPs

Table 4.5 Module code size overhead

Module	**Size**
WDP configuration, usage, and ISR	16 bytes/WDP
Voter usage	4 bytes
SWDT configuration and usage	8 bytes
Interference detection module (Algorithm 3.3)	92 bytes
Graceful degradation recovery	4 bytes
Hard recovery	8 bytes

Table 4.6 Total code size overhead

Use-case	**Dual-core**	**Quad-core**
Avionic	148 B	180 B
Space	196 B	228 B

monitoring non-critical partitions is composed of just one system call to reboot or halt the system partition, therefore its overhead is just 4 bytes per WDP. Recovery actions implementations are implemented just once. The contribution of the recovery actions is of 4 bytes for the graceful degradation, corresponding to just one system call. The hard recovery action is composed of one memory write activating the external signal that activates the stand-by spare computer, and one system call to halt the system, for a total of 8 bytes.

The total code size overhead is reported in Table 4.6 for all implementations considered in this thesis.

A different kind of area occupation is the area required for the implementation of support hardware on the companion chip. Considering an FPGA implementation, the area occupation of one WDP is of 260 look-up tables (LUTs), which is the basic implementation block in FPGA technology. Considering that a typical modern FPGA contains tens of thousands of LUTs, the area occupation of a WDP is rather small.

4.6.2 Performance Overhead

ODIn's detection mechanisms require execution of very few instructions; therefore, measuring performance overhead is quite hard. Nonetheless, overhead can be estimated in a few clock cycles, which is a negligible overhead.

A heavier overhead can be introduced by recovery policies. In particular, the graceful degradation recovery, removes parts of system functionality - albeit non-critical ones. This may result in a performance degradation for the end user. On the other hand, the hard recovery should not introduce any perceivable performance overhead, since the switch to a stand-by spare is managed entirely in hardware.

4.7 Chapter Summary

In this chapter, the experimental evaluation of the ODIn architecture has been described. First of all, the target hardware platforms have been described. Then, the components of the benchmark applications used in the experimental evaluation have been described. Such components are the RICE compression algorithms, which is a loss-less compression algorithm based on the RICE encoding, the Sobel edge detection algorithm, based on the Sobel filter, control-law modules, and sensor logging modules. After that, the fault injection system used for the experiments has been described, following the general architecture of a fault injection system described in Section 1.3. Finally, the experimental evaluation has been described for both the ODIn-A and ODIn-S. For each architecture, the configuration on both target hardware platforms has been described before presenting and discussing the experimental results. At the very end of the chapter, the area and performance overhead estimations for the ODIn architecture are presented.

Experimental results show that the ODIn architecture, in both avionic and space use-cases, can make a system immune to the faults considered in the experiments, which are representative of a large class of faults. Moreover, the overhead introduced by the architecture implementation is very limited. The limited size of both additional code and hardware IPs required by ODIn, contribute to reduce the cost of an actual certification proposal, by reducing the time needed to perform the kind of detailed analysis required by certification agencies.

Chapter 5

Conclusions and Future Developments

5.1 ODIn: summary of the architecture and its validation

Avionics and space applications pose several peculiar challenges, different from those posed to most other industry domains, due to the very strict safety requirements. In particular, the avionic industry has a very strict certification process and must adhere to several standards, as those cited in Section 1.4.

As a consequence, avionic and space industries are struggling with the adoption of technologies like MPSoCs, which are already widespread in other domains such as consumer electronics, IoT, etc. . . Most applications in the cited industries, are based on single-core SoC architecture, which are being phased out of the market, forcing industries still using them to look for new solutions.

This thesis is part of the research effort that avionic and space industry players, together with institutions, have been making to find a way to use MPSoCs, keeping the needed adherence to the safety standards. The contribution of this thesis is towards a new reference architecture, to be used in the design of new certifiable systems based deployed on MPSoCs. Using results of previous research in the field of dependable embedded system design, verification, and validation, this thesis proposes a way to deploy several applications on a single hardware equipment based on a MPSoC, to exploit the parallelism provided by the underlying hardware architecture.

The main issue, in this effort, is the consolidation on the same hardware of applications with different levels of criticality. The choice of proposing a MCS in this thesis, is due to the collection of industrial requirements performed in the frame of the Artemis/ECSEL JU AIPP5 research program "EMC^2". Cooperation with the industrial partners in the project, especially with the team involved in the avionic tasks of the project, allowed collection of a set of requirements, though non-formal, that guided the design of the ODIn architecture described in this thesis.

The main requirement - beside the adherence to safety standards - is the reduction of SWaP consumption, by exploiting the parallelism of a single equipment to implement several applications. Although it is in theory possible to deploy several critical applications on a single MPSoC, the possible occurrence of common mode errors capable of disrupt functionality of several safety-critical applications at once, makes this an unwise option. Therefore, implementation of a MCS was the better option to satisfy the SWaP reduction requirement.

A MCS is a system composed of several applications, which can be classified at different levels of criticalities. The application of the ODIn architecture, is a MCS with one application classified at a high criticality level - e.g., DAL-A - and several other applications classified at a lower criticality level - e.g., DAL-C or DAL-D. This choice allows to integrate on a single equipment as much as N applications, with N number of the cores in the equipment, of which at most one is classified as a DAL-A application - or an equivalent level of a different safety standard. This allows a reduction of a factor N of the needed OBE, for a given system. However, it is not possible to just deploy applications on a MPSoC: some provisions to keep the system adherent to the safety standards must be made.

The issue of system scheduling and feasibility when considering a MCS has been a active research topic in the past decade, as discussed in Section 2.1. However, most of such solutions where just considering the scheduling problem, and - to the best of our knowledge - very few efforts were in the direction of a reliable MCS, as discussed in the aforementioned section.

5.1.1 ODIn architecture

This thesis is not focused on the system scheduling issue - for which several solutions were proposed - instead, it is focused on the partitioning problem, as defined in Section 3.1.1, where the partitioning problem is subdivided in *spatial* and *temporal* partitioning. Once identified such division, each problem is tackled separately, then the solutions are integrated to achieve a comprehensive reference architecture, which has been named the *online detection of interference* (ODIn) architecture.

The spatial partitioning problem, has been solved by using the functionalities provided by the MMU, which is a hardware component of most CPU micro-architectures. The MMU can be in theory managed by the application itself; however, direct management of different MMUs in different micro-architectures can result in a non-negligible design overhead. Therefore, the ODIn architecture, relies on the functionalities provided by a type-1 hypervisor. A type-1 hypervisor is a virtualization software running directly on the MPSoC, without interposition of a host operating system. Commercial type-1 hypervisors are available for most micro-architectures, and can be ported to new MPSoCs with limited effort - provided that cores' micro-architectures remains the same. Moreover, some commercial type-1 hypervisors are actively trying to achieve certification of their product for several safety standards. Using a type-1 hypervisor, ODIn can define a set of resource partitions, each containing a portion of the whole MCS. By design, ODIn can support different levels of granularities - i.e., a partition can contain a single task, an application, a set of applications at the same criticality level - although it has been so far validated only for the application granularity level - i.e., each partition contains a single application. When performing system integration, the designer should identify the resource partitions, keeping in mind that each application can only access resource that have been granted to its partition. Moreover, the cores of the MPSoC can be considered as resources as well, therefore an application can only run on the core(s) granted to its partition.

The temporal partitioning problem has been solved using a bounded interference approach, composed of an offline analysis phase and an online detection mechanism or safety net. The interference that an application running on a MPSoC suffers by applications running in parallel on the same MPSoC, is difficult to measure precisely. Moreover, there are several sources of non-determinism that makes this analysis very complex. As a consequence, WCET estimation on MPSoCs tends to be very pessimistic, causing a severe under-utilization of the hardware resources in the system. Several proposals have been made to ease this analysis, as described in Section 2.1.3. The solution adopted for the ODIn architecture, can be classified as a bounded interference approach.

The offline phase of this approach is based on a statistical profiling of a metric which is sensitive to interference introduced by applications running in parallel with a monitored one. The metric is measured a significant number of times in nominal conditions. In such conditions, there is an expected level of interference on the observed metric, including a variance due to both error measurements and intrinsic non-determinism sources. Once the metric has been statistically characterized in such conditions - i.e., once a distribution fitting the measures is found - a set of three thresholds can be determined, according to the following definitions (see Section 3.3):

$$T_D : P(X \geq T_D) \leq C_D \iff F_X(T_D) \geq (1 - C_D)$$

$$T_W : P(X \geq T_W) \leq C_W \iff F_X(T_W) \geq (1 - C_W)$$

$$\alpha : [P(T_W \leq X \leq T_D)]^\alpha = (C_W - C_D)^\alpha = C_\delta^\alpha$$

The T_D threshold or *detection* threshold is used to detect a sudden interference able to cause a deadline miss in the monitored task. The T_W and α thresholds are used together. The T_W threshold delimits a warning region, $[T_W, T_D]$. If the metric is measured in this region, an unexpected interference might be active in the system; however, the probability that the metric is measure in this region without any unexpected interference; to detect the presence of such a subtle interference, the α threshold is used, which is the maximum number of consecutive times that the metric can be measured in the warning region before assuming that an unexpected interference is active in the system.

From the definitions the following operative equations, to be solved numerically, can be derived to compute the thresholds knowing the statistical characterization of the observed metric:

$$\int_{-\infty}^{T_D} f_X(x)\mathrm{d}x = 1 - C_D$$

$$\int_{-\infty}^{T_W} f_X(x)\mathrm{d}x = 1 - C_W$$

$$\alpha = \left\lceil \frac{ln\,(1-C_G)}{ln\,C_\delta} \right\rceil$$

Such equations can be simplified if the metric is characterized by a normal distribution, as below, where μ is the mean of the standard distribution characterizing the metric, σ is its standard deviation, and $\Phi(x)$ is the standard normal CDF:

$$T_D = \mu + 3\sigma$$

$$T_W = \mu + 2\sigma$$

$$\alpha = \left\lceil \frac{ln\,(1-C_G)}{ln\,(\Phi(3)-\Phi(2))} \right\rceil$$

The online safety-net used in the ODIn architecture, exploits the performance counters available in many MPSoC architectures. Such counters are usually provided for debug and profiling purposes. ODIn requires that on the target architecture, such counters can be configured and read through the ISA, as opposed as through a debug interface. The performance counter are configured to monitor the selected metric at bootstrap. The detection module, which is proposed for addition to the operating system scheduler, reads the value of the performance counter when the monitored task yields control to the scheduler to check if it is in the warning region. The performance counters are configured to raise an IRQ in case the metric is above T_D even during execution.

Besides this method to detect interference, ODIn uses two additional detection mechanism. Both mechanisms uses a kind of watchdog. The first mechanism uses a SWDT, which is a watchdog timer able to send an external signal when triggered. The second mechanism use a set of WDPs. Each WDP monitors one application, applying a CFC strategy based on predefined block signatures.

The recovery actions implemented in ODIn are of two types:

1. *graceful degradation*, reacts to the IRQ of a WDP monitoring a non-critical application or to the warning rule of the interference detection.
2. *hard recovery*, reacts to the SWDT, to the WDP of the critical application, or to the alarm rule of the interference detection.

5.1.2 Experimental validation

The experimental platforms were designed considering a benchmark application which was representative of a typical application in the two considered domains: avionic and space.

Both benchmark applications were deployed on two different hardware platforms:

1. the Xilinx Zynq-7000 APSoC, which is a dual-core MPSoC with a companion FPGA integrated on the same chip;
2. the NXP i.MX6Q, which is a quad-core MPSoC. The i.MX6Q was deployed on a development board designed by SanitasEG: the Inventami board, which features a companion Lattice FPGA to implement the functionalities of the companion chip, as required by the ODIn architecture.

All versions of the benchmark features the same modules arranged in different ways to respond to specific requirements. Such modules implement:

- a compression algorithm based on the RICE encoding - also referred to as the *RICE compression algorithm*;
- an edge detection algorithm based on the Sobel filter - also referred to as the *Sobel edge detection algorithm*;
- a sensor log module;
- a control law module.

The control law module represents the critical application in all evaluated versions of ODIn, whereas the other modules were composed in different ways for each version.

The dual-core avionic benchmark. This benchmark was composed of the control law module and of a RICE compression algorithm module. The two applications were integrated in the ODIn-A architecture as described in Section 4.4.1.

The quad-core avionic benchmark. This benchmark was composed of the control law module, the RICE compression algorithm, the Sobel edge detection algorithm, and the sensor log module. All applications were integrated in the ODIn-A architecture as described in Section **??**.

The dual-core space benchmark. This benchmark was composed of the control law module and of the RICE compression algorithm. Both applications were integrated in the ODIn-S architecture as described in Section **??**.

The quad-core space benchmark. This benchmark was composed of the control law module, the RICE compression algorithm, and the sensor log module. All applications were integrated in the ODIn-S architecture as described in Section 4.5.2.

Each version of the final application was evaluated through fault injection campaigns. There were three types of fault injection campaigns:

1. hardware fault injection, emulating effects of SEE on the hardware;
2. software fault injection, emulating effects of bugs on non-critical applications;
3. artificial bug injection, emulating effects of bugs in non-critical applications which introduced an unexpected interference on the critical application.

All campaigns were performed through a dedicated fault injection system, whose designed in detailed in Section 4.3.1.

5.1.3 Conclusions

Results of the campaigns, presented in Section 4.4.3 and 4.5.3, show that the ODIn architecture is resilient to both hardware and software faults, and that the implemented interference detection strategy is able to reduce the impacts of unexpected interference by triggering appropriate recovery strategies.

The level of reliability reached by applications integrated in the ODIn architecture, as it can be inferred from fault injection campaigns results, show that a system developed according to such architecture can be a valid proposal for certification of a new avionic equipment, or a valid basis for a new contract between a space agency and an industrial contractor.

The main goal of this thesis, which was to propose a new reference architecture for avionic and space applications deployed on MPSoC, is achieved by means of hardware and software strategies integrated on a single system architecture, which is able to integrate at least as many applications as there are core on the underlying MPSoC architecture. Although the validation performed on the scope of this thesis is only referred to specific industrial requirements, nothing forbids integration of an even higher number of applications on a single MPSoC. The partitioning strategies provided by ODIn, would allow to treat each core as a separate single-core system, allowing integration of entire multi-application equipment - developed according to IMA - on a single MPSoC-based equipment.

5.2 Future Developments

5.2.1 Automotive

Besides the avionic and space use cases considered throughout this thesis, there is a new push towards high performance in embedded systems coming from the automotive industry. The trend in consolidating infotainment functionalities – such as radio, navigation, traffic information – with the instrument cluster – including safety critical information such as oil level, water temperature, safety warnings – is a clear example of a mixed criticality workload. Such systems could greatly benefit from the use of MPSoC in a way similar to the avionic systems analyzed in this thesis. In this case, besides the reduction of SWaP, another favorable outcome of adopting MPSoCs would be the reduction of the unit cost, which is far more important to the relative high-volumes of automotive when compared with the very low volumes of avionics.

Another requirement for an increase in performance in the automotive industry comes from the autonomous driving trend. In this case, several components such as vision algorithms or sensor fusion algorithms are required to provide data to control algorithms designed to drive the vehicle without any human intervention. This use case would benefit from the introduction of MPSoCs as well.

Since in both cited automotive use-cases, safety requirements are similar to those used to develop the ODIn-A architecture, the solution proposed in this thesis could be adapted for use in an automotive context.

5.2.2 Next-Gen MPSoCs

Current SoCs have seen a remarkable increase in complexity and number of integrated peripherals. This phenomenon required the adoption of a new architecture for on-chip communications, since the limits of the bus-based communication were to be overcome quickly [84]. The main limit of bus-based architectures, is that bus do not scale very well with the number of connected nodes; moreover, most bus architecture require introduction of an arbitration policy to solve contentions. Complex arbitration policies introduce jitter - i.e., they are a source of non-deterministic behavior - and lowers overall throughput [85]. Solutions like crossbars have been explored, but they introduce a non-negligible area and power overhead [86].

To solve such issues, the network-on-chip (NoC) architecture has been proposed. Using concepts from the computer networks, NoCs can interconnect a very high number of nodes and allow for new MPSoC architectures to integrate even a higher number of IP cores. This new solution is of course interesting from the perspective of MCS integration, which is the main focus of this thesis.

Using a NoC-based MPSoC to integrate an MCS, requires analyzing the NoC itself as a shared resource and several efforts have been made in this direction in the literature [87] [88][89][90][91], which can be paired with other effort in the direction of fault tolerant NoC-based systems [92][93]. Most of such solutions doe not fully satisfy requirements of an application as those the ODIn architecture was developed to integrate.

To answer to such requirements, in the future the ODIn architecture should be ported and adapted to work on COTS NoC-based MPSoCs. The requirement of being deployed on COTS NoC-based MPSoCs, forbids the introduction of special purpose hardware in the NoC itself, which might include sophisticated adaptive routing algorithm for fault-tolerance, or special scheduling strategies for the communication happening on the NoC. Without such hardware support, a successful port of ODIn to NoC-based MPSoC should only rely on peripheral hardware and common-place monitoring hardware in such MPSoCs; these are very similar requirements to those that driven the development of the ODIn architecture.

The ODINoC architecture as already been outlined in [94] and is currently under development.

References

[1] J. Rushby. Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance. Technical report, NASA Langley Research Center, 1999.

[2] Position Paper CAST-32, Multi-core Processors. Technical report, Certification Authorities Software Team, 2014.

[3] Xavier Jean, Mark Gatti, Guy Berthon, and Marc Fumey. MULCORS - Use of Multicore Processors in airborne systems. Technical report, European aviation safety agency, 2012.

[4] Werner Weber, Alfred Hoess, Jan van Deventer, Frank Oppenheimer, Rolf Ernst, Adam Kostrzewa, Philippe Dore, Thierry Goubier, Haris Isakovic, Norbert Druml, Egon Wuchner, Daniel Schneider, Erwin Schoitsch, Eric Armengaud, Thomas Soderqvist, Massimo Traversone, Sascha Uhrig, Juan Carlos Perez-Cortes, Sergio Saez, Juha Kuusela, Mark van Helvoort, Xing Cai, Bjorn Nordmoen, Geir Yngve Paulsen, Hans Petter Dahle, Michael Geissel, Jurgen Salecker, and Peter Tummeltshammer. The EMC2 Project on Embedded Microcontrollers: Technical Progress after Two Years. In *2016 Euromicro Conference on Digital System Design (DSD)*, pages 524–531, 2016.

[5] International Electrotechnical Commission. Functional safety of electrical/electronic/programmable electronic safety related systems, 2000.

[6] RTCA Inc. RTCA/DO-178C Software Considerations in Airborne Systems and Equipment Certification., 2011.

[7] International Standardization Organization. ISO 26262 Road vehicles — Functional safety, 2011.

[8] Landwehr Carl Algirdas Avižienis, Laprie Jean-Claude, Randell Brian. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[9] O. Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. *Software-Implemented Hardware Fault Tolerance*. Springer Science and Business Media, 2006.

[10] Cristian Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, 2003.

[11] Automotive Electronics Council. AEC-Q standards. http://www.aecouncil.com/AECDocuments.html. Accessed: 2018-02-12.

[12] Jeffrey A Clark and Dhiraj K Pradhan. Fault injection: A method for validating computer-system dependability. *Computer*, 28(6):47–56, 1995.

[13] V Pouget, A Douin, D Lewis, P Fouillat, Université Bordeaux, G Foucard, P Peronnard, V Maingot, J B Ferron, L Anghel, R Leveugle, and R Velazco. Tools and Methodology Development for Pulsed Laser Fault Injection in SRAM-based FPGAs. *Methodology*, 2007.

[14] Johan Karlsson, Peter Lidén, Peter Dahlgren, Rolf Johansson, and Ulf Gunneflo. Using Heavy-lon Radiation to Validate Fault-Handling Mechanisms. *IEEE Micro*, 14(1):8–23, 1994.

[15] Ghassem Miremadi and Jan Torin. Evaluating Processor-Behavior and Three Error-Detection Mechanisms Using Physical Fault-Injection. *IEEE Transactions on Reliability*, 44(3), 1995.

[16] E. L. Petersen. The SEU figure of merit and proton upset rate calculations. *IEEE Transactions on Nuclear Science*, 35, 1998.

[17] Q. Chen, D. M. Hiemstra, H. Wang, L. Chen, and V. Kirischian. High Energy Proton Irradiation Results for the DSP Cores of the KeyStone II System-on-Chip (SoC) 66AK2L06. In *Radiation Effects Data Workshop (REDW), 2017 IEEE*, 2017.

[18] H. Wang, Q. Chen, L. Chen, D. M. Hiemstra, and V. Kirischian. Single Event Upset Characterization of the Tegra K1 Mobile Processor Using Proton Irradiation. In *Radiation Effects Data Workshop (REDW), 2017 IEEE*, 2017.

[19] J. M. Bird, M. K. Peters, T. Z. Fullem, M. J. Tostanoski, T. F. Deaton, K. Hartojo, and R. E. Strayer. Neutron Induced Single Event Upset (SEU) Testing of Commercial Memory Devices with Embedded Error Correction Codes (ECC). In *Radiation Effects Data Workshop (REDW), 2017 IEEE*, 2017.

[20] D. Lambert, F. Desnoyers, D. Thouvenot, O. Riant, J. Galinat, B. Azaïs, and T. Colladant. Single Event Upsets Induced by a few MeV Neutrons in SRAMs and FPGAs. In *Radiation Effects Data Workshop (REDW), 2017 IEEE*, 2017.

[21] Stefano Esposito and Massimo Violante. System-Level Architecture for Mixed Criticality Applications on MPSoC : A Space Application. In *Proceedings of the 2017 IEEE International Workshop on Metrology for AeroSpace (MetroAeroSpace)*, pages 479–483. IEEE, 2017.

[22] Eric Jenn, Jean Arlat, Marcus Rimen, Joakim Ohlsson, and Johan Karlsson. Fault injection into vhdl models: the mefisto tool. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 66–75. IEEE, 1994.

[23] Jérome Boué, Philippe Pétillon, and Yves Crouzet. Mefisto-l: a vhdl-based fault injection tool for the experimental assessment of fault tolerance. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 168–173. IEEE, 1998.

[24] Todd A Delong, Barry W Johnson, and Joseph A Profeta Iii. A fault injection technique for vhdl behavioral-level models. *IEEE Design & Test of Computers*, 13(4):24–33, 1996.

[25] Daniel Gil, R Martinez, JV Busquets, Juan Carlos Baraza, and Pedro J Gil. Fault injection into vhdl models: Experimental validation of a fault-tolerant microcomputer system. In *Dependable Computing—EDCC-3*, pages 191–208. Springer, 1999.

[26] Luis Berrojo, Isabel González, Fulvio Corno, Matteo Sonza Reorda, Giovanni Squillero, Luis Entrena, and Celia Lopez. New techniques for speeding-up fault-injection campaigns. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 847–852. IEEE, 2002.

[27] Synopsys Inc. TetraMAX ATPG. http://www.synopsys.com/Tools/Implemen–tation/RTLSynthesis/Test/Pages/TetraMAXATPG.aspx. Accessed: 2018-02-12.

[28] B Parrotta, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. New techniques for accelerating fault injection in vhdl descriptions. In *On-Line Testing Workshop, 2000. Proceedings. 6th IEEE International*, pages 61–66. IEEE, 2000.

[29] Joao Carreira, Henrique Madeira, João Gabriel Silva, et al. Xception: Software fault injection and monitoring in processor functional units. *Dependable Computing and Fault Tolerant Systems*, 10:245–266, 1998.

[30] RTCA Inc. RTCA/DO-254 Design Assurance Guidance for Airborne Electronic Hardware, 2000.

[31] SAE International. ARP4761 - Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. SAE International, 1996.

[32] Institute of Electrical and Electronics Engineers. IEEE Std 1228-1994 - IEEE Standard for Software Safety Plans, 2010.

[33] European Cooperation for Space Standardization. ECSS-Q-ST-40C Rev.1 – Safety. http://ecss.nl/standard/ecss-q-st-40c-rev-1-safety-15-february-2017, 2017. Accessed: 2018-02-12.

[34] European Cooperation for Space Standardization. ECSS-Q-ST-80C – Software product assurance. http://ecss.nl/standard/ecss-q-st-80c-software-product-assurance, 2009. Accessed: 2018-02-12.

[35] European Cooperation for Space Standardization. ECSS-Q-ST-40-02C – Hazard analysis. http://ecss.nl/standard/ecss-q-st-40-02c-hazard-analysis, 2013. Accessed: 2018-02-12.

[36] European Cooperation for Space Standardization. ECSS-Q-ST-60C Rev.2 – Electrical, electronic and electromechanical (EEE) components. http://ecss.nl/standard/ecss-q-st-60c-rev-2-electrical-electronic-and-electromechanical-eee-components-21-october-2013/, 2008. Accessed: 2018-02-12.

[37] Christopher B. Watkins and Randy Walter. Transitioning from federated avionics architectures to Integrated Modular Avionics. In *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, 2007.

[38] Integrated modular avionics. In *Aerospace and Electronics Conference, 1992. NAECON 1992., Proceedings of the IEEE 1992 National*, volume 1, pages 39–45, 1992.

[39] Paul J. Prisaznuk. Arinc 653 role in Integrated Modular avionics (IMA). In *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, 2008.

[40] A. Mahmood and E.J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37(2):160–174, 1988.

[41] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings - Real-Time Systems Symposium*, pages 239–243, 2007.

[42] Alexandre Esper, Geoffrey Nelissen, Vincent Nélis, and Eduardo Tovar. How realistic is the mixed-criticality real-time system model? In *Proceedings of the 23rd International Conference on Real Time and Networks Systems - RTNS '15*, pages 139–148, 2015.

[43] Sanjoy Baruah and Steve Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Proceedings - Euromicro Conference on Real-Time Systems*, pages 147–155, 2008.

[44] JH Anderson, SK Baruah, and BB Brandenburg. Multicore operating-system support for mixed criticality. In *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.

[45] Malcolm S. Mollison, Jeremy P. Erickson, James H. Anderson, Sanjoy K. Baruah, and John A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Proceedings - 10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, ScalCom-2010*, pages 1864–1871, 2010.

[46] Angeliki Kritikakou, Olivier Baldellon, Claire Pagetti, Christine Rochange, Matthieu Roy, and Fabian Vargas. Monitoring on-line timing information to support mixed-critical workloads. In *2013 IEEE Real Time Systems Symposium*, pages 9–10, 2013.

[47] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *2013 Proceedings of the International Conference on Embedded Software, EMSOFT 2013*, 2013.

[48] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. Mapping mixed-criticality applications on multi-core architectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*, pages 1–6, 2014.

[49] C M Krishna. Fault-Tolerant Scheduling in Homogeneous Real-Time Systems. *ACM Computing Surveys*, 46(4):48:1–48:34, 2014.

[50] Michael Paulitsch, Oscar Medina Duarte, Hassen Karray, Kevin Mueller, Daniel Muench, and Jan Nowotsch. Mixed-criticality embedded systems-A balance ensuring partitioning and performance. *Proceedings - 18th Euromicro Conference on Digital System Design, DSD 2015*, pages 453–461, 2015.

[51] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing predictability on multi-processor systems with shared resources. In *Embedded Systems Week-Workshop on Reconciling Performance with Predictability*, pages 87–92, 2009.

[52] Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic execution model on COTS hardware. In *Architectures of Computing Systems*, pages 98–110, 2012.

[53] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for COTS-based embedded systems. In *Real-Time Technology and Applications - Proceedings*, pages 269–279, 2011.

[54] Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *Proceedings - Euromicro Conference on Real-Time Systems*, pages 109–118, 2014.

[55] Jan Nowotsch, Michael Paulitsch, Arne Henrichsen, Werner Pongratz, and Andreas Schacht. Monitoring and WCET analysis in COTS multi-core-SoC-based mixed-criticality systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*, pages 1–5, 2014.

[56] Ankit Agrawal, Gerhard Fohler, Jan Nowotsch, and Sascha Uhrig. Slot-Level Time-Triggered Scheduling on COTS Multicore Platform with Resource Contentions. page 8641, 2016.

[57] Moritz Neukirchner, Philip Axer, Tobias Michaels, and Rolf Ernst. Monitoring of workload arrival functions for mixed-criticality systems. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 88–96. IEEE, 2013.

[58] Moritz Neukirchner. *Establishing Sufficient Temporal Independence Efficiently. A monitoring approach*. PhD thesis, Technische Universität Braunschweig, 2014.

[59] R. Hillman, G. Swift, P. Layton, M. Conrad, C. Thibodeau, and F. Irom. Space processor radiation mitigation and validation techniques for an 1, 800 MIPS processor board. *European Space Agency, (Special Publication) ESA SP*, 2003(536):347–352, 2004.

[60] D. R. Czajkowski, M. P. Pagey, P. K. Samudrala, M. Goksel, and M. J. Viehman. Low power, high-speed radiation hardened computer & flight experiment. In *IEEE Aerospace Conference Proceedings*, 2005.

[61] Michel Pignol. DMT and DT2: Two Fault-Tolerant Architectures developerd by CNES for COTS-based Spacecraft Supercomputers. In IEEE, editor, *12th IEEE International Online Test Symposium*, 2006.

[62] Stefano Esposito, Cristian Albanese, Monica Alderighi, Fabio Casini, Luca Giganti, Maria Livia Esposti, Claudio Monteleone, and Massimo Violante. COTS-Based High-Performance Computing for Space Applications. *IEEE Transactions on Nuclear Science*, 62(6), 2015.

[63] Guy Berthon, Marc Fumey, Xavier Jean, Helen Misson, Laurence Mutuel, and Didier Regis. White Paper on Issues Associated with Interference Applied to Multicore Processors. Technical Report January, Federal Aviation Agency, 2016.

[64] Certification Authorities Software Team (CAST) Position Paper CAST-32A. Technical report, Certification Authorities Software Team, 2016.

[65] Sehry Avramenko, Stefano Esposito, Massimo Violante, Marco Sozzi, Massimo Traversone, Marco Binello, and Marco Terrone. An Hybrid Architecture for Consolidating Mixed Criticality Applications on Multicore Systems. In *2015 IEEE 21st International On-Line Testing Symposium*, pages 26–29, Halkidiki, 2015. IEEE.

[66] Stefano Esposito and Massimo Violante. On the Consolidation of Mixed Criticalities Applications on Multicore Architectures. *Journal of Electronic Testing: Theory and Applications (JETTA)*, 33(65):65–76, 2017.

[67] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rives, and Clifford Stein. *Introduction to Algorithm*. Boston, MA, 3 edition, 2009.

[68] ARM. *Cortex-A9 Technical Reference Manual, Issue I.* 2012.

[69] NXP. *e6500 Core Reference Manual, Rev. 0.* 2014.

[70] Jean D. Gibbons and Subharata Chakrabort. *Nonparametric Statistical Inference*. Boca Rato, FL, 5th edition, 2010.

[71] M. A. Stephens. EDF Statistics for Goodness of Fit and Some Comparisons. *Journal of the American Statistical Association*, 69(347):730–737, 1974.

[72] F. W. Scholz and M. A. Stephens. K-Sample Anderson-Darling Tests. *Journal of the American Statistical Association*, 82(399):918–924, 1987.

[73] Gregory W. Corder and Dale I. Foreman. *Nonparametric Statistics: A Step-by-Step Approach*. New York, NY, 2nd edition, 2014.

[74] Digilent. *ZedBoard Hardware User's Guide version 1.1*. 2012.

[75] Louise Crockett, Ross Elliot, Martin Enderwitz, and Bob Stewart. *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC*. Strathclyde Academic Media, first edition, 2014.

[76] Xilinx. *Zynq-7000 AP SoC Technical Reference Manual*. 2015.

[77] Inventami board homepage. http://www.inventami.com. Accessed 2018-10-01.

[78] NXP. *i.MX 6Dual/6Quad Applications Processor Reference Manual*. 2015.

[79] S. Provost, M. Le Roy, B. Mamdy, G. Flandin, and T. Paulsen. GAIA video processing embedded algorithms. Prototyping and validation activities. *European Space Agengy, Special Publication ESA SP*, 2007.

[80] Solomon W. Golomb. Run-Length Encodings. *IEEE Transactions on Information Theory*, (July):399–401, 1966.

[81] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons.

[82] Irwin Sobel and G Feldman. A 3x3 Isotropic Gradient Operator for Image Processing, 1968. presented at the Stanford Artificial Intelligence (SAIL) project.

[83] Henrique Madeira, Diamantino Costa, and Marco Vieira. On the emulation of software faults by software fault injection. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, number 2, pages 417–426, 2000.

[84] W.J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference*, pages 684–689, 2001.

[85] Thomas D. Richardson, Chrysostomos Nicopoulos, Dongkook Park, Vijaykrishnan Narayanan, Yuan Xie, Chita Das, and Vijay Degalahal. A hybrid SoC interconnect with dynamic TDMA-based transaction-less buses and on-chip networks. In *Proceedings of the IEEE International Conference on VLSI Design*, volume 2006, pages 657–664, 2006.

[86] Jinuk Luke Shin, Dawei Huang, Bruce Petrick, Changku Hwang, Kenway W. Tam, Alan Smith, Ha Pham, Hongping Li, Timothy Johnson, Francis Schumacher, Ana Sonia Leon, and Allan Strong. A 40 nm 16-core 128-thread sparc SPARC SoC processor. *IEEE Journal of Solid-State Circuits*, 46(1):131–144, 2011.

[87] Théodore Marescaux and Henk Corporaal. Introducing the SuperGT network-on-chip. In *Design Automation Conference*, pages 116–121, 2007.

[88] Evgeny Bolotin, Israel Cidon, Ran Ginosar, and Avinoam Kolodny. QNoC: QoS architecture and design process for network on chip. *Journal of Systems Architecture*, 50(2-3):105–128, 2004.

[89] J. Joven, A. Marongiu, F. Angiolini, L. Benini, and G. De Micheli. An integrated, programming model-driven framework for NoC-QoS support in cluster-based embedded many-cores. *Parallel Computing*, 39(10):549–566, 2013.

[90] Radu Andrei Stefan, Anca Molnos, and Kees Goossens. DAElite: A TDM NoC supporting QoS, multicast, and fast connection Set-Up. *IEEE Transactions on Computers*, 63(3):583–594, 2014.

[91] Benoît Dupont de Dinechin, Yves Durand, Duco van Amstel, and Alexandre Ghiti. Guaranteed Services of the NoC of a Manycore Processor. *International Workshop on Network on Chip Architectures 2014 (NoCArc'14)*, pages 11–16, 2014.

[92] Martin Radetzki, Chaochao Feng, Xueqian Zhao, and Axel Jantsch. Methods for fault tolerance in networks-on-chip. *ACM Computing Surveys*, 46(1):1–38, 2013.

[93] Sebastian Werner, Javier Navaridas, and Mikel Luján. A Survey on Design Approaches to Circumvent Permanent Faults in Networks-on-Chip. *ACM Computing Surveys*, 48(4):1–36, 2016.

[94] Stefano Esposito and Massimo Violante. Deterministic network on chip for deploying real time applications on many-core processors. In *2017 IEEE 23rd International Symposium on On-Line Testing and Robust System Design, IOLTS 2017*, pages 21–24, 2017.