ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

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

Network-on-Chip -based Multi-Processor System-on-Chip: Towards Mixed-Criticality System Certification

By

Serhiy Avramenko

******

Supervisor(s):
Prof. M. Violante

Doctoral Examination Committee:
Prof. A. Bosio, École Centrale de Lyon, France
Prof. M. Jenihhin, Tallinn University of Technology, Estonia
Prof. M. Ottavi, Università degli Studi di Roma Tor Vergata, Italy

Politecnico di Torino
2019

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Serhiy Avramenko
2019

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

To my family, for everything I am.

To my friends, for being part of my family.

Acknowledgements

I would like to thank my advisor M. Violante for his teachings, insights and patience.

I would also like to thank S. Esposito, for his precious insights.

Abstract

The usage of single-core–based microprocessors is slowly disappearing, while multi-core–based devices are becoming the only normally used nowadays. This is true also for the critical domains, like for instance avionics domain. However, the key point for such domains is that the multi-core microprocessors are de facto used as single-core devices. In fact, all-but-one executing cores are powered off. Meanwhile, the market offers more and more sophisticated solutions, proposing so called Multi-Processor System-on-Chip (MPSoC) device generation. These devices integrate tens or even hundreds of processing cores and peripherals on the same chip, connected through the network-on-chip (NoC) interconnection

The main obstacle for the usage of multi-core and MPSoC devices in the context of critical systems is the high certification effort required to ensure the proper level of dependability (and especially safety) for a system based on such kind of devices. Considering the commercial off-the-shelf (COTS) devices, these do not present any safety-dedicated features as the critical domains market size is relatively negligible. In fact, the high non-recurring engineering cost prevents the development of any COTS MPSoC specifically designed for the safety-critical applications.

This thesis presents several solutions to address the main dependability issues, preventing the usage of MPSoC-based systems in the critical systems context. In particular the avionics domain is considered, and the main focus is put upon the safety issues related to the usage of a shared interconnection, for what it concerns the temporal isolation between the software components. Although, the avionics domain is explicitly considered, the techniques presented are expected to be applicable for a generic safety-critical and even mission-critical domains, given that the required adaptation to the domain-specific peculiarities is done. As further contribution, this dissertation presents solutions to easier the dependability assessment of software components at executing core level.

Considering the executing core level, a proper fault tolerance against radiation induced soft errors was and remains crucial for the space applications. However, as the geometries keep shrinking and power saving techniques are becoming more and more aggressive, the issue of soft errors is no longer a space domain's prerogative. This thesis proposes an approach to rank a set of candidate software modules according to their intrinsic robustness to the soft errors. The main advantage of the proposed approach is that it's almost agnostic of the actual architectural details of the executing platform as it is based on software-level fault injections. The information collected during the proposed high-level analysis can also be used optimize the hardening phase.

As main contributions of this thesis, I address the main safety issues concerning the shared (NoC) interconnection of an MPSoC, both considering COTS and custom components. In particular, I focus on the temporal isolation of software components running on the same MPSoC. A typical safety-critical scenario is considered, where the software components have different levels of criticality (or severity of failure), i.e., mixed criticality scenario. As the main contribution, I propose some partitioning techniques to enable the usage of (NoC-based) MPSoC for the mixed criticality systems, both considering COTS and custom devices. For what it concerns the usage of COTS devices, where the main effort was spent, the proposed technique exploits the deterministic routing algorithm of the NoC. The proposed solution is suitable for an ample range of MPSoC devices as its requirements consider a fairly common MPSoC characteristics. The partitioning technique is intended to have a purely software implementation, as a module of a real-time operating system, which targets both the certification potential and reusability aspects. For what it concerns the custom NoC architectures, a set of simple solutions has been derived with the specific purpose of high certification potential.

Contents

List of Figures xi

List of Tables xiii

Nomenclature xv

1 Introduction 1

1.1 Mixed-Criticality System: NoC-based MPSoC 3

1.2 Functional Safety: Temporal Partitioning 6

1.2.1 State-of-the-art 8

1.2.2 Author’s Contribution (Functional Safety) 11

1.3 Reliability Analysis: Comparing Program Susceptibility to Soft Errors 14

1.3.1 State-of-the-art 15

1.3.2 Author’s Contribution (Reliability Analysis) 15

1.4 Outline 17

2 Processor-based Critical Systems 21

2.1 Multi-processor System-on-Chip 21

2.1.1 Network-on-Chip: Structure and Design Space 23

2.1.2 NoC Performance Parameters 32

2.1.3 MPSoC examples 34

2.2 Dependability 36
2.2.1 Attributes 36
2.2.2 Threats 37
2.2.3 Means 43
2.3 Safety-critical, Mission-critical and Non-critical Systems 44
2.4 Mixed-criticality: Requirements and Issues 45
2.4.1 Critical System Design 45
2.4.2 Mixed-criticality System: Processor Level 51
2.5 MPSoC-based Mixed-criticality System 52

3 NoC partitioning and RTOS filtering module: COTS NoC-based MCS 54
3.1 COTS NoC-based MCS: State-of-the-art 55
3.1.1 NoC Resource Sharing -based techniques 56
3.1.2 NoC Resource Permanent Reservation -based techniques . . 58
3.1.3 Gaps Inside the State-of-the-art 59
3.2 Proposed Partitioning Solution 59
3.2.1 Considered NoC Model 60
3.2.2 Off-line Partitioning: Applications Placement 61
3.2.3 On-line Monitoring: Filtering 65
3.2.4 On-line Connectivity Reduction Mitigation: Redirection . . 66
3.3 Filtering Module Implementation 67
3.4 Redirection Module Implementation 74
3.5 Experimental Evaluation 77
3.6 Summary 81

4 QoSinNoC: hardware-implemented NoC partitioning for mixed criticality 83
4.1 Custom NoC–based MCS: State-of-the-art 85

4.2 Proposed Partitioning Solution 87
4.2.1 Considered NoC Model 87
4.2.2 Application Placement Phase 90
4.2.3 Partitioning Enforcement 92
4.3 NoC Architectures and Partitioning Solution Implementation 92
4.3.1 Baseline LBDR 93
4.3.2 Extended LBDR 93
4.3.3 Routing-table Based 94
4.4 QoSinNoC 94
4.5 Experimental Evaluation 96
4.6 Summary 99

5 High-level ranking of candidate software susceptibility to soft errors 102

5.1 Radiation Effects Evaluation: State-of-the-art 103
5.2 Proposed Ranking Technique 104
5.2.1 The Overall Approach 104
5.2.2 Use Case: Sensor Data Lossless Compression 107
5.2.3 Candidate Executing Hardware Platform 112
5.2.4 Run-time Environment 112
5.2.5 Fault Injection Environment 113
5.2.6 Post-processing and Candidate Ranking Process 116
5.3 Experimental Evaluation 118
5.3.1 Validation Environment 118
5.3.2 Results Analysis 119
5.4 Summary 120

6 Conclusions and Future Work 124

6.1 Contribution 1 125
6.2 Contribution 2 127
6.3 Contribution 3 130
6.4 Future Work 132

References **135**

List of Figures

2.1 A schematic view of a 3x3 (2D) mesh topology NoC; N_0 to N_5 packet path under XY routing algorithm. 24

2.2 A detailed view of a 3x3 2D mesh topology NoC; R - router; NI - network interface; CE - connected element. 25

2.3 A schematic view of a simple NoC router architecture for 2D mesh topology: input-buffered, based on a single physical network with no virtual channels, link made of two physical channels (one per direction). 32

3.1 A schematic view of a 3x3 (2D) mesh topology NoC; N_0 to N_5 packet path under XY routing algorithm. 61

3.2 A detailed view of a 3x3 2D mesh topology NoC; R - router; NI - network interface; CE - connected element. 62

3.3 A schematic view of a NoC router architecture; a simplest input-buffered router architecture is considered, based on a single physical network with no virtual channels; link made of two physical channels, one per direction. 63

3.4 MPSoC with 4x4 mesh topology NoC, XY routing; mixed-criticality system example *mcs*_1: one C_app, one NC_app; black arrows: critical traffic paths; gray-shaded: critical regions (or domains) a) placing scenario *mcs*_1_*a*: C_app_1 - N_0 and N_{15}, NC_app_1 - all the remaining nodes; b) placing scenario *mcs*_1_*b*: C_app_1 - N_0 and N_{14}, NC_app_1 - all the remaining nodes; 70

3.5 MPSoC with 4x4 mesh topology NoC, XY routing; mixed-criticality system example *mcs*_2: two C_apps, one NC_app; black arrows: critical traffic paths; gray-shaded: critical regions (or domains) a) placing scenario *mcs*_2_*a*: C_app_1 - N_0 and N_{10}, C_app_2 - N_5 and N_{13}, NC_app_1 - all the remaining nodes; b) placing scenario *mcs*_2_*b*: C_app_1 - N_0, N_1, N_4 and N_6, C_app_2 - N_5 and N_{13}, NC_app_1 - all the remaining nodes; 73

3.6 MPSoC with 4x4 mesh topology NoC, XY routing; mixed-criticality system example *mcs*_3: three C_apps, one NC_app; black arrows: critical traffic paths; gray-shaded: critical regions (or domains) placing scenario *mcs*_3_*a*: C_app_1 - N_0 and N_{10}, C_app_2 - N_5 and N_{13}, C_app_3 - N_3 and N_{15}, NC_app_1 - all the remaining nodes; . . 80

4.1 A schematic view of a 3x3 (2D) mesh topology NoC; N_0 to N_5 packet path under XY routing algorithm. 88

4.2 MPSoC with 4x4 mesh topology NoC; mixed-criticality system example: one C_app, one NC_app; black arrows: critical traffic paths; gray-shaded: critical regions (or domains) a) placing scenario *mcs*_1_*a*: C_app_1 - N_0 and N_{15}, NC_app_1 - all the remaining nodes, rout_alg_A is used for C_app_1; b) placing scenario *mcs*_1_*b*: C_app_1 - N_0 and N_{15}, NC_app_1 - all the remaining nodes, rout_alg_B is used for C_app_1; 89

4.3 MPSoC with 4x4 mesh topology NoC; mixed-criticality system example: one C_app, one NC_app; black arrows: critical traffic paths; gray-shaded: critical regions (or domains) a) placing scenario *mcs*_2_*a*: C_app_1 - N_0 and N_{14}, NC_app_1 - all the remaining nodes, rout_alg_A is used for C_app_1; b) placing scenario *mcs*_2_*b*: C_app_1 - N_0 and N_{14}, NC_app_1 - all the remaining nodes, rout_alg_B is used for C_app_1; 97

5.1 The experimental environment. 106

List of Tables

2.1 DAL: failure condition severity, probabilities [1/flight hour], and levels 48

3.1 Placement scenario *mcs*_1_*a*: connectivity table for N_2 72

3.2 Placement scenario *mcs*_1_*a*: redirection table for N_2 76

3.3 Critical traffic latency (ns): baseline setup. 78

3.4 Critical traffic latency (ns): proposed solution implemented. 78

3.5 Reachability without redirection for *mcs*_1_*a* scenario (non-critical nodes only). 79

4.1 Architectural only figures of merit (4x4 network). 98

4.2 Figures of merit: mcs_1. 98

4.3 Figures of merit: mcs_2. 99

5.1 Compression programs characteristics. 111

5.2 Compression programs figures of merit: compression ratio and compression rate. 112

5.3 Number of actually used general purpose registers. 119

5.4 Results of fault injection into program variables, normalized per number of used registers. 120

5.5 Results of fault injection into program variables, normalized with respect to both the number of used registers and the execution time. 120

5.6 Results of fault injection into general purpose registers. 121

5.7 Results of fault injection into general purpose registers, normalized with respect to the execution time. 121

Nomenclature

Acronyms / Abbreviations

1D	one-dimensional
2D	two-dimensional
APEX	APplication EXecutive
API	application programming interface
ASIL	automotive safety integrity level
C_app	critical application
CE	connected element
COTS	commercial off-the-shelf
DAL	design assurance level
EASA	European Aviation Safety Agency
FAA	Federal Aviation Administration
FDAL	functional development assurance level
flit	flow control unit
GPR	general-purpose registers
HI	high-criticality
I/O	Input/Output

IDAL	item development assurance level
IP	intellectual property
IP core	intellectual property core
LBDR	logic-based distributed routing
LO	low-criticality
MBU	multiple-bit upset
MCS	mixed-criticality system
MMU	memory management unit
MPSoC	multiprocessor system-on-chip
NC_app	non-critical application
NI	network interface
NoC	network-on-chip
NRE	non-recurring engineering
PE	processing element
PIR	packet injection rate
QoS	quality of service
RAM	reliability, availability and maintainability
RTOS	real-time operating system
SBU	single-bit upset
SEB	single event burnout
SEE	single event effect
SEFI	single event functional interruption
SEL	single event latchup

SET	single event transient
SEU	single event upset
SIL	safety integrity level
SoC	system-on-chip
SWaP	space, weight and power
TDM	time-division multiplexing
TID	total ionizing dose
WCTT	worst-case traversal time

Chapter 1

Introduction

In the distant 1971, the first commercial microprocessor was born to create a calculator. That microprocessor was Intel 4004 and the calculator was Busicom 141-PF. Nowadays it is hard to imagine a system which does not contain a computer, from a refrigerator to an aircraft, computer systems penetrated all the areas of technology.

Among the processor-based systems, *mission-critical systems* (e.g., satellite systems) and *safety-critical systems* (e.g., power plain control, avionics) form a niche domain. These systems (from now on referred as *critical systems*) are both characterized by the fact a failure of one of such systems can lead to extremely negative consequences. For a safety-critical system such consequences can be injuries or death of people and/or severe environmental damage, for a mission-critical system – a huge economic damage.

For such systems, the nonfunctional requirements (such as cost and energy efficiency) become secondary with respect to requirements like *dependability* and in particular *safety*. The dependability can be defined in many ways, one of which is presented in [1] as:"*the trustworthiness of a computing system that allows reliance to be justifiably placed on the services it delivers.*" The safety is an important attribute of the dependability, and concerns the "*absence of catastrophic consequences*" [2].

On the other hand, each new generation of processors exhibit higher densities and lower operating voltages [3]. This decreases the dependability of such devices as several phenomena (e.g., susceptibility to hardware faults caused by electromagnetic radiation) become more and more relevant.

Unfortunately, the history knows some episodes when the hypothetical consequences of a critical system failure actually took place. Among the most known cases there is the one concerning some Toyota car models. These cars were reported to autonomously accelerate and even contemporary reduce the controllability of the car by the driver. This issue was active between 2000 to 2010 and led to at least 89 casualties [4, 5]. The exact cause is unknown, but the radiation induced bit flips (and no protection against them) are assumed to be a possible cause [6].

Driven by the incidents like the aforementioned, by the end of 2011, first edition of ISO26262 standard [7] was created. The standard, entitled *Road vehicles – Functional safety*, became indispensable for automotive companies to assure their products are as safe as possible. The ISO26262 derives from the IEC-61508 standard [8] and it is adopted for the specific requirements of automotive domain. Other safety critical domains have similar standards as for instance RTCA DO-178C [9] and the RTCA DO-254 [10] for avionics. For a safety-critical system to have a dependability certification in general — and safety certification in particular — is a must. The standards are developed, among other, to reduce the certification effort as the latter is extremely high for a safety-critical system.

Apart from facing the issue of faults (and bugs) as such, a critical system must provide a proper level of isolation between the functions it implements. This isolation is required in order to assure a faulty function will not be able to corrupt other functions of the system. I.e., given a function, made of a set of hardware and software components, it is mandatory to assure that a fault can not propagate from one of its components to a component of another function.

Given this isolation requirement, for a critical system it was historically strongly discouraged to execute multiple applications on the same processing platform, while in some domains (e.g., avionics) this was explicitly forbidden [11]. Thus, each application (or a set of tightly correlated applications) was deployed on a dedicated computer, isolated from the resources of the other, although the computers could be interconnected. This paradigm is called *federated architecture* and aims to minimize any resource sharing. The reason was the difficulty to assure, in case a set of application would share a resource, one of those application will not corrupt the data or degrade the timing properties of the other applications. For the same reason the usage of multi-core processors was absolutely disregarded for critical

systems. In fact, a multi-core processor will for sure have at least the interconnection infrastructure to be shared between the cores.

Section 2 will provide precise terminology and more insight into concepts briefly introduced so far (as well as related to those concepts introduced by the following part this introduction chapter). The following Section 1.1 will briefly introduce the context of this dissertation – MPSoC-based critical systems. Then, in Section 1.2 and Section 1.3 the contributions of this dissertation will be briefly introduced. Section 1.4 gives an outline of this dissertation.

1.1 Mixed-Criticality System: NoC-based MPSoC

The concept of *level of assurance against failure* is crucial for the design of a safety-critical system and thus for this dissertation. Considering an avionic example, the flight control system and the system in charge of in-flight entertainment, are both systems expected to work properly. However, the flight control system is expected to be developed in such a way to assure the related component will never fail during the functioning of the system. On the other hand, the failure in-flight entertainment is a function much less important with respect to the previous one.

The concept of level of assurance against failure is adopted by each critical domain or sector with slightly different peculiarities and it is referred to with different names. *Safety integrity level* (SIL) defined by IEC-61508 as generic functional safety concept, *automotive safety integrity level* (ASIL) is defined by ISO 26262 for automotive domain, *design assurance level* (DAL) is defined by RTCA DO-178C for avionic domain, etc. Considering for instance the avionic domain, if a certain function is considered of the highest DAL (i.e., DAL A), then the developer is required to certify that a failure of such function is virtually impossible. Analogously, a function is considered DAL E if its failure will have no effect. A key consequence is *a much lower effort required to develop a DAL E component*.

Industrial standards usually define four or five levels of criticality. However, in this introduction (and in the rest of the dissertation) I will consider two levels of criticality. This simplification will allow a simpler problem modeling (and solution description), without removing nothing from the latter. In detail, I refer to them as *low-criticality* (LO) task (or application) and the *high-criticality* (HI) one. I will also

consider LO to be a *non-critical application* (NC_app), while HI to be a *critical application* (C_app). Thus, in this dissertation LO and HI are synonyms of NC_app and C_app respectively.

The existence of the concept of a *component* having a given level of assurance against failure directly implies the existence of the concept of *isolation* between the components. If the independence between components cannot be demonstrated that all the components must be considered as a single macro-component. In this case the DAL assigned to this macro-component is the highest among the DALs of the components it is made of. This eventuality will result in much higher development cost, which can easily make the product economically unfeasible. In the first place, several components would require a higher DAL, as all the components should have a DAL equal to the highest among the DALs of the components the macro-component is made of. Furthermore, this macro-component will exhibit a complexity level well above the one related to the set of independent components.

From the aforementioned, it is clear how the federated architecture was an easy solution to assure the isolation between the components of the system. However, the ever-growing number of functions and their complexity made federated architecture paradigm no longer sustainable. Driven by the growing maintenance cost and other factors, industry developed several standards (e.g., [12], [7]) to allow multiple functions or applications to run on the same hardware platform. Furthermore, even the execution of software components having different DAL (or level related to the specific domain) was standardized. Thus, the *mixed-criticality systems* (MCS) paradigm was born, as components having different levels of assurance against failure was executed on the same platform.

When academic literature and the industrial standards are considered, the term *criticality* is used to indicate similar — but still different — concepts. In this dissertation *level of criticality* will be used as a synonym of level of severity of consequences of failure.

Nowadays, the mixed criticality is a common practice, for most critical domains and it is supported by the widely accepted standards. However, the executing platform of such systems have be *single-processor–based*. Considering the avionic domain, in case multi-core processors are used, the certification authority requires the designer to disable all the cores but one and to demonstrate that the disabled cores will remain disabled even in case of unexpected behavior.

On the other hand, the cutting-edge COTS processors can integrate tens of intellectual property (IP) cores on the same chip. These devices, generally referred to as *multiprocessor system-on-chip* (MPSoC), make use of a network-on-chip (NoC) interconnection which meets the scalability needs. From the aforementioned, it is evident the gap between the market products and the industry single-core–based systems.

As the number of required functionalities (and its complexity) keeps growing, it is reasonable to consider that the single-core–based system paradigm will become no longer sustainable, the same way it was for federated architecture paradigm. For instance, considering the avionics and space domains, the space, weight and power (SWaP) are extremely important from the economic point of view. The usage of multi-core would allow to move applications currently running on a set of single-core processors to much smaller number of multi-core processors. Thus, an important reduction of number of onboard hardware components will be achieved.

Both academic and industrial sectors are active in the study of issues related to the usage of multi-core systems in the context of mixed criticality. Considering avionics domain, some certifications are expected by the end of 2019 [13], while currently no certified (thus, no deployed on the field) multi-core–based MCS exists.

The main reason the multi-core–based systems are not used by most critical domains is the high complexity to certify a proper dependability level. The main issue is represented by the resources shared by the applications (or tasks) running on a multi-core processor or a MPSoC. This sharing can create data corruption and unacceptable delays. As already briefly introduced before, the isolation between the components must be demonstrated, otherwise the very concept of component disappears and the whole system can only be seen as *monolithic entity*.

For a critical system a missed deadline is (generally) considered a failure. Thus, it should be certified a sufficiently low probability of a system failure provoked, among the other, by different tasks competing for the same resource. In practice, this requires estimating, with a sufficiently high confidence, the worst-case interference between applications running on the different cores of a multi-core processor. This estimation should consider both hardware and software faults, and should be sufficiently convenient for the certification entity.

An additional factor for the complexity to achieve such estimation is the fact that, on one hand the custom design struggles to justify the non-recurring engineering

(NRE) as the number of required components is relatively small; on the other hand, the commercial off-the-shelf (COTS) components have almost no mechanism to enforce the bounded interference or at least facilitate the interference analysis (as many micro-architectural details remain confidential).

From the certification point of view, the usage of MPSoC-based systems requires a further effort with respect to the multi-core-based systems. This is due the additional complexity of NoC interconnection with respect to the classic bus interconnection used multi-core processors. In order to get closer to the creation of MPSoC-based MCS we need a set of sufficiently simple techniques to face the issues currently open [14].

The thesis focuses on safety improvement and reliability analysis techniques for a MPSoC-based MCS creation, in particular in avionics and space domains.

1.2 Functional Safety: Temporal Partitioning

The main focus is put on deriving techniques to create a COTS MPSoC-based MCS with a certification potential. As already described in the previous section, the presence of shared resources is the main reason the MPSoC technology is not used for critical systems design. A shared resource which is for sure present in any realistic usage of MPSoC is its on-chip interconnection infrastructure, i.e., NoC. The main contribution of this thesis are the techniques aiming to solve the issues related to this specific component. Alongside with COTS MPSoC, this dissertation presents some of the solutions derived considering a custom NoC architectures.

When a safety-critical system is considered, a specific set of standards must be followed during all the phases of its developments. In this thesis I will mainly refer to the IEC-61508 and RTCA DO-178C, where the former is a basic functional safety standard while the latter is the standard concerning software development of an avionic system. Functional safety of a system is defined by IEC-61508 as active or passive means to "*prevent hazardous events arising or providing mitigation to reduce the consequence of the hazardous event*".

Considering the avionics, RTCA DO-178C describes a set of techniques to prevent software failures and/or limit their effects to the system functions. The main requirement to achieve this goal is to demonstrate that the software macro-

components will execute with a sufficient *independence*. If this independence cannot be demonstrated, then all those software components will be seen as a single software component when assigning the software DAL. This, in turn will imply that the DAL assigned to this software macro-component will reflect the highest failure severity of the software components it is composed of. Thus, the cost of this macro-component development will be much higher and will undermine the economical feasibility of the system.

Under RTCA DO-178C, the following software design methods are described: *partitioning*, *dissimilarity* (of *multi-versioning*) and *safety monitoring*. The software component isolation should be implemented by the partitioning. This is one of the most important design instruments (or concepts) to safety-critical systems (not only for avionics domain). The multi-versioning consists in two or more components implementing the same function to be developed independently. The goal of this technique is to provide the redundancy (thus to improve the reliability) to the system, while avoiding common source of errors. Finally, the safety monitoring is an active mechanism to implement the protection against specific failures. This mechanism also monitors the correct functioning of the partitioning technique.

The partitioning mechanism, described by the RTCA DO-178C, must address both the extent and the scope of the interactions between the partitioned components as well as how to isolate the components from each other. The isolation must be achieved in both spatial and temporal domains. *Spatial isolation* means that one application shall not corrupt data of another application. *Temporal isolation* on other hand concerns the timing properties and means that one application shall not cause failure (i.e., deadline miss) of another application by blocking a shared resource (e.g., CPU, interconnection). To achieve spatial isolation is relatively easy as the hardware means like memory management unit (MMU) can be exploit. The situation with the temporal isolation is more complex. The safety-critical domain is niche market so the COTS MPSoC features no hardware means to address with temporal isolation issue. On the other hand, the development of custom solutions should justify the related NRE cost.

An eventual timing interference between applications is due to implicit contention for micro-architectural resources (e.g., processor caches, interconnection) and by explicit contention for resources (e.g.., processor time, peripherals, shared data structures). In this dissertation I target a specific source of implicit contention: NoC

interconnection. On the other hand, the explicit contention is out of the scope of this thesis.

1.2.1 State-of-the-art

To cope with the temporal isolation issue in MPSoC, the literature provides several solutions. Some of these solutions are adopted from bus-based multi-core related techniques, while the other have been created specifically for NoC-based MPSoC. The main issues with these solutions are certification potential, as them do not consider the industrialist's perspective. A more extended discussion on the state-of-the-art is presented in the chapters dedicated to the single contributions of this thesis.

An important distinction I make in this thesis is whether a technique can be applied for COTS components or relies on some custom hardware design. This distinction is done as this thesis contributions concerning the timing isolation can be classified in the aforementioned way. While the solutions from the COTS group can be used for the custom design solutions, the contrary is not true.

The techniques that can be applied to COTS MPSoC can be further classified according to the main idea behind them. This classification can be done according to several criterion. In the scope of this dissertation I identify two macro-categories of techniques, according on how the NoC resources (links, FIFOs, etc.) are managed to ensure timing isolation: resource sharing and permanent resource reservation. Where the techniques of *permanent resource reservation* (or *resource privatization*) macro-category refer to the opposite situation with respect to the *resource sharing* scenario. I.e., each NoC resource is dedicated to an application an it is not shared with other applications running on the MPSoC. The QoS is often mentioned by the techniques from resource sharing macro-group, as the HI tasks are guaranteed with bounded transmission latency, throughput o similar metrics.

Another minor distinction concerns whether a technique is conceived for a specific MPSoC or it can be applied for a generic MPSoC (although fulfilling a set of requisites).

A first group of solutions are those targeting robust estimation WCET, explicitly facing the shared (NoC) interconnection issue. Some of these solutions face the temporal isolation issue explicitly, while other either face that issue implicitly and

indirectly or directly ignore the issue. Those techniques which ignore the temporal isolation issue are considered out of the scope of this dissertation. The authors in [15] explicitly target the time partitioning issue by implementing a NoC usage monitoring and bandwidth enforcing mechanism. The proposed mechanism is fairly simple and does not require VCs to be implemented. However, the NoC is considered as a black box for what it concerns the WCTT, which makes the whole solution incomplete. In [16] the authors provide a technique to estimate WCTT of a NoC, which could complete the solution proposed in [15]. However, the overall solution must be carefully evaluated as it seems to induce a heavy resource under-utilization. Furthermore, this overall solution appears to be complex and have several other small issues. Another approach to estimate WCET is the one making usage of VCs, and the related priority-based preemption mechanism [17]. One of the issues related to VCs usage is the complexity of the priority-based preemptive arbitration, which will play against the certification potential. This kind of solutions are also partial as there is no isolation between the processes having the same level of priority, and there are also some other minor issues. There are also some solutions aiming to monitor the interconnection actual congestion and to identify the anomalies [18]. This technique can be used to cope with the issues of the previously described techniques. However, at the best of my knowledge, these solutions are so far only for bus-based systems. Furthermore, an integration of this technique inside the previously described ones could make to overall complexity rise above the certification feasibility threshold.

Another group of techniques is based on hierarchical scheduling. This approach is based on the usage of a scheduling entity which queues the tasks of a certain priority. This scheduling entity will execute the tasks starting from the queues having higher priority. For instance, in [19] authors propose a solution which uses nodes of the NoC as centralized arbitration units called *resource managers*, which are in charge of controlling network access. The main issue of hierarchical scheduling is a high complexity and issues to connect two distant nodes.

Apart from the techniques derived for a class of NoC architectures, there are some conceived specifically for specific MPSoC (i.e., its NoC architecture). Among the latter, several techniques based on hierarchical scheduling are conceived for Kalray MPPA® 256 [20]. These techniques [21–23] exploit the special hardware the MPSoC is featuring. The main limit of these techniques is that they rely on a complex hardware could be an issue under the certification point of view.

An alternative to the resources sharing techniques (presented so far) consists in permanent resource reservation or resource privatization. In this macro-category, the NoC is not seen as a monolithic resource, instead it is considered as a set of resources (i.e., links, FIFOs, routers, etc.) potentially independent from each other. One of these techniques [24] is developed for for Tilera Tile Processor [25]. This technique (permanently) partition the NoC into regions, each node belonging to a region can only communicate to nodes of the same region. This solution is simple, which is an advantage from the certification point of view. The issues related to this approach are limits of proximity requirements for the nodes, as well as the specific hardware like the one featured by Tilera Tile Processor.

Considering the techniques relying on custom NoC architecture design, I will use the same classification used for techniques related to COTS MPSoC.

QNoC [26] was one of the first efforts to design a NoC architecture suitable for the creation of an MCS. The proposed architecture was based on preemptive priority scheduling. The architecture makes use of wormhole packet-based round-robin scheduling which is used by the NoC architectures of many modern MPSoC. Several similar solutions, also based on the concept of priority based preemptive packet-based usage of the NoC were derived [27–33]. Each of these solutions is characterized by some peculiarities, but all make use of virtual channels, which is their main disadvantage when certification potential is considered.

One of the first solutions was also Æthereal [34] architecture implementing time-division multiplexing (TDM)-based circuit switching approach, where wires and buffers remain reserved for certain points in time. The resources are reserved for critical applications, while the non-critical applications use leftover bandwidth from the critical applications. This solution however presents several problems like the difficulty to support communication between distant nodes and presents a not negligible complexity as well.

The authors in [35] present a NI modification to allow TT-based scheduling of the NoC. This solution grants a bounded latency on high-criticality traffic by using a contention-free channel. However, to support the proposed approach the overall system should present a high level of complexity.

A different approach to the problem is *link division multiplexing technique* [36]. In this solution, each physical link is partitioned to simultaneously transmit serialized packets belonging to distinct traffic flows. The idea of this solution is appealing;

however, it presents a high complexity from both hardware and conceptual point of view.

Apart from the (NoC) resource sharing technique presented so far, there are some solutions based on avoiding resource sharing. In this macro-category, the NoC is not seen as a monolithic resource, instead it is considered as a set of resources (i.e., links, FIFOs, etc.) potentially independent from each other.

The authors in [37] use hardware-enforced segregation between a safety-critical domain and a non-safety-critical domain. The main limit of this solution, as it was for [24], is the high cost required to connect two distant nodes. The authors in [38] present a NoCDepend method to allow communication between critical regions, implemented by a set of nodes used as input and output gateways. Thus, this feature can solve the issue of connecting two (or more) distant nodes. However, this solution presents a certain complexity which can be a prohibitive for the certification cost. In addition, the authors in [38] present a dynamic reconfiguration technique. This can be used as an alternative to the inter critical region communication, however this solution presents the same disadvantages of circuit-switching–based solutions.

1.2.2 Author's Contribution (Functional Safety)

As the main contributions of this thesis, I propose a technique to solve temporal isolation issues related to implicit contention of NoC interconnection. Both the COTS- and custom NoC-based MPSoC were considered, mainly targeting the avionic field.

All the proposed techniques provide the temporal isolation to critical applications and all are based on traffic flows isolation. For the COTS NoC–based MPSoC this isolation is implemented purely in software, targeting several architectures. For the custom-NoC–based MPSoC the temporal isolation is implemented purely by hardware means.

From the QoS point of view the proposed techniques offer two levels of service: guaranteed service (GS) and best effort (BE). An application running assigned with GS level is guaranteed to have bounded latency and throughput. On the other hand, applications running under BE level have no guarantees. The proposed techniques support an unbounded number of criticality levels, as each application running under the GS level is granted with the temporal isolation.

The rest of this section will first describe the contribution related to COTS-based MPSoC devices (seesectionsec:intro-contrib-NOC-COTS), then the custom NoC-based MPSoC will be considered in seesectionsec:intro-contrib-qosinnoc. Finally, in seesectionsec:intro-summary-contrib-functional an itemized summary of this contributions will be provided.

RTOS filtering module: COTS NoC-based MCS

Considering the COTS MPSoC, a technique requiring few information about the implementation details of NoC infrastructure was derived [39]. The proposed technique has been developed as a software module to be interred in a certified real-time operating system (RTOS). Once the RTOS is upgraded with the proposed module, the whole RTOS must undergo the certification process, as any modification invalidates the existing certification. This certification will be required only once, and the proposed solution can be reused for the future projects without requiring a new dedicated certification. It should worth to notice that generally (and especially for avionics), a new system must be certified as a whole. However, if the proposed solution will be already certified, then the whole certification process is easier.

This solution is based on traffic flows isolation and can be applied to a set of architectures. From the quality of service (QoS) point of view, it features two levels of service: guaranteed service (GS) and best effort (BE). However, the proposed solution allows an unbounded number of applications and criticality levels to be implemented. Each application, which is set to have GS level of service, is guaranteed to have bounded latency and throughput. I.e., the proposed approach guarantees that no interference from the other applications does exists. I analyze the limits of the proposed solution especially for it concerns the connectivity issue and draw some rules to allow an efficient usage of the proposed methodology.

QoSinNoC: framework to custom NoC comparison for mixed criticality

Concerning the custom NoC design, QoSinNoC framework [40] was developed to analyze a set of NoC architectures and techniques to allow their usage in the context of mixed criticality. The goal of the framework is to provide to the designer a tool support him or her in the chose of the NoC architecture. The architectures currently supported by the framework are:

- a basic version of logic-based distributed routing (LBDR) [41] with no direct support for mixed criticality;
- a modification of LBDR to better support the mixed criticality;
- a basic version of logic-based distributed routing, to also consider an architecture able to support a non-minimal routing.

As low complexity is fundamental to have a certification potential, all the considered NoC architecture are intentionally chosen to be as simple as possible. Thus, the considered architectures do not feature virtual channels and are based on a single physical network.

The framework supports a technique, for each of the considered architectures, to allow their usage in the context of the mixed criticality. As the placement of the applications is fundamental when reservation-based techniques are used, the framework helps the designer to collect precious information during design space exploration. In particular, the framework configures the chosen architecture according to the placing and communication constrains of GS applications, eventually signaling the impossibility to meet such constrains. If the configuration is possible, it will be possible to collect some useful information (i.e., throughput and logical connectivity between nodes) about the leftover resources for BE applications.

Summary: Functional Safety Contributions

The main ideas and aspects of the contributions described in Section 1.2.2 can be represented by the following itemized list:

- COTS NoC-based MPSoC:
  - software components implicit NoC contention: temporal isolation;
  - purely software solution
  - main idea:
    - critical software component: temporal isolation (guaranteed service - GS);
    - non-critical software component: no guarantees (best effort - BE);
    - traffic isolation established during application mapping phase;
    - traffic isolation monitored by a dedicated RTOS module;

  - main requirement: deterministic and known routing being used by the NoC;
  - avionic field;
  - certification-friendly solution: as simple as possible;

- custom NoC-based MPSoC:
  - software components implicit NoC contention: temporal isolation;
  - set of NoC architectures and techniques to allow their usage in the context of mixed criticality
  - main idea:
    * critical software component: temporal isolation (guaranteed service - GS);
    * non-critical software component: no guarantees (best effort - BE);
    * hardware-implemented traffic isolation established at application mapping time;
    * framework to perform a comparison of different NoC architectures;
  - avionic field;
  - certification-friendly solution: as simple as possible.

1.3 Reliability Analysis: Comparing Program Susceptibility to Soft Errors

A further contribution of this thesis focuses on the reliability of the MPSoC related to core-level. In detail, I present a technique to assess the software reliability at early stages of the system design cycle. The technique considers the issue of radiation induced transient hardware faults also known as soft errors. The technique is based on fault injection and only considers the software data structures (i.e., variables). The technique can be used when the final computing platform is not known yet, in fact only a limited knowledge of the latter is required.

Considering the MPSoC at core level, this thesis presents a technique to assess the software reliability at early stages of the system design cycle. In detail, this dissertation addressed the robustness against the radiation induced transient hardware

faults also known as soft errors. The technique is based on fault injection and only considers the software data structures (i.e., variables). The technique can be used when the final computing platform is not known yet, in fact only a limited knowledge of the latter is required. Considering a candidate for final computing platform, such information can be gathered from its compiler and an execution time estimator with no need to actually execute the software on that computing platform.

1.3.1 State-of-the-art

Evaluating the dependability characteristics of a compression program when no information about the target execution platform is available yet is a challenging task. Considering the early phases of the design process, only a high-level version of the algorithms (e.g., in C or C++) to be considered available, while the target executing platform has still to be decided.

At this stage, a possible approach for performing reliability analysis that could provide useful information about the robustness of the considered compression algorithms is based on executing them using a meaningful workload, injecting faults modeling the radiation effects and observing the resulting behavior.

From a technical point of view, such a kind of analysis typically adopts simulation-based fault injection [42], where bit flips are injected inside the data structures of the program [43–49]. In some cases, other representations of the compression program may also exist (e.g., a Matlab's Simulink model): in these cases, a preliminary analysis can be performed by executing fault injection campaigns on this model [50]. The formal technique [51, 52] can be used as an alternative to the fault injection simulation.

The main issue concerning the aforementioned is the lack of precision. Having no information about the hardware, the injected faults could never precisely model the radiation effects [53].

1.3.2 Author's Contribution (Reliability Analysis)

As a further contribution of this thesis, I propose a solution to address the reliability issue of COTS MPSoC used for space and avionics systems. The proposed technique focusses on the node level of the MPSoC and considers the reliability of the software

components against radiation induced soft errors. The purpose of the proposed technique is to help the designer in the choice of most suitable (in terms of reliability) software component for a given system function, when more candidates for that function exist. The proposed technique is intended to be used in the early stages of design phase, when the target executing platform has not been chosen yet.

The rest of this section will first describe the contribution related to High-level technique for ranking candidate software susceptibility to soft errors (seesectionsec:intro-focus-FI-contrib-ranking), then seesectionsec:intro-summary-contrib-reliability will provide a itemized summary of this contribution.

High-level technique for ranking candidate software susceptibility to soft errors

The proposed technique allows to perform a ranking of a set of different software implementations of a given functionality. A typical situation, the proposed technique is intended to be used for, is when different algorithms or even paradigms exist to implement the required function (e.g., data compression function). Thus, the proposed solution is intended to help the designer in the choice of the most suitable (in terms of reliability) software implementation for a given system function, when more candidates for that function exist. The overall solution also provides some information to be used during an eventual hardening phase, as an extra feature (naturally implemented by the ranking process).

The proposed solution is aware that the main drawback of any high-level approach is the limited capability to capture the exact behavior of the hardware affected by radiation-induced soft errors [53]. Indeed, the aim of the proposed technique is to perform a ranking of a set of software components, while any usage to compute the quantitative reliability figures is discouraged. Thus, this technique benefits from the fact that a comparison requires a lower level of precision with respect to the quantitative reliability figures computation. However, main challenge of this technique is to achieve a sufficient precision to allow a meaningful comparison.

The proposed approach is evaluated by ranking a set of lossless compression programs, candidate for the data logging sub-system. To validate the proposed approach, the results of high-level fault injection have been compared with those gathered with register-level fault injection simulation. In detail, general-purpose

registers (GPRs) of a candidate executing platform have been considered. The injection of faults inside the memory has not been considered, as memories (both on chip and external) used in space applications, are typically protected by error detection/correction capabilities [54, 55], and can be considered as immune to radiation-induced soft errors. Despite the GPR-level injections do not provide precise reliability figures, I explain why it is reasonable to assume that their precision allows the comparison among candidate programs of a particular type (e.g., compression programs). In detail I analyze how the fault not covered by GPR-level fault injection will contribute mainly as common mode error.

Summary: Reliability Analysis Contributions

The main ideas and aspects of the contribution described in Section 1.3.2 can be represented by the following itemized list:

- reliability analysis:
    - assessing software component robustness against radiation induced soft errors;
    - ranking a set of software components implementing the same function;
    - early stage design;
    - almost hardware independent;
    - program-variables-level fault injection simulation;
    - main idea:
        * hardware-specific effects obtained from low-level fault injection are mostly common-mode.

1.4 Outline

This section gives an outline of this dissertation.

- Chapter 2 provides more background on the context of this dissertation. The issues related to the (NoC-based) MPSoC usage in the critical domain, already introduced in the current chapter, will be comprehensively covered. The

chapter will provide precise terminology and will define basic terms and metrics. The focus will be put mainly on the safety and reliability aspects.

- Chapter 3 describes the first contribution of this dissertation: a technique to solve one of functional safety issues related COTS (NoC-based) MPSoC usage in the context of mixed criticality. In detail, the technique addresses the temporal isolation issue related to implicit contention of the NoC interconnection. The approach is to reserve NoC resources and avoid their sharing. This is done by virtually partition the system and to enforce this partitioning by filtering faulty traffic (i.e., by blocking NoC usage that would break the partitioning). The traffic partitioning exploits the deterministic routing used by the NoC. Thus, the main requisite for the proposed solution is that the routing algorithm (used by the NoC) is deterministic and known (which is fairly common). The traffic filtering is implemented as purely software module to be inserted inside the RTOS. The overall solution is deliberately simple to exhibit a high certification potential. Furthermore, a solution to improve the overhead of the reservation approach is proposed.

  The chapter will also further extend the analysis of the-stat-of-the-art landscape, allowing a better locate the proposed solution inside the latter.

- Chapter 4 describes the second contribution of this dissertation: a technique to solve one of functional safety issues related (NoC-based) custom MPSoC usage in the context of mixed criticality. In detail, the technique addresses the temporal isolation issue related to implicit contention of the NoC interconnection. The approach relies on privatization of NoC resources, thus sharing avoidance. QoSinNoC framework was developed to analyze a set of NoC architectures and techniques to allow their usage in the context of mixed criticality. The goal of the framework is to provide to the designer a tool to support him or her in the chose of the NoC architecture. The architectures currently supported by the framework are:

  - a basic version of LBDR with no direct support for mixed criticality;
  - a modification of LBDR to better support the mixed criticality;
  - a basic version of routing table–based routing, to also consider an architecture able to support a non-minimal routing.

As low complexity is fundamental for certification purposes, all the considered NoC architecture are intentionally chosen to be as simple as possible. Thus, the considered architectures do not feature virtual channels and are based on a single physical network.

The framework supports a technique, for each of the considered architectures, to allow their usage in the context of the mixed criticality. As the placement of the applications is fundamental when reservation-based techniques are used, the framework helps the designer to collect precious information during design space exploration. In particular, the framework configures the chosen architecture according to the placing and communication constrains of GS applications, eventually signaling the impossibility to meet such constrains. If the configuration is possible, it will be possible to collect some useful information (i.e., throughput and logical connectivity between nodes) about the leftover resources for BE applications.

The chapter will also further extend the analysis of the-stat-of-the-art landscape, allowing a better locate the proposed solution inside the latter.

- Chapter 5 describes the third contribution of this dissertation: a solution to address the reliability issue of COTS MPSoC used for space and avionics systems. The proposed technique focusses on the node level of the MPSoC and considers the reliability of the software components against radiation induced soft errors. The purpose of the proposed technique is to help the designer in the choice of most suitable (in terms of reliability) software component for a given system function, when more candidates for that function exist. The proposed technique is intended to be used in the early stages of design phase, when the target executing platform has not been chosen yet. The proposed technique performs a reliability comparison between a set of software components candidates, when more options exist for a given system function. This comparison is done at high level, considering very few information about the computing hardware platform and concerns the robustness against radiation-induced soft errors, especially single event upset errors. In detail, the proposed solution is based on the execution of the software component and injecting single bit-flip faults in the program variables during its execution. Finally, observing the effects of the injected faults on the program execution and on its results. In this way, we can perform a first level analysis of the fault

masking and self-convergence capabilities of the code, as well as get some information about its intrinsic capabilities to detect errors.

The chapter will also further extend the analysis of the-stat-of-the-art landscape, allowing a better locate the proposed solution inside the latter.

- Chapter 6 concludes my dissertation. The contributions are summarized as well as the related current limitations. Finally, some suggestion for future research are presented.

Chapter 2

Processor-based Critical Systems

A critical system, as the name suggests, is system which proper functioning is strongly required. This chapter will define the basic terminology on the critical systems and the dependability of a system. This chapter will provide background on context of this dissertation, i.e., Multi-processor System-on-Chip (MPSoC) devices as well as the requirements and issues of a mixed-criticality system (MCS).

This chapter will first describe (Section 2.1) what is a Multi-processor System-on-Chip (MPSoC).

Then, in Section 2.2 it will be described the concept of dependability. In particular, it will be described how the dependability of a system can be defined and measured (i.e., dependability attributes), which are the threats to its dependability, and which are the means to cope with these threats.

Once the basic dependability concepts are defined, a critical system will be defined as well as the related requirements and issues will be analyzed (taking into account the avionics domain), respectively, in Section 2.3 and Section 2.4.

Finally, Section 2.5 will specifically target the multi-core processors in general and the MPSoC devices in particular.

2.1 Multi-processor System-on-Chip

The first integrated circuit (IC) was produced in 1969 and contains 1,200 transistors. These devices represented the main step of the third industrial revolution, known as

digital revolution, and marked the beginning of the Information (or Computer) Age. Since then, these devices followed the Moore's law in an ever-improving size, speed, and capacity.

However, in the early 2000s, general-purpose processors manufacturers were observing the single-core paradigm to hit several *technological walls*. To overcome these walls the multi-processor design paradigm was adopted [56]. In detail, these are power wall, memory wall, instruction-level parallelism wall and complexity wall. The *power wall* was hit by the single-core processor design around 2004. The trend of scaling down the gate size, reducing supply voltage and increasing the clock frequency became no longer feasible, mainly due to the heat dissipation issue [57]. The *memory wall* is provoked by the much slower memory performance growth, when compared to the performance growth [58]. The sophisticated multi-level cache hierarchy architectures became no longer sufficient to face the memory latency bottleneck. The *instruction-level parallelism wall* is the impossibility (or unfeasibility) to improve instruction-level parallelism extraction for singe-core paradigm [59]. The *complexity wall* is the issue related to the ever-growing architectural complexity to deal with the previously listed issues. This issue has a huge impact as the verification costs grows enormously with complexity growth. The multi-core paradigm overcomes the power wall by replacing the single fast-but-power-hungry core with a set of slower cores. These slower cores will cooperate providing much more computational power and also solving the heat generation issue (as the power consumption is linearly proportional to the clock frequency). For what it concerns the power wall, multi-processor paradigm faces this issue in two ways. First, as the frequency is lower than for single-core approach, the performance-latency gap becomes less evident. Second, the latency gap is compensated by the overall bandwidth of the system, which scales much better [60]. The instruction-level parallelism wall issue is solved by considering the task-level parallelism instead. This allows to move at parallel programming level, rather than complex hardware to (transparently) extract instruction-level parallelism from the instruction stream. Finally, multi-core approach overcomes the complexity wall by considering much simpler cores. Furthermore, a core must be designed once and replicated multiple times inside a chip, which further cuts the verification effort.

Alongside with the growing number of processing cores integrated on the same chip, also the on-chip peripherals number increases. This creates a system which is often referred to as system-on-chip (SoC). As the number of on-chip components

connected to the same bus-based interconnection was ever-growing, this interconnection infrastructure quickly became no longer suitable to sustain the meet intensive parallel communication requirements. To solve this communication bottleneck an off-chip networking approach was investigated. This led to the implementation of an on-chip switching network, called network-on-chip (NoC), to interconnect the IP modules in SoCs.

In the scope of this dissertation a *multi-processor SoC* (*MPSoC*) is referred to a SoC featured with NoC interconnection infrastructure. This term will be referred to a generic design which could be either homogeneous or heterogeneous for what it concerns its computing cores.

This section will fill first provide the NoC background both on its structure and its metrics. Finally, some examples of state-of-the-art MPSoC will be briefly described.

2.1.1 Network-on-Chip: Structure and Design Space

As an MPSoC is characterized by a high number of on chip IPs, a salable interconnection is a must. NoC interconnection has been created to answer this scalability requirement, as the main idea behind the NoC is the adaptation of the network concept to the context of on-chip interconnection. Furthermore, a NoC allows a much larger number of solutions for what it concerns, not only routing and arbitration strategies, but also the organizations of the communication infrastructure as such. This provides to the designer a large design space and thus allows him/her to adopt a solution that better fit the project peculiarities.

The first classification of a NoC is made according to its topology. The ring and mesh topologies are the simplest ones, but also different types of torus are used (also for commercial products). A schematic view of a NoC with a two-dimensional (2D) mesh topology can be seen in Fig. 2.1. The figure also shows the path taken by a packet send by N_0 towards N_5, assuming XY routing algorithm is used by the NoC. A detailed view of the same NoC can be seen in Fig. 2.2. It can be seen how nodes connected through the links forms a network. Each node of such network is formed by a *connected element* (CE) connected to a *router* through the dedicated *network interface* (NI). The nodes are connected to each other through the two mono-directional *communication links* (or *channels*). Thus, a NoC can be seen as composed of tree building blocks: link, router and NI. The structure of

the NoC is more complex than the one of the bus. This extra complexity does not only meet the scalability requirement but also allows to implement several different network communication protocols to better fit a specific set of requirements of a given application. A CE is not a part of the NoC; however, it will be briefly described as the only goal of the NoC interconnection is to implement the communication among CEs.

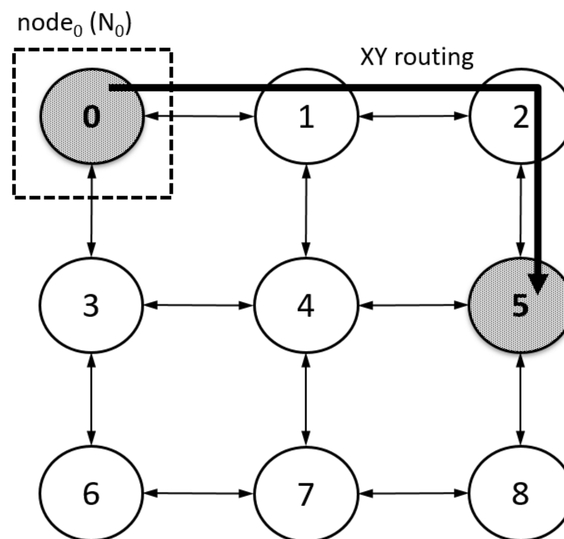

Fig. 2.1 A schematic view of a 3x3 (2D) mesh topology NoC; N_0 to N_5 packet path under XY routing algorithm.

Link

A *communication link* is made of a set of physical wires and connects two routers in the network as well as a router and a NI. Each link can be composed of one or more physical or logical *channels*, which are groups of wires connection two entities. Typically, a NoC link has two physical channels making a full-duplex connection between the routers, i.e., there are two uni-direction channels in opposite directions. The number of wires per channel, called channel bitwidth, is uniform throughout the network. A synchronization protocol between the N_{source} and $N_{destination}$ must be considered as part of link implementation. Some implementation based on asynchronous links were also proposed to implement globally asynchronous locally synchronous (GALS) systems. However, normally synchronous links are used, implemented

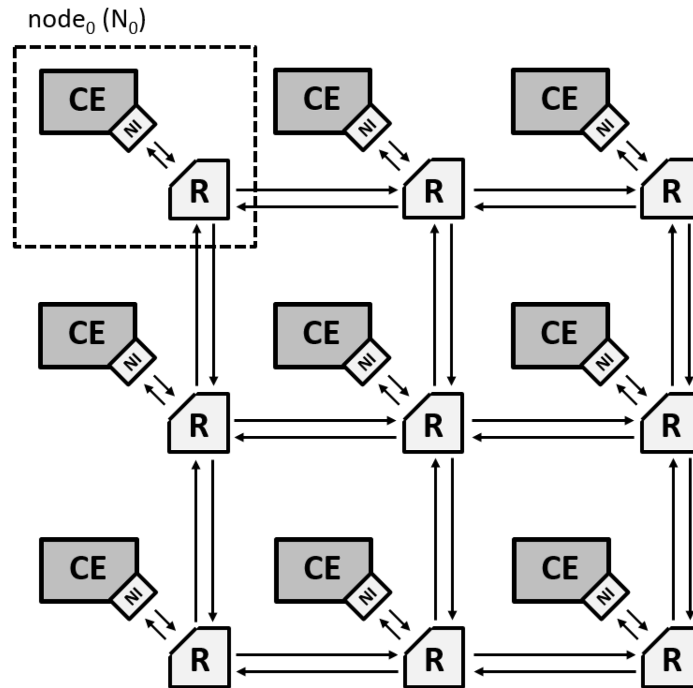

Fig. 2.2 A detailed view of a 3x3 2D mesh topology NoC; R - router; NI - network interface; CE - connected element.

either by some dedicated wires to support the synchronization mechanism or through other approaches such as FIFOs. It comes natural for designers to implement fast, low-power and reliable wiring between the nodes of the NoC.

From the information point of view, a *packet* is composed of a set of *flow control units* or *flits*. A flit represents an atomic unit of packet. On the other hand, the minimum amount of data that is transmitted in one link transaction is called *phit*. However, usually a flit is exactly equal to a phit, i.e., (once a transmission is initiated) a flit is transmitted at each NoC clock cycle.

Connected Element

In this dissertation I will call *connected element* (CE) whatever IP core connected (through the NI) to a router of the NoC. Thus, in the considered NoC model, a CE can be either a passive slave (e.g., memory) or an active element (e.g., processor). Considering the MPSoC-level scope, each CE sees other CEs as global resources connected through the NoC interconnection. Moreover, a CE can also have local

resources connected to the local bus. Each node of the NoC (i.e., the related CE) is considered as a remote resource and mapped to a memory map as any local resources (if any). Local resources are accessed as usual through the local bus, whereas remote resources are mapped to the driver in charge of instruct the NI in order to reach the global resources. Thus, the idea is to make NoC transparent to the CE.

Network Interface

The NI (also called as network adapter) is a peripheral on the local bus of the CE it is connected to. It receives requests from the local CE according to the protocol defined on the local bus. The CE can then wait for a response (bus-transaction communication model) or it can continue its computations, pending an interrupt that signals that requested data is available at a predefined location on a local memory (DMA communication model).

The NI performs the following tasks (although some of these tasks can be partially performed by a software module acting as the NI peripheral driver):

1. Receives requests from the local CE to remote resources, transforms them in NoC packets, and injects packets inside the NoC;

2. Receives responses from remote resources and sends them to the local CE according to the BT model or to the DMA model;

3. Receives requests from remote nodes and forwards them on the local bus to the local CE. The local CE should respond to such requests according to the implemented model of communication (i.e., bus-transaction or DMA).

Each NI has a look-up table to translate each global resource's address from the local memory map to a set of coordinates that identify the position of that resource on the NoC. The CE in the considered NoC model is either a passive slave or a processor. A NI can be divided into a front end and a back-end part. The front end interacts with CE and it is usually implemented as a socket (e.g., AXI [61]). This part purpose is to make NoC transparent to the CEs it is connected to. The back-end part is connected to the router and actually manages the communication protocol. The detailed description of NI is out of the scope of this dissertation.

Router

A router can be seen as a component, having several input ports and output ports, and which is in charge of connecting the input ports to the output ports. Apart from defining the routing algorithm, this component is also in charge of packet collision management and other tasks. The *flow control policies* (routing, arbiter, etc.) defines the overall strategy for moving packets through the NoC. Thus, not only router-level policies are involved, but also the NoC-level issues are addressed, for instance deadlock-free routing. *Deadlock* occurs when network resources are fully occupied and waiting for each other to be released to proceed with the communication, that is, when two paths are blocked in a cyclic fashion [62]. *Livelock* occurs when the status of the resources keeps changing (there is no deadlock) but the communication is never completed.

The flow control policies can also address the optimization of the NoC resources usage, and the communication guarantees insurance. The concept of *quality-of-service* (QoS), adopted from its original meaning in networking, describes different levels of quality. Each of these levels is characterized by different guarantees on the communication performance. QoS will be detailed further on in Section 2.1.2. The flow control is typically implemented following the distributed control paradigm, and this dissertation will only consider this implementation. For distributed control, each router makes decisions locally.

The concept of *virtual channels* (VCs) is strictly related to the concept of QoS. VCs implement the concept of multiplexing a single physical channel over several logically separate channels with individual and independent buffer queues. Each of this queue buffer has a different level of priority so same does the related logic channel. The main goal of a VC implementation is to improve performance by introducing the concept of priority, and thus the concept of QoS.

The overall strategy of how the data, once injected into NoC by N_{source}, moves through the NoC to reach $N_{destination}$ is defined by several aspects:

- The ***routing algorithm*** selects the output port to forward the packet to. This decision is usually based on the information contained inside the packet header (generally contained inside the first flit). There are several possible routing algorithms that can be used in a NoC, representing different trade-offs between

cost and performance. There are several possible ways to classify the routing algorithms, here on some of these classifications will be listed.

A routing algorithm can be classified according to whether the path is computed by each router based on the destination information or whether the entire path is already pre-computed by the sender. The *source routing* paradigm refers to the latter scenario, while the former is called *destination-based routing*. For the source routing all the information about the path is contained inside the packet, so the router logic requirements are minimal. The opposite situation is with destination-based routing, as the routers must be able to compute the path of a packet based only its $N_{destination}$ information (as this is the only information contained in the packet). On the other hand, source routing will require more bandwidth with respect to the destination-based paradigm, due to the extra information describing the path. Furthermore, the amount of this information (and thus the bandwidth overhead) is proportional to the path length. Generally, it is considered that to preserve the bandwidth is more valuable than a lower router complexity. This consideration, combined with the scalability issues of source routing, make the destination-based routing the most common type of routing [63].

Yet another important classification regards whether, given two specific nodes, the path of the packet is always the same or can change (e.g., to avoid a broken node). The former approach is called *deterministic routing* paradigm, while the latter is called *adaptive routing*. A common deterministic routing is *XY routing*. In XY routing, the packet moves through the same row it is injected inside (i.e., X axis), once it reaches the column $N_{destination}$ belongs to, the packet moves through that column (i.e., Y axis) to the $N_{destination}$ [64]. In the adaptive routing, alternative paths between two nodes may exist. Two examples of adaptive routing algorithm are *negative first* and *west first* algorithms [65].

A concept similar to deterministic-adaptive paradigm is the static-dynamic routing paradigm. In the *static routing* paradigm, paths between cores are defined at system design time an does not change over time. The *dynamic routing*, on other hand, generally refers to the possibility of a run-time change of the routing strategy (e.g., to implement load balancing strategy).

Another classification is based on either there is a single target for each packet or there are multiple targets. An *unicast routing* indicates that a packet has a

single target whereas in the *multicast* routing a packet can be sent to multiple NoC nodes simultaneously. Very similar to the former are the broadcast and narrow-cast communication paradigms. In the *broadcast communication*, a packet targets all nodes of the NoC. Finally, a *narrow-cast communication* is a communication initiated by a master to target a set of slaves.

A routing algorithm can also be classified according to the length of the path followed by a packet. A *minimal routing* is a routing which always guarantees the shortest path. On the other hand, *non-minimal routing* algorithms allow a packet to take paths which are not the shortest. In literature it does exist several more or less curious routing algorithms, for instance *deflective routing*. In this routing scheme a packet that cannot be accepted by the target node, is deflected into the network (to return later). An example of deflective routing scheme is *potato routing*, in which each packet follows the lowest-delay path. This approach, although potentially solving some congestion issues, present a high complexity and can potentially lead to bigger issues than the solved ones.

Finally, a routing algorithm should consider deadlocks and livelocks and be deadlock-free and livelock-free [65, 66].

- While the routing algorithm is in charge of selecting an output port for a packet, the ***arbitration policy*** decides which input port to select when multiple packets are simultaneously requesting the same output port. One of the most important classification of arbitration approach is whether a *delay communication model* or a *loss communication model* is used. The former model allows the packets (flits) to be delayed (stalled) but does not allow a packet to be destroyed (dropped). In the loss model a packet can also be destroyed (e.g., to implement preemption) [64]. However, the loss communication model is not trivial to implement. A packet can be distributed through the network (e.g., using wormhole routing) and its destruction will be neither immediate nor free. Furthermore, a re-transmission logic must be implemented.

  As any of the policies, also the one implemented by the arbitration logic can be either *static* (*fixed*) or *dynamic* (*variable*).

  From the strictly implementation point of view, an arbiter can be either distributed or centralized. In the *centralized* approach there is a unique logic, multiple request lines are used by different input ports to request the usage of the required output port. The centralized arbiter can guarantee fairness,

and usually round robin arbitration scheme is used. In *distributed* approach is usually more complex than centralized one. This implementation approach associates a part the overall logic to each input port. The decision of such sub-units is that compared based on the priority or other factors. A centralized implementation usually targets the fairness and the simplicity, whereas distributed arbitration targets the latency and the priority-based schemes.

- The ***switching policy*** defines how the data is transmitted from the source node to the destination one. The main distinction is whether between the store-and-forward switching and packet switching approaches. In the *circuit switching* (also known as *store-and-forward switching*) approach the whole path (including routers and channels) from source to node is previously established (by the header) and reserved for the transmission of the whole packet. The advantage of such approach is to increase throughput and also requires less memory elements (i.e., buffers) with respect to the other approaches. The cost, on the other hand, is the increase of latency. This is in part due to the time needed to establish a connection. But mostly there is an overall latency issue as a part of circuit is privatized for the time of transmission, which makes many other communications impossible for all the time, starting from circuit reservation procedure start till the end of transmission. In the *packet switching* approach on the other hand, the first flit of a packet (i.e., header flit) progressively establishes the connection as it moves through the NoC. There are several different buffering and forward strategies to implement this approach.

  In the *store-and-forward* strategy, a node receives (and stores) the complete packet before it starts to forward this packet to the next node in the path. This strategy requires that the buffer size is big enough to store the whole packet. The advantage of this strategy is not to occupy links in case of a stall condition.

  The *wormhole strategy*, on the other hand, the data (flit) is forwarded to next in the path node as soon as the latter can receive it (not necessarily the whole packet). This approach reduces the latency but, in case of stalling, many links can remain blocked.

  The *virtual-cut-through* mechanism can be seen as an improvement of store-and-forward mechanism, which creates a solution similar to the approach used in wormhole mechanism. As in store-and-forward approach, each router has buffers capable to store the whole packet. The data transmission is similar

to the wormhole approach but, flow control allocates buffers and channel bandwidth on a packet level. This means that a sender node waits for a confirmation that the whole packet can be accepted by the next node in the path, before forwarding any data (to that next node).

The main advantage of virtual-cut-through strategy over wormhole strategy is that the former does not occupy any links in case of stalling. This advantage comes with a cast of bigger buffers needed, as the whole packet must be stored in cased of stalling.

- The ***buffering policy*** is the decision on number, location and size of buffers used to store packets data. In this dissertation I will use FIFO as a synonym of buffer, as it is most common buffer implementation and basically the only which is relevant in scope of this thesis. As already seen for switching policy, the size of a buffer can be either big enough to store an entire packet of data or it can be smaller. Additionally, it must be decided whether to have a single shared (by all the ports) buffer or to have a dedicated buffer per each port, or even have a dedicated buffer both for both input and output direction (two buffers per port).

  As a memory element size is much bigger with respect to the size of a logic element, the buffering policy has a huge impact on performance, router size and its power consumption.

  A distributed approach based on a single buffer shared among all the router ports will provide a huge chip area and power consumption advantage with respect to the design based on per-port buffer. On the other hand, the distributed approach and/or an unduly small FIFOs can, under certain condition, provoke a high congestion levels and make the NoC very inefficient or even completely blocked.

There are several very different NoC architectures, both from academic and industrial side. However, it can be observed the presence of some common design trends among available NoCs [67, 68]. The most common implementation is a 2D mesh topology. The wormhole strategy is mostly used as it allows an efficient usage of NoC resources. Considering the mesh topology XY routing is very common as it is both simple and efficient. Concerning the buffering, usually input buffering is the only implemented as this solution is a good trade-off between cost and performance. Fig. 2.3 depicts a typical router architecture (used in 2D mesh NoCs).

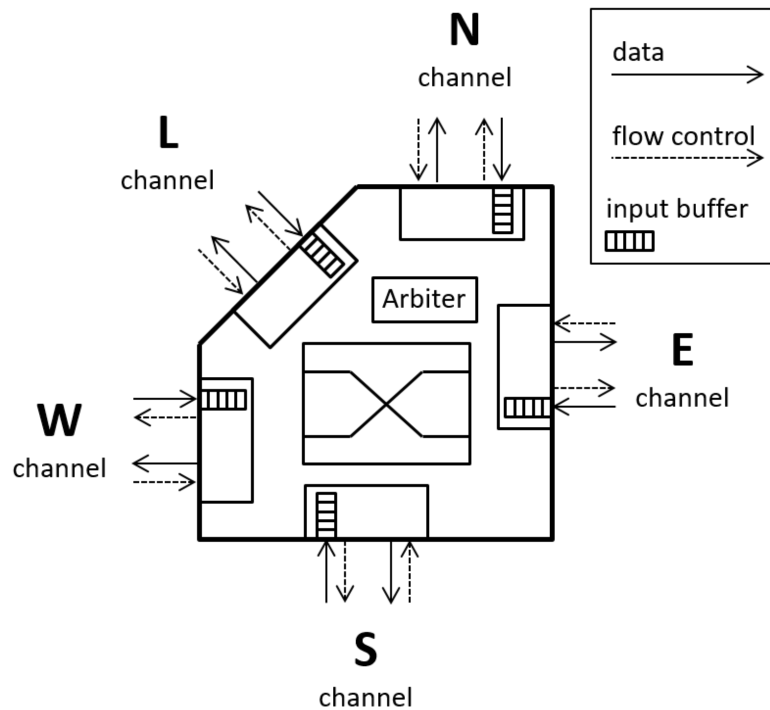

Fig. 2.3 A schematic view of a simple NoC router architecture for 2D mesh topology: input-buffered, based on a single physical network with no virtual channels, link made of two physical channels (one per direction).

2.1.2 NoC Performance Parameters

There are three main parameters to evaluate the performance of a NoC: bandwidth, latency and throughput.

- The ***bandwidth*** is the maximum speed at which data moves through the NoC. The unit of measure for bandwidth is bit per second (bps) and usually the bits of the whole packet (including heading and tail) are considered.

- ***Latency*** is the time elapsed from the moment the first data of the packet has been injected inside the NoC to the moment the packet has been completely received by $N_{destination}$. The unit of measure for latency is time, and often it is expressed in terms of (NoC) clock cycles. Normally the NoC is considered as a whole and the average latency is inferred as well as its standard deviation [69]. However, in the context of critical systems, the maximum latency of single

packet is the most important metric. The maximum latency is also called as worst-case traversal time (WCTT)

- ***Throughput*** is defined as the maximum traffic accepted by the network, i.e., the maximum amount of information delivered per time unit [69]. The unit of measure for throughput is packets per time (i.e., per second or per clock cycle). Normally the throughput is normalized by the size of the messages and by the size of the network. One can have a normalized throughput (independently from the size of the messages and of the network) by dividing it by the size of the messages and by the size of the network. In this way, normalized throughput will refer to the number of bits each node on average is sending per a unit of time (i.e., per second of per clock cycle).

QoS for on-chip networks is an adaptation of the off-chip QoS. It is used for the packet-switched telecommunication and does not measure the service quality, instead it describes the traffic prioritization policies and resource reservation control mechanisms. A possible definition of QoS is *as service quantification that is provided by the network to the demanding core* [64]. The previous definition needs a definition of a required service and a quantification measure. Usually, a required service is defined as a set of figures of the performance metrics of the network (e.g., latency, throughput) QoS levels can be grouped in two macro-groups [70]:

- *best-effort* (BE) QoS level: as the name suggests, this level does not provide any guarantee on the transmission. The only guarantees that are provided are those directly implied by the particular NoC architecture. For instance, in case arbitration has been implemented by the delay communication model, then it is guaranteed that the packet will not be destroyed (i.e., if not stuck forever, will sooner or later arrive to its $N_{destination}$).

- *guaranteed services* (GS) QoS level: the traffic having this level is granted with a set of specific guarantees. For instance, a maximum latency or a minimum throughput.

In the scope of this dissertations, GS level will only refer to case where the guaranteed metric is formally computed (i.e., not statistically inferred). The main reason for this, is the fact that predictability is a crucial aspect for a critical system. In case an architecture provides a stochastic GS, for instance minimum latency is statistically inferred, this architecture will be considered to be unable to provide a sufficiently

high level of QoS. There are several intermediate QoS levels proposed, which can be positioned somewhere in the middle between GS and BE levels. However, in the scope of this dissertation, the only relevant distinction is GS QoS level versus non-GS QoS level, where the latter will be referred as BE.

2.1.3 MPSoC examples

In this section I briefly present some of the MPSoC platforms. The scope of this section is to provide a general idea, while a more complete and detailed description can be found in literature [68, 67, 71].

- ***Arteris FlexNoC***
  Arteris FlexNoC [72] is not a MPSoC rather a suite for analyzing, synthesizing, and optimizing NoCs. Arteris, currently acquired by Qualcomm, is one of the leading commercial NoC tool providers [73], working with major semiconductor houses like Baidu, Mobileye, Samsung, Huawei / HiSilicon, Toshiba and NXP. The FlexNoC allows to automatically generate network topologies and to edit these interconnects down to the router level.

- ***Xeon Phi***
  Xeon Phi [74] denotes the Knights family of x86-based MPSoCs by Intel. This MPSoC was developed for high-performance computing and especially supercomputers, servers, and high-end workstations. It comprises up to 38 computing elements (called tiles), several memory controllers, I/O controllers and other IPs. These CEs are interconnected through a cache-coherent 2D mesh NoC. Each tile is composed of two cores, two vector processing units (VPUs) per core, and a 1-Mbyte level-2 (L2) cache that is shared between the two cores. Each core is an Intel Atom adaptation, heavily upgraded to incorporate features necessary for high-performance computing.

  However, Xeon Phi line was not the success that Intel was expecting. However, the serious issues Intel facing with the 10 nanometer manufacturing processes is the main factor for the sunset of Xeon Phi. Intel will ship Xeon Phi 7200s until July 19, 2019 [75].

- ***Tilera Tile***
  The Tile architecture was conceived as a research processor developed at

MIT [76]. Then Tilera start-up, founded by the original research group, first commercialized it. The Tile architecture was one of the first examples of a cluster-based many-core, featuring a NoC interconnection. Currently, TILE-Gx™ family is optimized for intelligent networking, multimedia and cloud applications. The device includes up to 72 identical processor cores (tiles) interconnected with the iMesh™ NoC. The latter is a 2D mesh NoC actually made of five distinct networks, using XY routing and wormhole switching. Each tile contains a single processor core, with local L1 (64 KB) and a portion (256 KB) of the distributed L2 cache.

- ***STMicroelectronics STHORM***
  STHORM [77, 78] stands for a ST(microelectronics) Heterogeneous lOw poweR Many-core, also known as Platform 2012. This MPSoC has been developed targeting to imaging, video, computational photography and augmented reality. STHORM is composed of four clusters, a fabric controller and memory controllers interconnected via two asynchronous NoC. Each cluster contains 16 STxP70 Processing Elements, each of which has a 32-bit dual-issue RISC processor with an operating frequency ranging up to 600 MHz. STHORM delivers up to 80 GOps (single-precision floating point) with only 2W power consumption.

- ***Kalray MPPA-256***
  The Kalray MPPA-256 processor [20] has been developed by the company KALRAY, targeting low-to-medium volume professional applications, where low energy per operation and time predictability are the primary requirements. The processor is manufactured in 28 nm CMOS technology and integrates a total of 288 identical very long instruction word cores. These cores are composed of 256 processing engines and 32 resource manager cores and are organized in 16 compute clusters and four I/O subsystems interconnected through the NoC. Each cluster is holds 17 cores: 16 processing engines and 1 resource manager core.

  The NoC is implemented by the two identical networks having 2D-wrapped-around torus topology and based on wormhole approach. It features bi-directional links providing a full duplex bandwidth up to 3.2 GB/s between two adjacent nodes.

2.2 Dependability

The *dependability* can be defined in many ways, one of which is presented in [1] as: "*the trustworthiness of a computing system that allows reliance to be justifiably placed on the services it delivers.*"

The dependability concept can be simplified to the concept of "*likelihood that the system will function properly*". At this point it is important to identify what does exactly means that a system is functioning properly, i.e., to identify dependability attributes. It is as well important to identify which are the threats to this proper functioning and which are the means to cope with these threats.

2.2.1 Attributes

There several accepted definitions for the dependability attributes. The definitions used in this thesis are [79]:

- ***availability***: the readiness for service or the probability that service delivered by a system is correct at any given time;
- ***reliability***: the probability that the system delivers a service correctly at a time T given that it was continuously delivering correct service from a time T_0, $T_0 < T$;
- ***integrity***: the capability of avoiding improper alterations:
  - 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.
- ***safety***: the capability of avoiding injuries for the individuals or catastrophic consequences for the environment;
- ***maintainability***: the capability of being repaired and evolved.

Safety and reliability are the aspects this dissertation puts focus upon. There are three aspects of system safety, which are [80]:

- *primary safety*: consequences inflicted directly by hardware (e.g., burns, electric shock);

- *functional safety*: consequences directly inflicted by a system failure (e.g., aircraft crash due to the flight-control system failure);
- *indirect safety*: the indirect consequences of a system failure (e.g., corruption of a medical database).

2.2.2 Threats

Dependability *threats* are any phenomena that can negatively affect the dependability of a system. In this thesis I will use the following terminology [79]:

- ***fault***: a defect in a system, which presence may or may not lead to a failure of the system;
- ***error***: a fault evolved into a discrepancy, inside the system boundary, between the intended behavior of a system and its actual behavior. The presence of an error may or may not lead to a failure of the system;
- ***failure***: an error evolved into a discrepancy, outside the system boundary, between the intended behavior of a system and its actual behavior. I.e., a discrepancy affecting the functionalities provided by the system.

In the literature it can be found several slightly different definitions of the term defect, especially for what it concerns its relationship with the concept of fault. From the above definition of fault, it is implied that a *defect* is defined as a variance between expected and actual of any aspect of the system. In order to avoid confusion, the term *fault* can be defined as: "any aspect of the system able, under a given circumstances, to provoke an error."

A fault inside the system can exist both inside a hardware component (*hardware fault*) or inside a software component (*software fault* or more commonly *bug*). The nature of a fault can be very different as very different can be the time the fault appears. In order to evolve into an error, a fault must be *excited*. For instance, a fault present in a system module which is never used, will never be exited and will never become an error.

Once a fault evolves into error, the same way seen for fault excitement, this error may or may not affect the services delivered by the system. For instance, a variable

can have a wrong value, but if this variable is never used, then the error will never have a chance to evolve into system failure.

An error is located, by definition, inside the system boundary. This means that the function delivered by the system is not affected and thus the presence of the error cannot be understood by observing the functionalities delivered by the system. However, the presence of an error can be identified by the dedicated mechanisms.

Taxonomy of Faults

The faults are normally classified in eight *elementary fault classes*, according to the following basic viewpoints [79]:

- *Phase of creation or occurrence*:
  - Development faults;
  - Operational faults;
- *System boundaries*:
  - Internal faults;
  - External faults;
- *Phenomenological cause*:
  - Natural faults;
  - Human-Made faults;
- *Dimension*:
  - Hardware faults;
  - Software faults;
- *Objective*:
  - Non-Malicious faults;
  - Malicious faults;
- *Intent*:
  - Non-Deliberate faults;
  - Deliberate faults;

- *Capacity*:
  - Accidental faults;
  - Incompetence faults;
- *Persistence*:
  - Permanent faults;
  - Transient faults;

This section will not go into detail of each and every of these classes, an interested reader can find more information in [2, 79]. The classes relevant in the scope of the dissertation are:

- The *life cycle of the system* is divided into two periods: a *development phase* and a *use phase* (also called *mission phase*). *Development faults* are those introduced during the development phase, while the *operational faults* are related to the mission phase of the system.
- *Human-made faults*, as the name states, are introduced by the human. On the other hand, *natural faults* are caused by natural phenomena without human participation.
- An important, although obvious distinction is the one of *hardware faults* which appear inside a hardware component and the *software faults* which appear inside software components.
- Finally, a *permanent faults* is a fault which can never disappear once it is inside the system, while a *transient fault* can disappear. An *intermittent faults* are often considered as a further type of fault for the persistence class. This fault is a particular type of transient fault and it is characterized by the high frequency it appears and disappears.

Taxonomy of Errors

When errors are considered, a slightly different notation is adopted. Often, the same word is used to describe slightly different situations, which can create confusion. In order to avoid this confusion, in this thesis I will use the following terminology:

- *Damage level*:

  - Destructive errors;
  - Non-destructive errors;
- *Persistence*:
  - Soft errors;
  - Hard errors;

This classification is important to distinguish different situations. The *destructive errors* are defined as those which will persist inside the system once they appear. The *non-destructive errors* are the opposite of the former. For what it concerns *soft errors* they are defined as errors for which "*a reset or rewriting of the device causes normal device behavior thereafter.*" On the other hand, a *hard error* is not defined as the evolution of a permanent fault, instead it is the opposite of a soft error. In practice that can either mean that normal device behavior can be restored by a power cycle or that it cannot be restored. This classification allows to identify the case in which a transient fault evolves in a non-destructive hard error. As the error is non-destructive and fault is transient, this means that the problem is only located inside the logical dimension and can be removed by a computer reboot. An implication of the above definitions is that a soft error can only be non-destructive.

Thesis Contribution Focus

The main contribution of this thesis (Chapter 3) targets the software faults of whatever class. The second contribution of this thesis (Chapter 4) targets the software faults of whatever class as well. The third contribution of this thesis (Chapter 5) concerns the hardware natural mission-phase transient faults. In particular, the focus is on the radiation-induce soft errors, which are described in detail in the following part of this section.

Radiation-Induced Faults and Errors

A radioactive environment is defined as an environment capable of exchange energy with the system in the form of radiation particles. This exchange can induce failure into the system.

Originally these radiation-induced faults were a prerogative of aerospace systems, as these systems can operate inside highly radioactive environment. However, due to the extremely shrink technology and low threshold voltages, this issue is becoming more relevant for avionics and even on ground for high performance computing and automotive.

A detailed analysis of either the effects of radiation on electronic circuits, or physical mechanisms radiation act, are beyond the scope of this dissertation.

The main effect of radiations on an electronic system concerns its integrated circuits. A particle strike on a transistor, if the particle has a sufficient energy, can create a current which can create a fault.

First classification of the radiation effects distinguishes between:

- ***single event effect*** (**SEE**): the effect of a single particle;
- ***total ionizing dose*** (**TID**): the effect of the total exposition to radiation;

The TID effect is a cumulative effect of the damages suffered by the system due to the particle strikes. TID effect is important when long space missions are considered, especially those operating inside the highly radioactive regions (e.g., Van Allen belts) of the space [81]. When the TID of a component exceeds its tolerable threshold, the component breaks down. As TID is a permanent fault, the system is incapable of providing its service till the damaged components are replaced. However, for a space mission it is often impossible to replace a component. This means that components with a sufficiently high TID tolerable threshold should be selected at design phase.

The focus of thesis, for what it concerns the contribution described in (Section 5), is put upon SEE instead. The effects of SEE are generally classified as follows:

- **single event upset** (**SEU**)
  (non-destructive, soft error)
- **single event latchup** (**SEL**)
  (non-destructive or destructive, hard error)
- **single event burnout** (**SEB**)
  (destructive error)

A *single event upset* SEU is a non-destructive soft error is defined by NASA [82] as: "a change of state or transient induced by an energetic particle such as a cosmic

ray or proton in a device."
A SEU can be further classified according to the spatial characteristic of the effect:

- single event transient (SET);
- single-bit upset (SBU);
- multiple-bit upset (MBU);

A *single-bit upset* SBU refers to the flipping of one bit inside a memory element of the system. A *multiple-bit upset* MBU is the bits causing simultaneous flipping of multiple bits of the memory elements of the system (provoked by the same SEU. Finally, the *single event transient* (SET) error considers the particle strike of the logic circuit and represents a voltage disturbance inside the latter. A SET can eventually propagate through the logic becoming either an SBU or an MBU.

A *single event upset* SEL is an either destructive or non-destructive hard error which can be defined as an error caused by a high current state. This error can be provoked, for instance, (considering CMOS technology) by PNPN parasitic structures activation. Although this is not a destructive error, a high current can induce several distributive faults.

A *single event burnout* SEB is a destructive error and identifies a permanent device destruction due to the high current state in a power transistor.

For what it concerns the failure classification, ***single event functional interruption*** (**SEFI**) failure is defined to identify a specific type of SEU induced failure. A SEFI is defined as a SEU-induced failure, with is abrupt and immediately observed. The main characteristic of a SEFI is that this failure does not require power cycling of the device (i.e., turn off and back on) to restore its normal functioning. A SEFI has been defined to describe a situation in which a bit flip (often in a bit register) makes the system restart or hang. This means that a (radiation-induced) fault immediately evolve in a severe failure of the system. The immediacy aspect of a SEFI is fundamental, as this means that no software-implemented approach can be adopted to prevent the error (SEU) from evolving into a failure. This means that, if hardware-implemented fault tolerance is not properly implemented a single ionization particle strike will provoke a complete failure of the system. It worth to notice how a SEFI differs in many aspects from a SEL induced failure. The more apparent difference concerns the fact that, to restore the correct functioning, the latter

will require to power cycle the device. However, the most important difference lies in the immediacy a SEU becomes a SEFI, which does not allow the error detection.

2.2.3 Means

The literature [2] categorizes means to achieve a desired dependability as:

- ***Fault prevention***: means to prevent the occurrence or introduction of faults;
- ***Fault tolerance***: means to avoid service failures in the presence of faults;
- ***Fault removal***: means to reduce the number and severity of faults;
- ***Fault forecasting***: means to estimate the present number, the future incidence, and the likely consequences of faults.

Fault prevention means aim to prevent the occurrence or introduction of faults. Examples of this category are: software development process improvement, chip manufacturing processes improvement, packaging protecting from radiations.

Fault tolerance means means aim to avoid service failures in the presence of faults. As the main contribution of this thesis falls in this category, this is particularly relevant in the scope of this dissertation. are the most relevant in the scope of this thesis, as it concerns the main contribution of the latter. Fault tolerance means aim at failure avoidance by either detection and recovery and/or masking. The detection can be done at both fault- and error-level and it is generally achieved by means of redundancy, which can be done at any system hierarchy level. An important classification of fault tolerance strategies concerns whether a fault detection takes place or not as well as the moment it takes place (in case fault detection is implemented). *Detection and recovery* implementation strategy is characterized by first detect the error and then to perform a recovery action. *Masking (and recovery)* implementation strategy, on the other hand, uses the redundancy means to mask the error and to provide the service continuation. Normally the error is also the detection feature, as this will allow a more efficient error handling (e.g., corrective maintenance process activation). Thus, this approach normally implements fault recovery features as well. The term *resilience* is sometime used in literature as a synonym of fault tolerance and sometime as the general characteristic of the system to resist the fault appearance (e.g., usage of packages shielding against radioactivity).

To avoid such ambiguity, in this dissertation I will use the term resilience in a broad sense, targeting both the previously listed cases.

Fault removal means aim to reduce the number and severity of faults. This means can be both applied at design time and mission time. Fault removal means applied at design time are based on verification, diagnosis, correction; An example of this case is the housekeeping sensor data analyses used to diagnose potential design problems, thus, the design is modified accordingly. An interested reader can refer to [83], where the author gives a small contribution to the described example, which is not described in this thesis. Fault removal means can be also applied at mission time by replacing physically damaged parts or applying patches.

Fault forecasting means aim to estimate the present number, the future incidence, and the likely consequences of faults. The main approaches for probabilistic fault-forecasting are modeling and (evaluation) testing, which are complementary.

2.3 Safety-critical, Mission-critical and Non-critical Systems

A mixed-criticality system can be seen as a generic case of a critical system; thus, the latter is a crucial concept of the context of this dissertation.

The term *critical system* is a generic name which can refer either to safety-critical system or to a mission-critical system. The difference between those systems is relevant also for this dissertation, so the following distinction must be done:

- **safety-critical system**;
- **mission-critical system**;
- **non-critical system**.

A *safety-critical system* is a system that in case of failure can provoke injuries or death of individuals. In some case a safety critical system also concerns severe environmental damage. The main requirement for such systems is safety (see Section 2.2.1). Typical examples of such systems are automotive, avionics and railway systems.

A *mission-critical system*, which failure will provoke considerable financial losses, but not the consequences characterizing a safety-critical system. The main requirements for such systems are reliability, availability and maintainability (RAM) (see Section 2.2.1). Typical examples of such systems are satellite systems.

Finally, a *non-critical systems* is a system which neither safety-critical nor mission-critical. The failure of such system is not able to affect the health of people, provoke environmental damage or considerable economical loss.

2.4 Mixed-criticality: Requirements and Issues

This section has two goals, first of all, to describe the requirements a mixed-criticality system (MCS) must fulfill, according to the industrial standards. Secondly, to identify the MCS-specific issues for what it concerns the usage of MPSoC.

First of all, it is fundamental to understand the relation between an MCS and the industrial standards it must fulfill. In fact, an MCS is just a particular type of a critical system, and there are industrial standards specifically targeting the *mixed criticality* nature of this system. In other words, industrial standards do not provide any requirements specifically for an MCS and the latter must fulfill the same requirements as any other critical system. Thus, the main point is that mixed-criticality attribute presents several additional issues, rather than a set of additional requirements.

In this dissertation I will mostly refer to the avionic standards (e.g., DO-178C [9]) and space standards (e.g., ECSS-Q-ST-80C [84]). However, the basic functional safety standard IEC-61508 [8] will be considered as well.

2.4.1 Critical System Design

During the development phase of a critical system lifecycle, a list of requirements must be created to list the system expected behavior and characteristics. These requirements are included the functional requirements, i.e., which are the services a system should provide, but also the non-functional ones. The latter concerns the dependability attributes desired figures and other constrains like energy budget.

This section briefly explains the main aspects of a safety-critical design flow, with a particular focus on avionic standards. The mission-critical system will not be explicitly considered; however, the fundamental concepts are essentially the same [14]. At a certain extent, a mission-critical system can be seen as a sort of safety-critical system in which the concept of *human safety* is replaced with the concept of *mission safety*. I.e., the focus is moved from avoiding harm to individuals to avoiding the mission failure and the key attributes become reliability, availability and maintainability (RAM).

The most important aspect for a safety-critical system, as the name suggests, is the safety aspect (attribute) of the system dependability. In this section and in this dissertation in general, I will only consider functional safety (see Section 2.2.1) as the latter is the more relevant for the topic of the dissertation. The safety assessment process is in charge of characterizing the failure conditions from the safety point of view (e.g. through a hazard analysis). Starting from desired safety figures, safety assessment process derives a list of safety-related requirements, concerning both hardware and software components. These requirements can be classified as mechanisms to:

- prevent a fault occurrence;
- mitigate the effects of errors and/or failures;
- avoid the propagation of errors and/or failures.

The rest of this Sub-section will first describe an avionic safety-critical system development lifecycle, providing a brief overview of the safety assessment process and the related the development assurance level (DAL) assignment. Then, the partitioning concept will be specifically considered.

Safety Assessment Process

A modern aircraft is an example of a complex system, for which the safety aspect is of paramount importance. Due to its complexity this system can be seen as a system of systems, thus a rigorous methodology is required to effectively address this complexity. The first revision (ARP4754 [85]) of ARP4754A (*Guidelines For Development Of Civil Aircraft and Systems*) [86] was published in 1996, written by a team of aircraft and avionics manufacturers with participation of certification

authorities. The current version purpose is to define an accepted development assurance process, where the development is related to entire system development process. ED-79A (Guidelines for development of civil aircraft and systems) [86] is an European equivalent of ARP4754A. In this thesis I will refer to the standards recognized by Federal Aviation Administration (FAA), which is governmental body of the United States. However, for each of them, an European equivalent exists, recognized by European Organization for Civil Aviation Equipment (EUROCAE).

With the purpose of the aircraft safety assessment, according to ARP4754A, the first task to be performed is typically the *Safety Program Plan*. It identifies the process to follow in order to assess the aircraft safety, safety standards will be applied, and other aspects related to the former process. This thesis will not go into detail on the ARP4754A process as such, nor into the details of all the integral processes it defines. Only the safety assessment process will be taking into consideration.

The *safety assessment process* identified by ARP4754A is intended to run parallel to the system development process and achieved by the means described in ARP4761 [87] (entitled *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*).

The first step of the safety assessment is hazard assessment. It is performed after the basic functionalities (and their conceptual design) are defined. A hazard is defined in FAA Order 8040.4 [88] as a "*condition, event, or circumstance that could lead to or contribute to an unplanned or undesired event.*" Chap. 2 of FAA System Safety Handbook [89] states that hazard identification and analysis shall "*identify the safety risks associated with the system or operations under evaluation. The risks shall be characterized in terms of severity of consequence and likelihood of occurrence ...*"

The following step, which can be generically called "fault analysis", assigns the severity levels to the functions associated with the identified hazards. Table 2.1 shows the severity classifications with the assurance levels and required probabilities for a large transport category aircraft.

The design assurance level (DAL) according to ARP4761 [87] is:

> *the measure of rigor applied to the development process to limit, to a level acceptable for safety, the likelihood of errors occurring during the*

Table 2.1 DAL: failure condition severity, probabilities [1/flight hour], and levels

Severity Classification	Design Assurance Level	Probability
Catastrophic	A	1×10^{-9}
Hazardous	B	1×10^{-7}
Major	C	1×10^{-5}
Minor	D	1×10^{-3}
No safety effect	E	1

development process of aircraft/system functions and items that have an adverse safety effect if they are exposed in service.

A simplified view of the DAL concept is considered in this thesis. Considering the reliability attribute to describe the continuity of service (as described in Section 2.2.1, each severity level implies a given reliability figure. Thus, a DAL can be seen as a set of design process requirements, aiming to assure that the estimated reliability (e.g., of a component) is trustworthy.

Safety assessment process performs, among others, DAL assignment at the system level. The software and electronic hardware components are called *items* in ARP4754A. RTCA DO-178C is applied to the software items, while RTCA DO-254 is applied to hardware items. Item DAL (IDAL) concerns the software and electronic hardware, while functional DAL (FDAL) concerns the system's functional level and it is out of the scope of this thesis. ARP4754A explains the IDAL assignment approach, specifying what DO-178C objectives apply for software or DO-254 objectives for electronic hardware. An interested reader will find more information in [90]. The IDAL is called a *software level* in DO-178C. However, in order to avoid confusion, in this thesis the term DAL will be normally used.

The means to perform the DAL assignment, as any other means used for safety assessment process, are specified in ARP4761. The most known techniques are Fault Tree Analysis (FTA) and the Failure Modes and Effects Analysis (FMEA). The first is a top-down technique while the second is generally used as a bottom-up approach. These can be seen as complementary techniques and Section 3.2 of ARP4761 explains how these techniques relay on each other.

Once safety assessment process is performed, each item is assigned with a DAL, which is the highest among the DAL of the functions that item implements. Each DAL implies a set of strict requirements concerning:

- failure prevention;
- failure propagation avoidance;
- failure effect mitigation.

Most importantly, a DAL specifies requirements in terms of methodologies to be used, to fulfill the aforementioned.

Fault Effects Isolation: Partitioning

As seen in the previous section, a DAL establishes a set of requirements on failure prevention and failure propagation avoidance. To achieve these requirements the components (i.e., items according to ARP4754A) must be properly isolated from each other to prevent error propagation and/or failure propagation.

The *partitioning* concept is of paramount importance for any critical system (not only for a safety-critical system nor for the avionics domain). The partitioning goal is to provide system components with sufficient *independence*, without which the very concept component falls. In fact, given two components (e.g., software components), if a sufficient independence cannot be demonstrated between two, then these are actually not *components* for what it concerns the hazard analysis (Section 2.4.1). Instead, these two objects have be considered as macro-component, having the highest failure severity of the two. For instance, if the isolation is not implemented at all, then there will not be a *component severity level* but only a severity level of a system as the whole.

The partitioning can be seen as tool to achieve hierarchical design and thus to implement the well-known *divide et impera* principle (i.e., the principle of splitting a problem into two more simple ones and solve them separately). In fact, given two independent functions, these can be implemented by (i) a monolithic system or (ii) by a dedicated sub-system each and the overall system done by the cooperation of these sub-systems. The second solution is the well-known hierarchical design approach, and it is possible if the sub-components are sufficiently isolated from each other (otherwise the design will result monolithic). It is easy to understand that in case it’s crucial to assure a system will (virtually) never fail, it is essential to split the system into simpler sub-systems to achieve a manageable level of complexity.

The software partitioning described in DO-178C is very similar to the description provided in IEC-61508. Under RTCA DO-178C, the following software design methods are described: *partitioning*, *dissimilarity* (of *multi-versioning*) and *safety monitoring*. The software component isolation should be implemented by the partitioning. This is one of the most important design instruments (or concepts) to safety-critical systems (not only for avionics domain). The multi-versioning consists in two or more components implementing the same function to be developed independently. The goal of this technique is to provide the redundancy (thus to improve the reliability) to the system, while avoiding common source of errors. Finally, the safety monitoring is an active mechanism to implement the protection against specific failures. This mechanism also monitors the correct functioning of the partitioning technique.

The partitioning mechanism, described by the RTCA DO-178C, must address both the extent and the scope of the interactions between the partitioned components as well as how to isolate the components from each other. The isolation must be achieved in both spatial and temporal domains. *Spatial isolation* means that one application shall not corrupt data of another application. *Temporal isolation* on other hand concerns the timing properties and means that one application shall not cause failure (i.e., deadline miss) of another application by blocking a shared resource (e.g., CPU, interconnection). To achieve spatial isolation is relatively easy as the hardware means like memory management unit (MMU) can be exploit. The situation with the temporal isolation is more complex. The safety-critical domain is niche market so the COTS MPSoC features no hardware means to address with temporal isolation issue. On the other hand, the development of custom solutions should justify the related NRE cost.

DO-178C requires the designer to address the partitioning implementation rigorously defining:

- all the interactions that will be allowed between the (partitioned) components;
- the (hardware, software or hybrid) techniques used to isolate the components from each other.

DO-178C also establishes five requirements on the partitioning implementation:

- the code and data of a software component must be isolated from the other components;

- software component is only allowed to consume CPU time during its scheduled period of execution;
- each component must implements proper error and failure containment techniques;
- software that implements the partitioning, should have the same or higher DAL than the highest DAL among the partitioned components.
- if the partitioning is implemented via hardware, the safety assessment performed on that hardware must explicitly consider its partitioning functionality.

2.4.2 Mixed-criticality System: Processor Level

A mixed-criticality system (MCS) is a critical system composed of components having different DALs (SIL, ASIL, etc.).

When academic literature and the industrial standards are considered, the term *criticality* is used to indicate similar — but still different — concepts. In this dissertation *level of criticality* will be used as a synonym of level of severity of the component failure.

Being an airplane a complex system, basically a system of systems, the mixed criticality concept can be applied at different hierarchical levels. The mixed-criticality concept concerning the high level, i.e., view of systems being themselves components of the overall system (i.e., airplane system), is not an open issue and it is out of the scope of this thesis. In fact, the open issue is a creation of a MPSoC-based system integrating multiple software components on the same processing platform, not the isolation between such systems (as the latter will be implemented in the exactly the same way it is implemented currently for non-MPSoC systems).

In this thesis I will focus on a particular case of MCS: software components having different DAL running on the same processing platform. I.e., mixed-criticality paradigm adopted to the computer level: functions (the related software modules) with different DAL running on the same processing element. From now on, I will use the term MCS to refer to this particular kind of MCS only.

The partitioning concept, described in Section 2.4.1 in not an MCS's prerogative (i.e., not an MCS's uniqueness). However, the partitioning is critical for such systems

and it can be identified as the main issue preventing multi-core and MPSoC from being used.

From the requirements and issues presented in Section 2.4.1, it should be clear how the federated architecture (i.e., one function - one computer) paradigm was an easy solution to assure the isolation between the components of the system. However, the ever-growing number of functions and their complexity made federated architecture paradigm no longer sustainable. Driven by the growing maintenance cost and other factors, industry developed several standards (e.g., [12], [7]) to standardize the possibility for multiple functions (related software components) to run on the same hardware platform. Furthermore, even the execution of software components having different DAL (or level related to the specific domain) was standardized. Thus, the mixed-criticality paradigm was adopted at processor level.

For avionics, ARINC 653 [12] defined APplication EXecutive (APEX) application programming interface (API) standard. This API was intended to decouple the operating system from the application software. This allowed to run more applications, even having different DAL, on the same computer.

2.5 MPSoC-based Mixed-criticality System

The concepts presented in the previous sections are related to a general processing element, without specifically targeting either single-core or multi-core processing element (chip). However, as the system evolution started from single-core processors, the standardization did the same. Currently, at the best of my knowledge, there are no multi-processor–based MCS standardized for the main critical domains.

The aviation domain community constantly increase their attention towards multi-core processor–based MCSs. Considering the avionic domain, the question on whether the DO-178B/C and DO-254 are adequate and sufficient to cover multi-core processor seems to be still unanswered. However, the issue is under investigation and it is addressed by several technical reports issued by or in collaboration with the certification authorities. These both considers the selection (and evaluation) of microprocessors [91, 92] and the specific issues related to the usage of COTS microprocessors [93].

Recently, a technical report was issued by FAA entitled *Assurance of multicore processors in airborne systems* [94]. This report identifies the *lack of predictability* as the main concern regarding the use of MCPs in the safety-critical aerospace domain. It also, among other things, explicitly targets the *interference-aware safety analysis*, identifying three sequential steps:

1. the identification of an interference path;
2. performance of an interference analysis to tag each interference channel as acceptable, unacceptable, unbounded, or faulty;
3. determination of interference mitigation.

Among the expected-soon certificated multi-processor–based MCS, the closest one seems to be the one in process by Rockwell Collins [13], which is expected by the end of 2019. This certification targets the NXP T2080 processor, featuring four cores and several peripherals.

The MPSoC device (discussed in Section 2.5) is an evolution of multi-core processor. The issues related to this kind of devices are identical to those related to multi-core processors. The key difference is the presence of NoC interconnection which presents an additional complexity with respect to the classic bus-based interconnection. However, the NoC brings more flexibility. This can be the key to cope with its complexity. Furthermore, the MPSoC paradigm is based on the concept of integrating more-but-simpler-cores, with respect to the tread of creating complex out of the order computing cores of a classical multi-core processor. Thus, the MPSoC paradigm is offering a complexity reduction at core level, which can be a precious aspect from the certification point of view.

Chapter 3

NoC partitioning and RTOS filtering module: COTS NoC-based MCS

This chapter describes the first contribution of this dissertation: a technique to solve one of functional safety issues related COTS (NoC-based) MPSoC usage in the context of mixed criticality. In detail, the technique addresses the temporal isolation issue related to implicit contention of the NoC interconnection. The proposed solution is based on a one-to-one bound of each NoC internal resource to a software module (i.e., application). Thus, the NoC is partitioned between the applications and the resources are prioritized to avoid any traffic interference. Furthermore, the solution does not allow any inter-application resource sharing. In this way, the temporal partitioning between software modules is achieved by denying any NoC internal resource sharing. This partitioning scheme is monitored, and it is enforced by filtering faulty traffic (i.e., by denying any NoC usage that would break the partitioning scheme). This traffic filtering exploits the deterministic routing used by the NoC. Thus, the main requisite for the proposed solution is that the routing algorithm (used by the NoC) should be deterministic and known. To leverage the connectivity reduction, innate for any resource privatization technique, a redirection feature is granted for the non-critical traffic. The filtering module, in charge of monitoring, is implemented as purely software module to be inserted inside the RTOS. The overall solution is deliberately simple to exhibit a high certification potential. The proposed solution also implements spatial partition between critical applications. However, the focus put on the temporal domain as the spatial partitioning is much easier to achieve and does not represent an open issue. Some of the security issues,

for instance DoS attack, are solved by the proposed solution as well. However, the security aspects are out of the scope of this dissertation and will not be considered. The proposed approach allows to overcome a strict domain-based partitioning by performing NoC resource reservation at router level, instead of node level.

The solution described in this chapter was first presented in [39] and extends the technique supporting only two levels of criticality presented in [95]. Where the technique proposed in [95] is an improved version of the original solution proposed in [96].

The rest of this chapter is structured as following: Section 3.1 presents the state-of-the-art solutions to face the NoC temporal partitioning issue; Section 3.2 presents proposed solution from the theoretical point of view; Section 3.3 presents the implementation of the proposed solution for what it concerns its core functionality; Section 3.4 presents the implementation of an auxiliary technique to mitigate the cost of the core functionality of the proposed solution; Section 3.5 presents the experimental evaluation of the proposed solution; finally, Section 3.6 summarizes the key aspects of the proposed solution;

3.1 COTS NoC-based MCS: State-of-the-art

To cope with the temporal isolation issue in MPSoC, the literature provides several solutions. Some of these solutions are adopted from bus-based multi-core related techniques, while the other have been created specifically for NoC-based MPSoC. The main issue with these solutions is certification potential, as them only indirectly consider the industrialist's perspective.

This section will provide an extended analysis of the state-of-the-art related to the usage of commercial-off-the shelf (COTS) MPSoC components in the scope of critical systems. The techniques targeting the usage of custom NoC design are out of the scope of this chapter. Those techniques will be considered in Section 4.1.

The techniques that can be applied to COTS MPSoC can be further classified according to the main idea behind them. This classification can be done according to several criterion. In the scope of this dissertation I identify two macro-categories of techniques, according on how the NoC resources (links, FIFOs, etc.) are managed to ensure timing isolation: resource sharing and permanent resource reservation.

Where the techniques of *permanent resource reservation* (or *resource privatization*) macro-category refer to the opposite situation with respect to the *resource sharing* scenario. I.e., each NoC resource is dedicated to an application and it is not shared with other applications running on the MPSoC. The QoS is often mentioned by the techniques from resource sharing macro-group, as the HI tasks are guaranteed with bounded transmission latency, throughput o similar metrics. Another minor distinction concerns whether a technique is conceived for a specific MPSoC or it can be applied for a generic MPSoC (although fulfilling a set of requisites).

3.1.1 NoC Resource Sharing -based techniques

A first group of solutions are those targeting robust estimation WCET, explicitly facing the shared (NoC) interconnection issue. Some of these solutions face the temporal isolation issue explicitly, while other either face that issue implicitly and indirectly or directly ignore the issue. Those techniques which ignore the temporal isolation issue are considered out of the scope of this dissertation.

In [15] the authors propose a temporal partitioning mechanism and a WCET estimation exploiting such mechanism. This temporal partitioning is an extension of the solution adopted for single-core processors to solve the shared bus contention between processor and I/O devices [97]. It is implemented by a run-time resource monitoring and enforcement tasks. NoC is considered one of a set of shared resources this temporal partitioning is applied to. Monitoring task, using a counter, measures the utilization of each shared resource by each process running on the MPSoC. If a predefined bandwidth is exhausted, then the temporal partitioning enforcement task (called *suspension task*) is invoked. The purpose of this task is to deny a further usage of that particular shared resource by that particular process for a period of time. The WCET calculation exploits the knowledge of maximum bandwidth imposed for each (process i, shared resource j) couple. This knowledge makes the interference-delay analysis easier. The authors propose an interference-delay analysis which computes the maximum possible inter-process interference and the resulting increase in execution time. The main issue with the proposed approach is to ignore the particular nature of NoC. For instance, considering wormhole flow control, a packet is injected inside NoC one flit per (NoC's) clock cycle. Let's assume at a given time t_k), $process_i$ exhausts its bandwidth associated to NoC usage. This means $process_i$ is prevented from injecting further flits inside the NoC. This will probably require

the generation of a special tail flit to release the resources and to signal the receiver that something went wrong. This is not an issue as the time to do this can be easily upper bounded, so the $process_i$ will stop injecting packets at time $t_{k+\delta}$. The true issue is that a packet (i.e., its flits) injected by $process_i$ can remain stuck inside the NoC for a long time, thus $process_i$ will continue using NoC although $process_i$ is not injecting packets any more.

The main open issue of the previous solution has been addressed by [16]. The authors adopt trajectory approach (used in the context of switched Ethernet networks) to the context of NoC. The authors present a WCTT analysis which does not require implementation of VCs. Although the WCTT analysis [16] could seem to integrate the time partitioning method presented in [15], it is all to be proven. In particular, this WCTT analysis integrated in the context of bandwidth monitoring described in [15], will result in extremely bad (from the performance point of view) figures. In fact, the worst scenario should consider the NoC congested (in the worst manner possible) by faulty traffic. This situation will have a huge impact on WCTT due to upstream and downstream indirect interference. Furthermore, this can easily make WCET explode as a process normally need multiple usage of the NoC. Furthermore, the overall solution exhibits a considerably high level of complexity.

Several solutions were derived which are relying on VCs to implement priority-based preemption. Considering the more recent works, the authors in [17] address wormhole NoCs with priority-based preemptive arbitration. This work presents an extended analysis model to estimate delay upper bounds for all router architectures and buffer sizes. One of the issues related to VCs is the complexity of the priority-based preemptive arbitration, which will play against the certification potential. A further issue is the absence of isolation between the processes having the same level of priority. A yet further issue is the cascade effect a faulty HI traffic which can induce starvation to all the processes having lower levels of priority.

There are some solutions aiming to monitor the interconnection actual congestion and to identify the anomalies. This technique can be used to cope with the issues of the previously described techniques. For instance, the authors in [18] make use of performance counters to identify the rise of cache-dependent stall cycles for a given application. However, these solutions are so far only for bus-based systems.

There are also several solutions based on probabilistic timing analysis (PTA) to estimate pWCET [98, 99]. However, as already explained in the dedicated section,

this methodology has robustness issues and it is not currently accepted by the critical domain industry.

Another group of techniques is based on hierarchical scheduling of the NoC. This approach is based on the usage of a scheduling entity which queues the tasks of a certain priority. This scheduling entity will execute the tasks starting from the queues having higher priority. The hierarchical scheduling offers a higher resource utilization, compared to the classical time-triggered scheduling. These advantages are especially relevant when considering the context of mixed criticality.

Considering the more recent solutions, [19] extends the approach proposed in [100]. The authors introduce the *self-aware network-on-chip control*. The goal this approach is a predictable timing of communication as well as traffic segregation for mixed-criticality applications. This is implemented using come nodes of the NoC as centralized arbitration units called *resource managers*, which are in charge of controlling network access. The RM is equipped with a scheduler, which is responsible for allocation and releasing of resources for ongoing transmissions. The main issue of hierarchical scheduling solutions is a high complexity. Furthermore, considering the proposed solution, it is not clear how the temporal isolation holds under a faulty behavior of one of the LO nodes.

Apart from the techniques derived for a class of NoC architectures, there are some conceived specifically for specific MPSoC (i.e., its NoC architecture). Among the latter, several techniques based on hierarchical scheduling are conceived for Kalray MPPA® 256 [20]. These techniques [21–23] exploit the special hardware the MPSoC is featuring. The main limit of these techniques is that they rely on a complex hardware could be an issue under the certification point of view.

3.1.2 NoC Resource Permanent Reservation -based techniques

An alternative to the resources sharing techniques (presented so far) consists in permanent resource reservation or resource privatization. In this macro-category, the NoC is not seen as a monolithic resource, instead it is considered as a set of resources (i.e., links, FIFOs, routers, etc.) potentially independent from each other. One of these techniques [24] is developed for for Tilera Tile Processor [25]. This technique (permanently) partition the NoC into regions, each node belonging to a region can only communicate to nodes of the same region. This solution is fairly

simple, which is an advantage from the certification point of view. The issues related to this approach are limits of proximity requirements for the nodes, as well as the specific hardware like the one featured by Tilera Tile Processor.

3.1.3 Gaps Inside the State-of-the-art

From the state-of-the-art analysis it can derived the list of issues of the existing approaches in dealing with temporal isolation issue. The main issue is the complexity of the solution, which has the direct impact on the certification feasibility. On the other hand, the low-complexity solutions are not complete as a set of issues is only cursorily and indirectly addressed. These issues are protection against timing failure propagation, dependency on a specific hardware, dependency on a rigid process placement (i.e., process to CE mapping) constrains and other. The solution proposed in this chapter aims to be a low-complexity one, still addressing the main issues of the temporal partitioning. Considered the state-of-the-art, the proposed solution is close to the one presented for Tilera Tile processor [24]. However, the similarity only refers to the resource privatization approach. Differently from the [24], my solution is not bound to a specific hardware and allows a much flexible NoC usage.

3.2 Proposed Partitioning Solution

Considering COTS MPSoC, a solution is derived to face the temporal isolation issues related to implicit contention of NoC interconnection, mainly targeting avionic field. The proposed solution is a partitioning technique based on resource privatization, thus there is no resource sharing between critical software components (i.e., applications). The overall solution can be considered composed of two parts: partitioning placement and safety monitoring. An off-line phase, done at system design time, creates a placement for critical applications. This placement is done in such a way to avoid any contention for NoC internal resources (from the critical applications). Thus, the temporal isolation is achieved for the critical applications. However, a mixed-criticality system must provide safety monitoring task in order to assure a failure cannot propagate from an application to another. This task is accomplished by the second and core part of the proposed solution. In detail, a module is in charge of monitoring the traffic attempting to enter the NoC. If this traffic, once injected inside

the NoC, would violate the temporal partitioning, then the monitoring module filters (discard) that traffic. The proposed solution also features a module to leverage the connectivity reduction, innate for any resource privatization technique. In detail this is done by means of a redirection feature for the non-critical traffic. The rest of this section is structured as follows: Section 3.2.1 describes the NoC model the proposed solution is intended to be applied to; Section 3.2.2 and Section 3.2.3 explain the idea, which is behind the proposed solution, respectively for what it concerns the off-line partitioning and the on-line monitoring and enforcement task (while the actual implementation is described inside Section 3.3); Section 3.2.4 explains the idea which is behind a technique to reduce the cost or the proposed solution (while the actual implementation is described inside Section 3.4).

3.2.1 Considered NoC Model

The proposed solution is developed targeting simple NoC architectures. As already explained, this choice allows to keep certification costs low. It is worth to highlight that a high architecture complexity does not prevent the proposed solution as such; the problem is instead that architecture itself, as its complexity will be a huge issue from the certification point of view.

In the scope of this chapter the proposed solution will be explained and validated for a specific NoC architecture, although the proposed solution can be applied to any NoC using a deterministic routing as well as having known routing algorithm and topology. This NoC architecture is a 2D mesh topology, using XY routing, as can be seen in Fig. 3.1. The figure shows the path taken by a packet send by N_0 towards N_5, according to XY routing algorithm. The considered NoC design uses wormhole switching and input-buffered ports. As mentioned in Section 2.1.1 this architecture is one of the most commonly used. Finally, it is based on a single physical network (i.e., not featuring VCs mechanism) implemented by links made of two physical channels, one per direction. This characteristic falls under the simplicity requirements discussed shortly above (in the previous paragraph). In case an architecture, featuring VCs mechanism, is considered, our solution is perfectly usable by simply not using the virtual networks. More information on the NoC implementation choices is present in Section 2.1.1.

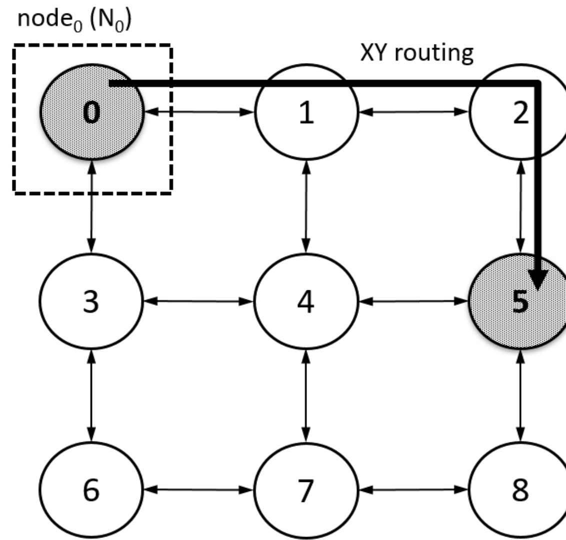

Fig. 3.1 A schematic view of a 3x3 (2D) mesh topology NoC; N_0 to N_5 packet path under XY routing algorithm.

Fig. 3.2, provides a detailed view of the NoC, showing how each node is made by a *connected element* (CE) connected to its dedicated *network interface* (NI) which is in turn connected to its *router* through two physical channels (one per direction). Fig. 3.3 depicts router architecture of the considered NoC design. More information on the NoC building blocks and the router implementation is present in Section 2.1.1.

3.2.2 Off-line Partitioning: Applications Placement

The partitioning is done during the applications placing phase, and it is the first part of the proposed solution. During this phase, for each application, it is decided which MPSoC resources (e.g., processing elements, peripherals, memory) will be used by that application. The NoC interconnection itself is seen as a set of resources (i.e., links, FIFOs, etc.) to be partitioned between the applications. Each critical application has a dedicated partition, formed by the resources dedicated to that application. In order to exhibit the lowest complexity possible, the proposed solution considers a scenario where no inter-partition resource sharing exist for critical partitions. This means that each critical application is using the NoC without its traffic being able to interfere with the traffic of any other application. Thus, concerning the critical applications, the temporal partitioning is done not only

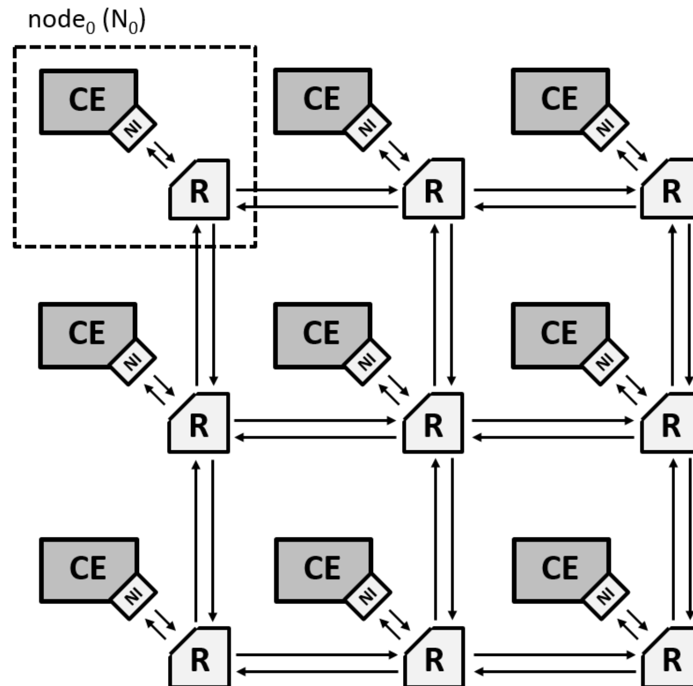

Fig. 3.2 A detailed view of a 3x3 2D mesh topology NoC; R - router; NI - network interface; CE - connected element.

between applications having different levels of criticality but also between those having the same criticality. For what it concerns the non-critical applications, these do not have a per-application dedicated partition. This means that it does exist a macro partition made of all non-critical applications. Thus, the non-critical applications share the MPSoC resources (not dedicated to any critical application). The proposed solution does not impose any resource sharing policy for non-critical applications, leaving in place the original MPSoC resource sharing policy. Thus, from the QoS point of view, the proposed solution implements two levels of service: GS and BE. Critical software modules are granted with GS level, and are guaranteed latency and throughput as no NoC contention do exist. Non-critical applications exhibit BE level, as only the guarantees originally provided by the MPSoC (which are usually best-effort) are in place. The proposed techniques support an unbounded number of criticality levels, as each application running under the GS level is granted with the temporal isolation.

Since there is no resource sharing between critical partitions, the proposed approach implements the spatial isolation (in the simplest way possible). However,

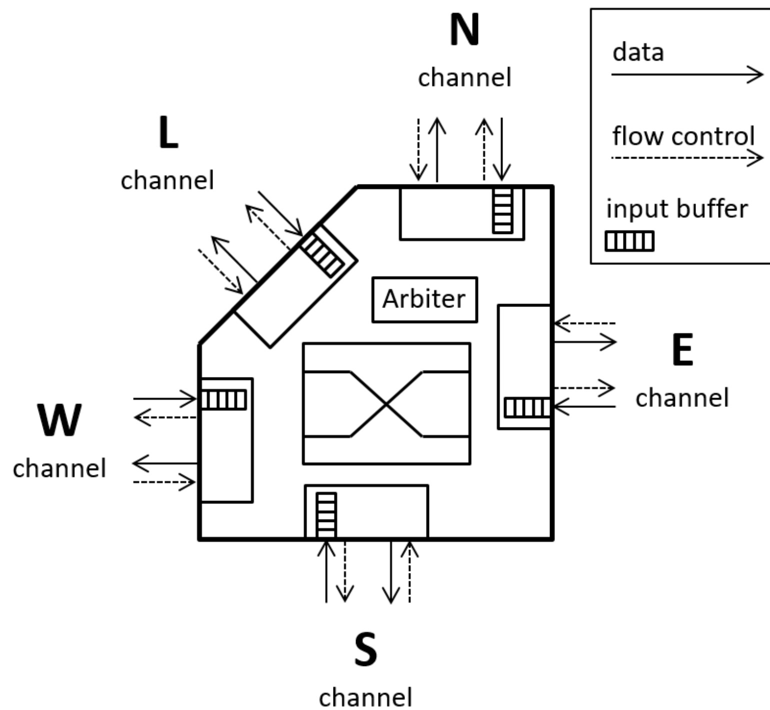

Fig. 3.3 A schematic view of a NoC router architecture; a simplest input-buffered router architecture is considered, based on a single physical network with no virtual channels; link made of two physical channels, one per direction.

the focus put on the temporal domain as the spatial partitioning is much easier to achieve and does not represent an open issue.

The proposed partitioning technique overcomes the classic *strict domain* (o *partition*) *segregation* scheme (e.g., the one proposed in [24]), for instance, partitions can overlap (without violating traffic isolation). The main aspect is that a router can be used in parallel (and simultaneously) by different software components (thus different partitions) as far as there is no contention of the router's internal resources. This means that a router, as the NoC itself, is shared by several software components (even having different levels of criticality). However, de facto there is no sharing as each component of the NoC is privatized by one and only one partition (i.e., software component). This is a strong point of the proposed solution as it exhibits a much higher flexibility and allows communication between distant nodes without reducing the overall NoC connectivity. Considering a heterogeneous MPSoC, this flexibility characteristic allows a much better system utilization with respect to the

strict partition segregation approach. This advantage becomes even more relevant considering the issues like the hot spots or dark silicon.

The detailed discussion of the placement algorithms is out of the scope of this dissertation. However, some considerations must be done. The main aspect to consider is permanent bound of a set of NoC components to a given (critical) partition, which deny any other partition from using those reserved NoC composts. This represents the most delicate aspect of any solution which is based on resource privatization instead of resource sharing. For the proposed technique, each critical $application_i$ will reserve all the NoC resources it needs to communicate. This means, for instance, that each link traversed by $application_i$'s packets will not be available to any other application — those links will be permanently reserved by $application_i$.

The direct consequence of the aspect described in the previous paragraph is the connectivity reduction of the what-is-left after each further application is placed (i.e., partition is created). Considering that the applications are placed one at a time, the second application to be placed will see the overall connectivity reduced by the placement of the first one, and so on for all the other applications. This reduction can provoke the *isolated CE* issue when a CE becomes unusable because no logic connection can be established with that CE. Furthermore, the proposed solution is sensitive to this issue as the original routing algorithm is preserved for the critical partitions. The reason to avoid adaptation of critical traffic routing algorithm, to better fit the available NoC resources, is the additional complexity that would reduce the certification potential. For non-critical traffic this is not problem, as there are safety requirements, and in fact the non-critical routing is adopted to improve the connectivity.

To limit the connectivity reduction overhead, the placement algorithm should find a placement which will minimize the number of NoC components reserved by each critical partition. It exists a direct relation between the size of a partition and the number of NoC components it will need to sustain the intra-partition communication. Thus, the placement algorithm should try to find a placement which minimizes the size of each critical region. The obvious rule of thumb to achieve the small size of critical regions is to minimize the distance between the nodes which should communicate with each other. To minimize the critical partition size can be far from being a trivial task when some issues, like for instance hot spots, are faced in a heterogeneous MPSoC. On the other hand, placement algorithms are

created to minimize the resource usage, optimizing communication speed, power, etc. considering a set of constraints. However, the final goal of the placement algorithm is to avoid isolated CE issue. The critical region minimization is a sort of necessary condition and it is for sure not sufficient condition to achieve this goal. Based on the routing algorithm used by the NoC some additional considerations can be done to optimize the placement, as described in the final part of the Section 3.3. The minimization considerations are related to critical partitions placing only, as there is only one non-critical partition and it just uses the resources which are still available after critical partitions have been mapped. However, the final goal is to avoid the isolated CEs and the critical region minimization is not a necessary condition for this goal. Thus, in case a CE is not used by non-critical partition due to its isolation, the placement algorithm can attempt a non-minimal placement for critical partitions, trying to make an isolated CE gain a logical connection.

3.2.3 On-line Monitoring: Filtering

The NoC components privatization described in Section 3.2.2 only provides temporal (and spatial) partitioning under the assumption of fault-free system. Thus, this technique represents only a partial solution as in the context of mixed criticality the partitioning must hold under the software components unexpected behavior (e.g., bug, DoS attack, etc.). The main element of the overall solution proposed in this chapter is the safety monitoring and isolation enforcement feature. This feature is implemented by a filtering module, which prevents the faulty packets from being injected inside the NoC. Thus, the filtering module allows to ensure the temporal (and spatial) partitioning between software modules even if one of these is faulty. This section describes the idea behind the proposed filtering module, while its implementation is described in Section 3.3.

In detail, the idea behind the proposed filtering approach is to exploit the knowledge of a deterministic routing algorithm used by the NoC. A *deterministic routing algorithm* is a routing in which, given a packet injected inside $node_{source}$ (N_{source}) with the destination $node_{destination}$ ($N_{destination}$), the path of that packet is given by a proper function of the only N_{source} and $N_{destination}$. I.e., considered a certain a couple (N_{source}, $N_{destination}$) the routing is deterministic if and only if a unique path is associated to this (N_{source}, $N_{destination}$) couple.

The main idea behind the isolation monitoring is to check the N_{source} and $N_{destination}$ of each packet before actually allowing the packet to enter the NoC. Given a deterministic routing algorithm, this information is sufficient to know the path of the considered packet. Thus, it will be possible to know whether the considered packet will violate the partitioning by using some NoC resources dedicated to another partition. This can be seen as a monitoring of whether, a partition is generating a traffic which can interfere with traffic of other partitions, and to discard such traffic before it actually enters the NoC. In this way the temporal (and spatial) isolation, defined during off-line application mapping, is protected against unexpected behavior like for instance bugs. This will grant the required measures to ensure a fault inside one software module cannot propagate to other software modules.

Considering a NoC consisting of a single physical network (i.e., without VCs or with VCs disabled), the following requirements must be satisfied:

1. the NoC should use a deterministic routing algorithm;
2. the routing algorithm should be known;
3. the topology of the NoC should be known.

The requirement of a deterministic routing algorithm could seem to be an important limitation, as dynamic reconfiguration solutions cannot be used. However, this is not an actual limitation as far as critical domain is considered. In fact, for the safety critical applications in general and especially for avionics, a dynamic reconfiguration is unfeasible as each possible configuration must be certified separately (and of course the sub-system in charge of re-configuring the system must be certified as well).

3.2.4 On-line Connectivity Reduction Mitigation: Redirection

To leverage the connectivity reduction, innate for any resource privatization technique, a redirection feature is granted for the non-critical traffic. This section describes the idea behind the redirection feature, while its implementation is described in Section 3.4.

The redirection feature goal is to allow the non-critical applications a better usage of the system resources. The reasons why this is a relevant issue are described

in Section 3.2.2. The main problem is that many NoC components (i.e., physical links, FIFOs, etc.) will not be available for non-critical applications as they will be reserved by some critical application. To leverage this connectivity reduction cost, we propose a technique based on traffic redirection. The traffic redirection can be simply seen as the usage of an intermediate node to re-transmit the traffic. This means that, considering an original communication scheme in which $node_A$ (N_A) is transmitting packets to N_B, in the redirected communication scheme there is an extra $N_{re-transmitter}$. Thus, in the overall transmission becomes $N_A \rightarrow N_{re-transmitter} \rightarrow N_B$. The redirection feature de facto updates the routing algorithm for the non-critical applications, despite the traffic always follows the original routing algorithm provided by the NoC (i.e., $N_A \rightarrow N_{re-transmitter}$ and $N_{re-transmitter} \rightarrow N_B$ both follow the original routing algorithm). A key point it worth to be pointed out, is that the redirection feature is to non-critical traffic only. This means that no certification is required for this technique and thus the latter does not add complexity to the solution which should undergo the certification process. Considering the performance point of view, the applications using the redirection feature experience a latency rice and also the CE in charge of re-transmitting the traffic will have a performance overhead to generate the redirection packets.

From the application placement point of view, the redirection will grant the non-critical nodes a more flexible routing algorithm. This will make easier to avoid the isolated CEs and fulfill connection constraints, which are the only goals of the redirection feature. From the traffic monitoring and filtering point of view, the redirection is completely transparent mechanism. Each traffic is checked to be allowed to be injected inside the NoC according to the path it will take inside the NoC. A redirected packet is just a normal packet if CE-to-CE path is considered, so no dedicated aspect is required from filtering module.

3.3 Filtering Module Implementation

This section will explain the implementation of filtering module described in Section 3.2.3. The proposed filtering technique is intended to be implemented as purely software module, as COTS components are targeted. The filtering module is intended to be integrated inside the a RTOS running on a NoC-based MPSoC. The redirection module, on the other hand, can be implemented at application level. This aspect

represents an improvement of the proposed solution with respect to its last version [39], where the redirection feature was embedded inside the filtering module.

The redirection module, as it only deals with non-critical traffic, can be certified at DAL D or DAL E (according to the DAL of the related software module). This will keep the certification process simple, as only filtering module must be certified at the high DAL. In detail, filtering module should have the same or higher DAL than the highest DAL of the software components this filtering module monitors partitioning (Sect. 2.4.2. of [9]).

The proposed filtering module acts as a driver for the NoC interconnect, supervising and eventually filtering software requests for the NI. As the assumption is to have a RTOS in our system, the natural position of the proposed filtering module is inside this RTOS.

From the certification point of view, after any modification the RTOS will require a new certification. However, the proposed module can be seen as a driver, filtering the traffic towards the (NoC) interconnection. This means that the certification should not reevaluate the RTOS by scratch. Furthermore, once certified, this updated RTOS will be reusable in several applications without submitting it to a new dedicated certification process in each new project. As stated in [9] the level of assurance should be selected according to the nature of the application. To be as general as possible, the module can be certified at the maximum level (DAL A), or it can be decided to undergo a much simpler certification process and go for DAL B certification. The latter choice will imply that the RTOS could not have DAL A certification, and thus must be carefully evaluated. Considering the effort for certification of a high DAL (e.g., DAL A) module, the filtering technique implementation simplicity is crucial. Furthermore, a simple implementation will grant a low performance overhead as well.

As already explained in Section 3.2.3, knowing both the routing algorithm and the topology of the network, it is known the path of each packet from the N_{source} and $N_{destination}$ of this packet. This information must be exploited, in the simplest way possible, to enforce the partitioning described in Section 3.2.2. The proposed implementation is a small software module checking each packet and allowing the packet to be injected inside the NoC if its $N_{destination}$ is allowed. In fact, as the filtering modules knows the N_{source} (i.e., N_{source} is the CE the filtering module is running on), there is a one-to-one relation between $N_{destination}$ and the

path the packet will follow once it is injected inside the NoC. The table of allowed destinations is computed offline, considering the communication paths defined during the application placement phase (see Section 3.2.2). In case an attempt of partitioning violation is detected, i.e., an attempt to inject a packet with not allowed $N_{destination}$ is detected, this packet is discarded, and this event can be signaled to the system level fault management mechanism. Thus, temporal (and spatial) partitioning is preserved as the faulty packet is denied from entering the NoC. The system level fault management mechanism will evaluate the severity of the failure and take the proper countermeasures, for instance enabling space platform and restarting the faulty one. This mechanism as well as the system level fault tolerance techniques are out of the scope of this dissertation.

To better explain the proposed filtering module, this section will consider a simple example of mixed-criticality system running on a (NoC-based) MPSoC. The NoC of the example will be a 2D 4x4 mesh topology NoC using XY routing. Although, this particular NoC architecture will be considered, the proposed approach is not limited to the 2D mesh topology, nor to the XY routing. The only limitations to the NoC, the proposed approach can be applied, are listed and explained in Section 3.2.

For what it concerns the software stack, the first example will consider the simplest mixed-criticality system, I call *mcs_1*: one critical application and one non-critical application. For the sake of simplicity, in this section I will call the former *C_app* and the latter *NC_app*. To further simplify the explanation, C_app will be using exactly two nodes, while NC_app will use all the remaining nodes.

The consideration will be done for *mcs_1* are directly extendable to the design case of one C_app and multiple NC_apps as the NC_apps all belong to the same partition (see Section 3.2.2). On the other hand, it is not easy to extend that simple example to the design case with multiple C_apps and/or with more than two nodes used by a C_app. Thus, these scenarios will be separately described further on in this section.

Fig. 3.4.a depicts a placement scenario of the mixed-criticality system we described in previous paragraph, I will call *mcs_1_a*. This scenario represents the worst-case placement, as the two critical nodes N_0 and N_{15} are at the maximum distance (according to the considered routing algorithm) from each other. Although, this is an unlikely scenario, it can still happen. In fact, an MPSoC is normally a heterogeneous system and in case it runs a realistic number of C_apps, it could

happen that the only available instances of a given resource are far one from the other. The hot-spot issue can also lead to this worst-case scenario.

In this dissertation I will call *critical domain* or *critical region* of a given C_app_i, as the NoC region traversed by the traffic of C_app_i. The critical region of C_app_1 can be seen in Fig. 3.4.a.

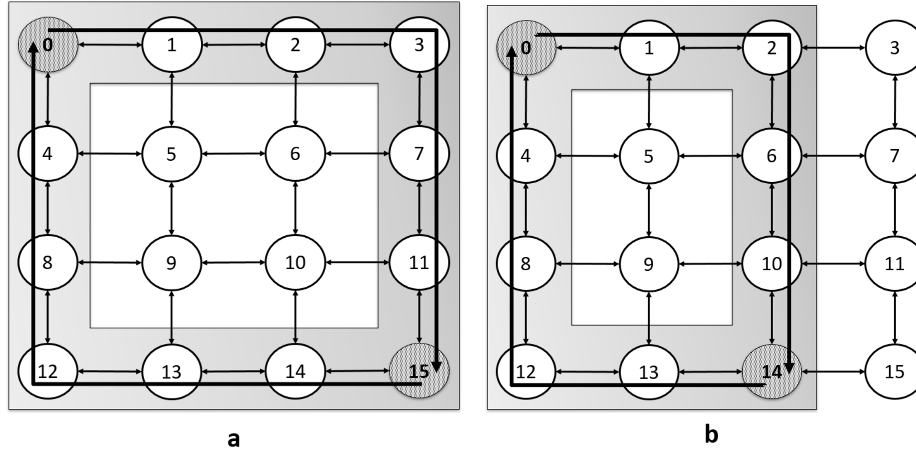

Fig. 3.4 MPSoC with 4x4 mesh topology NoC, XY routing;
mixed-criticality system example *mcs*_1: one C_app, one NC_app;
black arrows: critical traffic paths;
gray-shaded: critical regions (or domains)
a) placing scenario *mcs*_1_*a*: C_app_1 - N_0 and N_{15}, NC_app_1 - all the remaining nodes;
b) placing scenario *mcs*_1_*b*: C_app_1 - N_0 and N_{14}, NC_app_1 - all the remaining nodes;

Algorithm 3.1 describes the filtering module implementation. Table 3.1 reports the connectivity table for N_2 (scenario *mcs*_1_*a*). As can be observed, the entire filtering module implementation is just a one bit check inside a pre-computed array (1D table). In case the bit is *ALLOWED*, the packet is sent to NI to be actually injected inside the NoC. Otherwise, the packet is dropped, and the error event is generated. The proposed solution does not need to perform any on-line computation nor any synchronization with other CEs on the NoC. This means the filtering module is extremely small and simple and does not create computational nor communication overhead.

For what it concerns the connectivity tables, they only reflect the placing scenario inferred during design phase of the system (see Section 3.2.2). For a critical node, the allowed connections are created by design-time privatization of all NoC resources

Algorithm 3.1 Traffic filtering module

```
1: procedure TRAFFIC_FILTERING( dst_addr, payload )
2: //connectivity_table is internally defined
3:     if connectivity_table[dst_addr].ALLOWED then
4:         forward packet to NI
5:     else
6:         return error_code
7:     end if
8: end procedure
```

that node needs. On the other hand, the non-critical nodes do not privatize resources, thus the connectivity table reflects the remaining connectivity. This table considers the routing algorithm, the topology and the position of the nodes. A connectivity table can be inferred using the rules described in Section 3.2.2. These rules can be summarized as:

1. a node can only communicate with the nodes of the same partition;
2. a node can send a packet to another node if and only if this will not interfere with the traffic belonging to any other partition. Where, two traffic are interfering if and only if they compete for a physical link or for any internal to the router resource.

Applying these rules to the N_2 of the *mcs_1_a* (Fig. 3.4.a) yields the connectivity table described in Table 3.1. It worth to be noticed that the assumption to have two mono-directional links ensure that N_2 can communicate to N_{12} without interfering with the critical traffic. In fact, its traffic will flow in the opposite direction with respect to the critical traffic, thus no physical wire (nor internal to the router resource) will be shared between the two.

So far only the simplest example of mixed-criticality system, having only two levels of criticality (i.e., *mcs_2*, has been considered. An example a system with two C_apps and one NC_app is shown in Fig. 3.5. In both scenarios mcs_2_a and *mcs_2_b*, the two critical regions are overlapping, but still these scenarios are supported by the proposed approach. In *mcs_2_b* it can be even observed how N_5 belonging to C_app_2 is located inside the critical region of C_app_1. This example shows the flexibility of the proposed solution with respect to the strict critical region segregation approach.

Table 3.1 Placement scenario *mcs*_1_*a*: connectivity table for N_2

DESTINATION NODE	ALLOWED
0	0
1	1
3	0
4	1
5	1
6	1
7	0
8	1
9	1
10	1
11	0
12	1
13	1
14	1
15	0

From the performance perspective, the execution time of the proposed module should be first of all predictable and also sufficient small. The proposed solution meets these requirements, as the filtering module is extremely simple. The whole filtering process can be seen as consulting a bit of a small array, according to the destination of the packet and discard a packet in case that bit is 0. The filtering module is embedded inside the NoC driver managing NI access. Considering this original driver, the proposed module is added to the driver and it is the first one to be executed. The WCET of the few lines of code are easy to be upper bounded by means of static timing analysis, thus the overall WCET of the upgraded NoC driver will be just increased by this amount.

Considering the example of NoC of the *mcs*_1 and *mcs*_2, some further considerations on the placement can be done in addition to those seen in Section 3.2.2. Considering the case in which more than two nodes are used by a C_app, it is important to avoid a placing which will increase the size of the critical region. As we have seen in Section 3.2.2, this is a rule of thumb to avoid a high overhead due to connectivity reduction. In case of a 2D mesh topology NoC using XY (or YX) routing algorithm, we can observe that the critical region does not increase in size if:

- a node is added to a corner of the critical region;

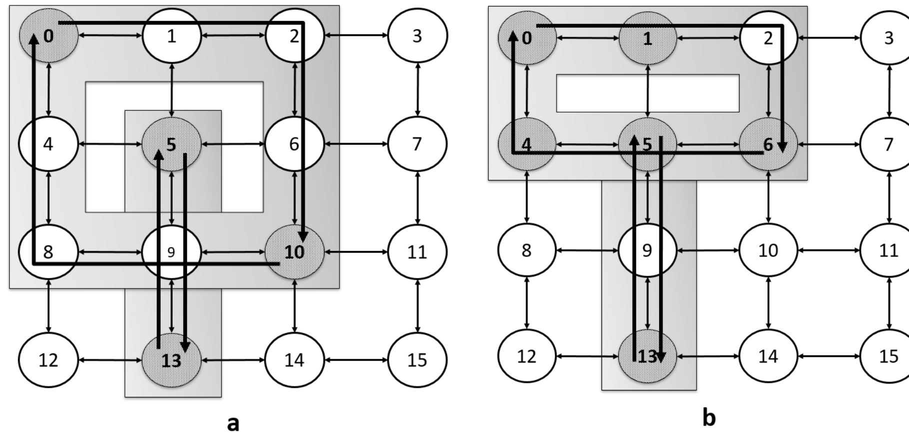

Fig. 3.5 MPSoC with 4x4 mesh topology NoC, XY routing;
mixed-criticality system example *mcs*_2: two C_apps, one NC_app;
black arrows: critical traffic paths;
gray-shaded: critical regions (or domains)
a) placing scenario *mcs_2_a*: C_app_1 - N_0 and N_{10}, C_app_2 - N_5 and N_{13}, NC_app_1 - all the remaining nodes;
b) placing scenario *mcs_2_b*: C_app_1 - N_0, N_1, N_4 and N_6, C_app_2 - N_5 and N_{13}, NC_app_1 - all the remaining nodes;

- a node is arbitrary placed inside the critical region, as far as it only communicates to the nodes located in the same side of the rectangle (critical region);
- a node is arbitrary placed inside the critical region, if there are no nodes encircled by the critical region.

For instance, considering the application using N_0 and N_{10} of the Fig. 3.5.a, more nodes can be added in the corners (N_2 and N_8) without any connectivity reduction. The nodes N_1, N_4, N_6 and N_9 can be also added without any connectivity reduction overhead, as far as they only communicate the nodes located in the same side of the rectangle. This means the, for instance N_1, should only communicate with N_0, (and N_2, if any). Fig. 3.5.b depicts another placement scenario which belong to the same example of mixed-criticality system as *mcs*_2_*a* do. In *mcs*_2_*b*, both critical regions are in condition described by the last point of the list, as their critical regions does not encircle any node. This means that the critical nodes can be arbitrary located inside the critical region without no restrictions on its connectivity, for instance N_1 can communicate with all the nodes of its critical region. These considerations must be done during the placement phase, which aim to minimize the size of the critical regions.

3.4 Redirection Module Implementation

This section will explain the implementation of redirection module described in Section 3.2.4. The proposed redirection technique is intended to be implemented as purely software module, as COTS components are targeted. Differently from the filtering module (Section 3.3), there is no reason to implement it at RTOS level. Thus, the redirection module can be implemented at application level, avoiding addition complexity to the certification of the overall solution. This aspect represents an improvement of the proposed solution with respect to the original version [39], where the redirection feature was embedded inside the filtering module.

The redirection module, as it only deals with non-critical traffic, can be certified at DAL D or DAL E (according to the DAL of the related software module). This will keep the certification process of the overall solution much simpler, as only filtering module must be certified at the high DAL.

Considering *mcs_1_a* scenario, from Table 3.1, it can be observed that N_2 (without the redirection feature) is not able to communicate to three other non-critical nodes. This is due to limitations imposed by its position within the critical region and the XY routing. This problem can be solved by adding a redirection functionality to the non-critical applications. This redirection functionality is implemented by using a reachable node as a re-transmitter ($N_{re-transmitter}$). Of course, $N_{re-transmitter}$ should be able to reach the final destination (eventually using the re-transmission itself).

The redirection module is implemented at application level. It is composed of transmission and receiving parts which are executed whenever a packet must be transmitted, or it is received by any NC_app accordingly.

The transmission part of redirection module is presented in Algorithm 3.2. Table 3.2 reports the redirection table for N_2 (scenario *mcs_1_a*). From the algorithm it can be seen how the first check somehow mimics the filtering module by checking whether the desired destination node is allowed. This is indeed a redundant task, which however is implemented at application level and thus can have a different purpose. The actual redirection is implemented by generating the redirection packet, which can either request or not the actual redirection (i.e., to make use of $N_{re-transmission}$). In detail, if the re-transmission is required, the address of node in charge of re-transmitting the packet becomes new destination address. A *REDIR* tag is added to the payload, followed by the actual destination address. In case $N_{destination}$

can be reached without any actual redirection, the *NO_REDIR* tag is added to the payload. Once the redirection packet is generated, it is normally transmitted. This means that it's payload and destination address will be received by the NoC driver and processed by the filtering module described in the previous section.

Algorithm 3.2 redirection module: transmission part.

```
1: procedure TRAFFIC_REDIRECTION_TX( dst_addr, payload )
2: //redirection_table is internally defined
3:     if redirection_table[dst_addr].ALLOWED then
4:         if redirection_table[dst_addr].RDR > −1 then
5:             //generate redirection packet
6:             new_dst_addr = redirection_table[dst_addr].RDR
7:             new_payload = (REDIR, dst_addr, payload)
8:         else
9:             //generate redirection packet
10:            new_dst_addr = dst_addr
11:            new_payload = (NO_REDIR, payload)
12:        end if
13:        transmit_packet(new_dst_addr, new_payload)
14:    else
15:        return error_code
16:    end if
17: end procedure
```

On the other hand, when a NC_app receives a packet, it executes the reception part of the redirection module, as described by Algorithm 3.3. As can be observed, the tag placed by Algorithm 3.2 is checked and the packet is processed accordingly, and it is either re-transmitted or forwarded to the application software.

Algorithm 3.3 redirection module: reception part.

```
1: procedure TRAFFIC_REDIRECTION_RX( payload )
2:     if payload.tag = REDIR then
3:         actual_dst_addr = payload.actual_dst_addr
4:         actual_payload = payload.actual_payload
5:         RAFFIC_REDIRECTION_TX(actual_dst_addr, actual_payload)
6:     else
7:         actual_payload = payload.actual_payload
8:         forward actual_payload to application software
9:     end if
10: end procedure
```

Table 3.2 Placement scenario mcs_1_a: redirection table for N_2

DESTINATION NODE	ALLOWED	REDIRECTION NODE
0	0	X
1	1	-1
3	1	6
4	1	-1
5	1	-1
6	1	-1
7	1	6
8	1	-1
9	1	-1
10	1	-1
11	1	10
12	1	-1
13	1	-1
14	1	-1
15	0	X

The *redirection table* is computed for each node and describes whether a node is reachable and in case the redirection is needed, which is the address of $N_{re-transmitter}$ the traffic must be redirected to. The redirection table is computed off-line, based on the same information used to compute the connectivity table. The connectivity tables themselves can be used, to infer the redirection tables, by creating paths between each couple of non-critical nodes. Several well-known approaches to compute such information does exist, thus techniques are considered out of the scope of this dissertation. It is important to highlight how, the redirection table computation, does not require the certification effort to be as high as the one required by the connectivity table computation. The redirected traffic will be considered as a normal traffic by the filtering module, thus a low DAL will no way affect the temporal (and spatial) solution described in the previous section.

The advantage of redirection feature can be observed by comparing the ALLOWED columns of redirection and filtering tables. For the considered example, comparing Table 3.2 and Table 3.1, it can be observed how the N_2 is able to reach all the nodes, apart those which belong to the C_app. In case the redirection feature would not be implemented, N_2 would not be able to reach N_3, N_7 and N_{11}.

3.5 Experimental Evaluation

Experimental evaluation has been performed using a simulation environment, considering the VHDL implementation of the NoC model. The proposed solution has been validated by considering a faulty partition generating high volumes of random traffic.

The logic-based distributed routing (LBDR) [41] architecture was used to implement NoC model described in Section 3.2.1. The overall NoC implementation is a simplified version of the NoC architecture presented in [38]. The LDBR router is configured to implement an XY routing algorithm, which is a fairly common, deadlock-free routing algorithm. Section 3.1 presents the reasons for the choice of such simple architecture as well as the considerations on the applicability of the proposed solution for different and more complex NoC models.

The experiments have been done considering the *mcs*_1_*a* scenario described in Fig. 3.4.a. Two simulation campaigns have been performed, considering the baseline system setup (with no temporal partitioning implemented) and the system implementing the proposed solution. In both cases, to evaluate the effects of temporal interference, the NC_app is implemented to have a faulty behavior. In detail, each node of NC_app randomly selects destination of the packets it injects inside the NoC. It worth to mention that the proposed solution only discriminates the NC_app from C_app for what it concerns the redirection feature to provide to the former. I.e., the proposed solution creates temporal (and spatial) isolation between any partition, even those having the same DAL. This means that, considering a more complex scenario (e.g., *mcs*_2_*b*) would add nothing to the proposed solution validation.

Experiments were performed considering several packet injection rates (PIRs), defined as the number of packets injected in the NoC at each clock cycle by any given node. For each considered PIR, 10,000 packets were injected by each node in the NoC.

The experimental results, related to baseline setup evaluation, are presented in Table 3.3. The figures exhibit the uncertainty in the communication latency among the two critical nodes (N0 and N15), as the non-critical traffic introduces congestion on the critical traffic path.

Once implemented, the proposed filtering module prevents the non-critical traffic to interfere with the critical traffic. This traffic isolation has the effect to cancel

Table 3.3 Critical traffic latency (ns): baseline setup.

PIR	MAX	MEAN	STD. DEV.
0.020	815	334.8	55.6
0.010	665	315.1	30.4
0.007	655	311.6	24.4

any variance in the critical communication. This is experimentally confirmed by evaluating the setup where the proposed solution has been implemented. The figures related to the system implementing the proposed solution are presented in the Table 3.3. As can be observed, the latency for the critical traffic to traverse the NoC becomes deterministic as it is always 305 ns. The proposed filtering solution allows to enforce the temporal partitioning between software components (partitions). This is crucial requirement to compute WCET by static timing analysis method.

Table 3.4 Critical traffic latency (ns): proposed solution implemented.

PIR	MAX	MEAN	STD. DEV.
0.020	305	305	0
0.010	305	305	0
0.007	305	305	0

The proposed solution, without the redirection feature, has the side effect of significantly reduce the logical connectivity for the NC_apps. The reasons of such reduction have been already explained in Section 3.2.4.

In the scope of this dissertation I will define *reachability* metric of N_i as the number of nodes to which N_i is able to send a packet. This metric must not be confused with the *connectivity* metric, which is usually defined as number of nodes to which a given node is physically connected. In fact, the existence of a physical connection between the two nodes is a necessary condition for these nodes to able to send a packet to each other, but it is not a sufficient condition. In fact, a further requirement to allow the traffic flow is the routing algorithm to allow the logical connection for the given physical connection existing between those nodes.

The reachability metric has been introduced to measure the logical connectivity improvement given by the redirection feature. This metric will be only considered for NC_apps, as C_apps already have all the required connectivity (see Section 3.2.2) and in any cannot use redirection feature. On the other hand, NC_apps are BE QoS level and thus can benefit from the extra connectivity given by the redirection feature.

Reachability figures of the setup which does not implement the redirection feature, related to *mcs*_1_*a* scenario, are reported in Table 3.5. The average reachability obtained in this case is 10.7, while the maximum reachability for each node is 13 (it cannot communicate with the two critical nodes and with itself). This means that, on average, each node cannot send packets to 1 other node. The redirection extension is implemented with the purpose of improving the reachability for NC_apps, in order to have a more usable system. Implementation of the redirection feature, allows to obtain the perfect reachability, rising the average reachability to 13. The cost of this reachability improvement is some additional communication latency for non-critical nodes, due to the intervention of the software module during transmission. However, this cost does not affect in no way the C_apps.

Table 3.5 Reachability without redirection for *mcs*_1_*a* scenario (non-critical nodes only).

NODE ID	UNREACHABLE NODES	REACHABILITY
1	2, 3, 6, 7, 10, 11, 14	6
2	3, 7, 11	10
3	7, 11	11
4	11	12
5	11	12
6	11	12
7	11	12
8	4	12
9	4	12
10	4	12
11	4	12
12	4, 8	11
13	4, 8, 12	10
14	1, 4, 5, 8, 9, 12, 13	6
AVERAGE	**N/A**	**10.7**

Considering the placement scenarios *mcs*_1_*b* (Fig. 3.4), *mcs*_2_*a* and *mcs*_2_*b* (Fig. 3.5), the redirection module allows to have the maximum reachability as well.

Although, the redirecting feature provides a huge boost in logic connectivity, it is still possible to have a node completely cut-out from the NoC. Fig. 3.6 presents a system with a further C_app with respect to the *mcs*_2_*a*. For this placing scenario, the presence of communication between N_1 and N_0 makes N_2 de facto isolated from the NoC. I.e., N_2 is not able to communicate to no one of the nodes. It worth to be

noticed that in case there were no communication between N_1 and N_0, N_2 would not be isolated.

The considered examples of placement, especially the last one, show the limit of the proposed solution. When the number of C_apps grows, some nodes can be prevented from communicate with each other; some node can be even completely isolated from the other nodes. In order to avoid this connectivity reduction overhead, there is a rule of thumb to minimize the size of the critical regions. However, as Fig. 3.6 shows, this is just a rule of thumb. In the real system, there is a set of applications to be mapped on the MPSoC, each with some resource requirements. On the other hand, the MPSoC has a set of resources and the premise is that these resources are sufficient to host the applications without the proposed approach. The applications must be mapped on the available resources during the design phase. At this phase, the placement algorithm should first of all find a placement for all the applications as described in Section 3.2.2. During this process, the placement algorithm will try to find a placement which would avoid reachability reduction and node isolation.

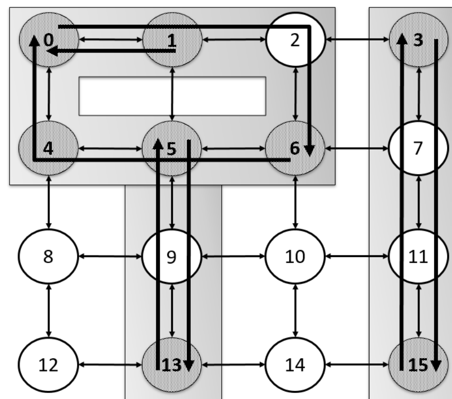

Fig. 3.6 MPSoC with 4x4 mesh topology NoC, XY routing;
mixed-criticality system example *mcs*_3: three C_apps, one NC_app;
black arrows: critical traffic paths;
gray-shaded: critical regions (or domains)
placing scenario *mcs_3_a*: C_app_1 - N_0 and N_{10}, C_app_2 - N_5 and N_{13}, C_app_3 - N_3 and N_{15}, NC_app_1 - all the remaining nodes;

3.6 Summary

Considering COTS MPSoC, this chapter proposes the main contribution of this dissertation: a solution to face the temporal isolation issues related to implicit contention of NoC interconnection, mainly targeting avionic field. The proposed solution is applicable to any NoC with a deterministic routing as far as the topology and the routing algorithm are known.

The proposed solution is developed targeting simple NoC architectures. As already explained, this choice allows to keep certification costs low. It is worth to highlight how a more complex architecture is perfectly suitable to adopt the proposed solution; the problem is not the proposed solution but instead that architecture itself, as its complexity will be a huge issue from the certification point of view. For instance, in case an architecture featuring VCs mechanism is considered, our solution is perfectly usable by simply not using the virtual networks.

In the scope of this dissertation the proposed solution will be explained for a specific NoC architecture, although the proposed solution can be applied to any NoC using a deterministic routing as well as having known routing algorithm and topology.

The proposed solution is a partitioning technique based on resource privatization, thus there is no resource sharing between software components (i.e., applications). At the applications placing phase, NoC is seen as a set of resources (i.e., links, FIFOs, etc.) to be partitioned between the applications. Each application has a dedicated partition, formed by the resources used by that application. In order to exhibit the lowest complexity possible, the proposed solution considers a scenario where no inter-partition resource sharing exist. This means that each application is using the NoC without traffic to interfere with the traffic of any other application. The core of the proposed partitioning technique is the monitoring and enforcement feature. This feature is implemented by a (software) filtering module, which prevents the faulty packets from being injected inside the NoC. Thus, the filtering module allows to ensure the temporal partitioning between software modules even if one of these is faulty. The filtering module is implemented as purely software module to be inserted inside the RTOS. The overall solution is deliberately simple to exhibit a high certification potential.

From the QoS point of view the proposed techniques offer two levels of service: guaranteed service (GS) and best effort (BE). An application running assigned with GS level is guaranteed to have bounded latency and throughput. On the other hand, applications running under BE level have no guarantees. The proposed techniques support an unbounded number of criticality levels, as each application running under the GS level is granted with the temporal isolation.

Furthermore, this chapter presents a technique to reduce the communication reduction overhead provoked by the proposed partitioning solution. This technique implements the redirection feature at application level and allows to restore a perfect connectivity between non-critical nodes by sacrificing some of the communication bandwidth and performance power.

Chapter 4

QoSinNoC: hardware-implemented NoC partitioning for mixed criticality

This chapter describes the second contribution of this dissertation: (i) a set of *specific hardware —related techniques* to allow the usage of the related NoC architectures in the context of mixed criticality, and (ii) QoSinNoC framework, which allows an easy comparison between these NoC architectures for a given set of software components to be executed.

The presented techniques, each related to a simple NoC architecture, are derived to address the software components temporal isolation issue. Where the latter is the main issue related to the usage of MPSoC in the context of mixed criticality. Thus, in this chapter these techniques will be sometime referred to as *mixed criticality enforcement techniques*. The proposed techniques are based on NoC resource privatization (thus sharing avoidance) paradigm. The overall solution is based on a one-to-one bound of each NoC internal resource to a software module (i.e., application). Thus, the NoC is partitioned between the applications and the resources are prioritized to avoid any traffic interference. Furthermore, the solution does not allow any inter-application resource sharing. In this way, the temporal (and spatial) partitioning between software modules is achieved by denying any NoC inter-partition interaction.

An important part of the overall contribution (described in this chapter) is QoSinNoC framework, which facilitates the comparison between the considered NoC architectures. QoSinNoC framework can be seen as the union of: (i) a set of

considered NoC architectures, (ii) the related techniques to allow their usage in the context of mixed criticality and (iii) a module in charge of computing a set of figures of merit in order to compare which architecture will better fit the implementation of a given set of applications.

The considered NoC architectures as well as the related mixed criticality — enforcement techniques are deliberately as simple as possible. This approach is adopted to make the overall solution be simple, thus, to exhibit a high certification potential. The proposed solution also implements spatial partition between critical applications. However, the focus put on the temporal domain as the spatial partitioning is much easier to achieve and does not represent an open issue. Some of the security issues, for instance DoS attack, are solved by the proposed solution as well. However, the security aspects are out of the scope of this dissertation and will not be considered. This partitioning scheme is enforced by physically isolating each critical partition (i.e., by making impossible any NoC usage that would break the partitioning scheme). This actual isolation is physically implemented by hardware means which are specific to each of the considered NoC architectures. The proposed approach supports an arbitrary high number of levels of criticality as each critical software component is granted with temporal (and spatial) isolation. The proposed approach allows to overcome a strict domain-based partitioning by performing NoC resource reservation at router level, instead of node level. The latter represents an advantage of the state-of-the-art solutions.

The techniques to allow the mixed criticality usage, used by QoSinNoC, are not limited to the particular kind of NoC architecture they were created for. On the contrary, these techniques can be applied for whatever COTS MPSoC as far as the latter presents the key characteristics for their applicability. I.e., the NoC architecture used by that COTS MPSoC should present the similarities with the NoC the technique was developed for (i.e., the NoC considered in QoSinNoC).

The QoSinNoC frameworks as well as the first implementations of the proposed privatization-based partitioning was first presented in [101]. The proposed techniques and the QoSinNoC framework itself were then improved in [40].

The rest of this chapter is structured as follows. Section 4.1 presents the state-of-the-art solutions to face the NoC temporal partitioning issue, focusing on hardware-implemented solutions; Section 4.2 and Section 4.3 presents the proposed partitioning solution. Section 4.2 explains the general technique used to achieve temporal isola-

tion as well as the NoC model such technique can be applied to. Whereas, in Section 4.3 the implementation specific details of the proposed techniques are explained for the relative NoC architectures. Section 4.4 presents QoSinNoc framework, created to facilitate the comparison between the considered NoC architectures for a specific set of applications to be implemented. Section 4.5 presents the experimental evaluation of the proposed solution; finally, Section 4.6 summarizes the key aspects of the proposed solution;

4.1 Custom NoC–based MCS: State-of-the-art

To cope with the temporal isolation issue in MPSoC, the literature provides several solutions. Some of these solutions are adopted from bus-based multi-core related techniques, while the other have been created specifically for NoC-based MPSoC. The main issue with these solutions is certification potential, as them only indirectly consider the industrialist's perspective.

This section will provide an extended analysis of the state-of-the-art related to the usage of custom NoC architectures in the scope of critical systems.

The techniques targeting the usage of COTS NoC design are also relevant in the scope of this chapter, as a custom solution can implement some of the features of a COTS design. Those techniques have been already analyzed in Section 4.1, in the scope of the main contribution of this thesis, thus will not be repeated here.

This section presents the state-of-the art solutions concerning the time isolation issue in the context of custom NoC architectures. Where the term *custom NoC architecture* is used as the opposite to *COTS NoC architecture* (used by COTS MPSoC).

Considering the techniques relying on custom NoC architecture design, I will use the same classification used for techniques related to COTS MPSoC.

QNoC [26] was one of the first efforts to design a NoC architecture suitable for the creation of an MCS. The proposed architecture was based on preemptive priority scheduling. The architecture makes use of wormhole packet-based round-robin scheduling which is used by the NoC architectures of many modern MPSoC. Several similar solutions, also based on the concept of priority based preemptive packet-based usage of the NoC were derived [27–33]. Each of these solutions is

characterized by some peculiarities, but all make use of virtual channels, which is their main disadvantage when certification potential is considered.

One of the first solutions was also Æthereal [34] architecture implementing time-division multiplexing (TDM)-based circuit switching approach, where wires and buffers remain reserved for certain points in time. The resources are reserved for critical applications, while the non-critical applications use leftover bandwidth from the critical applications. This solution however presents several problems like the difficulty to support communication between distant nodes and also present not negligible complexity.

The authors in [35] present a NI modification to allow TT-based scheduling of the NoC. This solution grants a bounded latency on high-criticality traffic by using a contention-free channel. However, to support the proposed approach the overall system should present a high level of complexity.

A different approach to the problem is *link division multiplexing technique* [36]. In this solution, each physical link is partitioned to simultaneously transmit serialized packets belonging to distinct traffic flows. The idea of this solution is appealing; however, it presents a high complexity from both hardware and conceptual point of view.

Apart from the (NoC) resource sharing technique presented so far, there are some solutions based on avoiding resource sharing. In this macro-category, the NoC is not seen as a monolithic resource, instead it is considered as a set of resources (i.e., links, FIFOs, etc.) potentially independent from each other.

The authors in [37] use hardware-enforced segregation between a safety-critical domain and a non-safety-critical domain. The main limit of this solution, as it was for [24], is the high cost required to connect two distant nodes. The authors in [38] present a NoCDepend method to allow communication between critical regions, implemented by a set of nodes used as input and output gateways. Thus, this feature can solve the issue of connecting two (or more) distant nodes. However, this solution presents a certain complexity which can be a prohibitive for the certification cost. In addition, the authors in [38] present a dynamic reconfiguration technique. This can be used as an alternative to the inter critical region communication, however this solution presents the same disadvantages of circuit-switching—based solutions.

From the state-of-the-art analysis it can derived the list of issues of the existing approaches in dealing with temporal isolation issue. The main issue is the complexity of the solution, which has the direct impact on the certification feasibility. On the other hand, the low-complexity solutions are not complete as a set of issues is only cursorily and indirectly addressed. Finally, the solutions that seem to be complete and sufficiently simple are presenting a high implementation cost. The solution proposed in this chapter aims to be a low-complexity one, still addressing the main issues of the temporal partitioning. Considered the state-of-the-art, the proposed solution is close to the one presented for Tilera Tile processor [24]. However, the similarity only refers to the resource privatization approach. Differently from the [24], my solution allows a much flexible NoC usage as it overcomes the strict domain-based partitioning.

4.2 Proposed Partitioning Solution

Considering a set of NoC architectures, a solution is derived to face the temporal isolation issues related to implicit contention of NoC interconnection, targeting critical domain. The proposed solution is a partitioning technique based on resource privatization, thus there is no resource sharing between critical software components (i.e., applications). The overall solution is implemented in hardware, although this hardware is supposed to be configured (by software at bootstrap time). The proposed solution is implemented entirely at system design time. In detail, critical applications are placed is such a way to avoid any contention for NoC internal resources (from the critical applications). Thus, the temporal isolation is achieved for the critical applications. In order to enforce this temporal isolation, thus prevent that a failure can propagate from an application to another, each proposed technique exploits the hardware characteristics of the NoC architecture it is used on. Thus, the partitions are physically isolated from each other.

4.2.1 Considered NoC Model

The proposed solution is developed targeting simple NoC architectures. As already explained, this choice allows to keep certification costs low. It is worth to highlight how a more complex architecture is perfectly suitable to adopt the proposed solution;

the problem is not the proposed solution but instead that architecture itself, as its complexity will be a huge issue from the certification point of view.

In the scope of this chapter the proposed solutions will be explained and validated for a specific set of NoC architectures, although the proposed solution can be applied to any NoC presenting the same characteristics. All the NoC architectures are a 2D mesh topology, as can be seen in Fig. 4.1. The figure also shows the path taken by a packet send by N_0 towards N_5, according to XY routing algorithm.

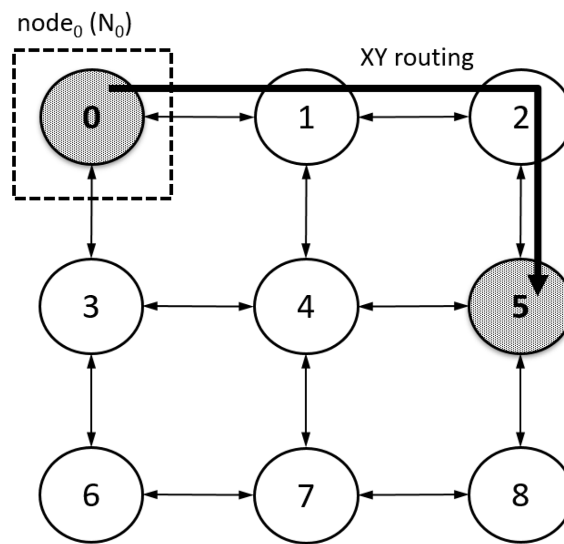

Fig. 4.1 A schematic view of a 3x3 (2D) mesh topology NoC; N_0 to N_5 packet path under XY routing algorithm.

The considered NoC model allows to implement two routing algorithms for the critical traffic. For the non-critical traffic, deadlock-free adaptive routing algorithms will be used. However, in the scope of the proposed approach, there is no constrain on the routing algorithm used for non-critical traffic, thus whatever algorithm can be used by the latter. The first routing algorithm used for critical traffic (rout_alg_A) is presented in in Fig. 4.2.a. This XY and YX routing this approach will allow to achieve minimal path routing while still prevent any deadlock since only two nodes are considered in this case. The second routing algorithm used for critical traffic (rout_alg_B) is the normal XY routing and it is presented in in Fig. 4.2.b. The rout_alg_A allows to minimize the critical region extension. On the other hand, rout_alg_B generally has a lower cost in terms of connectivity reduction.

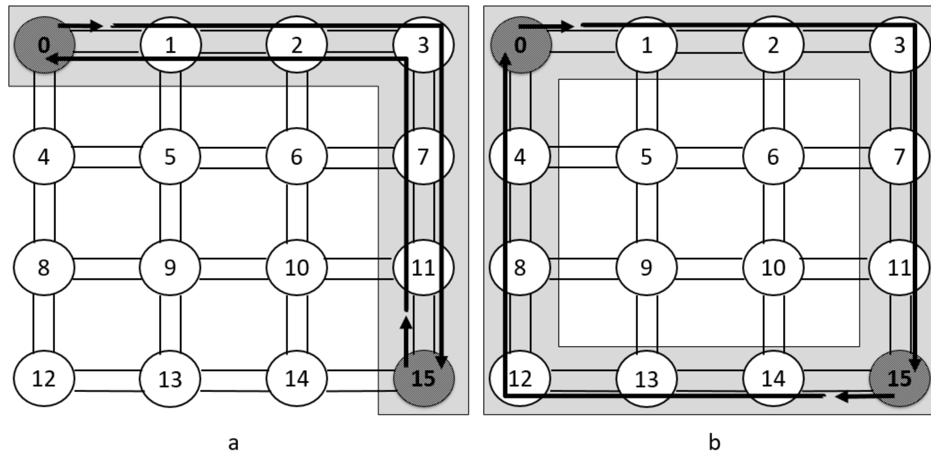

Fig. 4.2 MPSoC with 4x4 mesh topology NoC;
mixed-criticality system example: one C_app, one NC_app;
black arrows: critical traffic paths;
gray-shaded: critical regions (or domains)
a) placing scenario *mcs*_1_*a*: C_app_1 - N_0 and N_{15}, NC_app_1 - all the remaining nodes, rout_alg_A is used for C_app_1;
b) placing scenario *mcs*_1_*b*: C_app_1 - N_0 and N_{15}, NC_app_1 - all the remaining nodes, rout_alg_B is used for C_app_1;

All the considered NoC designs use wormhole switching and input-buffered ports. As mentioned in Section 2.1.1 these architectural characteristics are the most commonly used. Finally, all the considered NoC architectures are based on a single physical network (i.e., not featuring VCs mechanism) implemented by links made of two physical channels, one per direction. This characteristic falls under the simplicity requirements discussed shortly above (in the previous paragraph). In case an architecture, featuring VCs mechanism, is considered, our solution is perfectly usable by simply not using the virtual networks. More information on the NoC implementation choices is present in Section 2.1.1.

Fig. 2.2, provides a detailed view of the NoC, showing how each node is made by a *connected element* (CE) connected to its dedicated *network interface* (NI) which is in turn connected to its *router* through two physical channels (one per direction). Fig. 2.3 depicts router architecture of the considered NoC design. More information on the NoC building blocks and the router implementation is present in Section 2.1.1.

4.2.2 Application Placement Phase

The partitioning is done during the applications placing phase. During this phase, for each application, it is decided which MPSoC resources (e.g., processing elements, peripherals, memory) will be used by that application. The NoC interconnection itself is seen as a set of resources (i.e., links, FIFOs, etc.) to be partitioned between the applications. Each critical application has a dedicated partition, formed by the resources dedicated to that application. In order to exhibit the lowest complexity possible, the proposed solution considers a scenario where no inter-partition resource sharing exist for critical partitions. This means that each critical application is using the NoC without its traffic to be able to interfere with the traffic of any other application. Thus, concerning the critical applications, the temporal partitioning is done not only between applications having different levels of criticality but also between those having the same criticality. For what it concerns the non-critical applications, these do not have a per-application dedicated partition. This means that it does exist a macro partition made of all non-critical applications. Thus, the non-critical applications share the MPSoC resources (not dedicated to any critical application). The proposed solution does not impose any resource sharing policy for non-critical applications; thus, any resource sharing policy can be implemented for the latter.

As can be implied from the aforementioned constrains and definitions, from the QoS point of view, the proposed solution implements two levels of service: GS and BE. Critical software modules are granted with GS level, and are guaranteed latency and throughput as no NoC contention do exist. Non-critical applications exhibit BE level, as only the guarantees originally provided by the MPSoC (which are usually best-effort) are in place. The proposed techniques support an unbounded number of criticality levels, as each application running under the GS level is granted with the temporal isolation.

Since there is no resource sharing between critical partitions, the proposed approach implements the spatial isolation (in the simplest way possible). However, the focus put on the temporal domain as the spatial partitioning is much easier to achieve and does not represent an open issue.

The proposed partitioning technique overcomes the classic *strict domain (o partition) segregation* scheme (e.g., the one proposed in [24]), for instance, partitions

can overlap (without violating traffic isolation). The main aspect is that a router can be used in parallel (and simultaneously) by different software components (thus different partitions) as far as there is no contention of the router's internal resources. This means that a router, as the NoC itself, is shared by several software components (even having different levels of criticality). However, de facto there is no sharing as each component of the NoC is privatized by one and only one partition (i.e., software component). This is a strong point of the proposed solution as it exhibits a much higher flexibility and allows communication between distant nodes without reducing the overall NoC connectivity. Considering a heterogeneous MPSoC, this flexibility characteristic allows a much better system utilization with respect to the strict partition segregation approach. This advantage becomes even more relevant considering the issues like the hot spots or dark silicon.

The detailed discussion of the placement algorithms is out of the scope of this dissertation. However, some considerations must be done. The main aspect to consider is permanent bound of a set of NoC components to a given (critical) partition, which deny any other partition from using those reserved NoC composts. This represent the most delicate aspect of any solution which is based on resource privatization instead of resource sharing. The direct consequence of this aspect is the connectivity reduction of the what-is-left after a partition is placed. This reduction can provoke the *isolated CE* issue when a CE becomes unusable because no logic connection can be established with that CE. To cope with this issue, the proposed solution allows two routing algorithms (see Section 4.2.1) to be used for a critical partition (for each critical partition one of the two is chosen during the application placement phase).

To limit the connectivity reduction overhead, the placement algorithm should find a placement which will minimize the number of NoC components reserved by each critical partition. It exists a direct relation between the size of a partition and the number of NoC components it will need to sustain the intra-partition communication. Thus, the placement algorithm should try to find a placement which minimizes the size of each critical region. The obvious rule of thumb to achieve the small size of critical regions is to minimize the distance between the nodes which should communicate with each other. To minimize the critical partition size can be far from being a trivial task when some issues, like for instance hot spots, are faced in a heterogeneous MPSoC. On the other hand, placement algorithms are created to minimize the resource usage, optimizing communication speed, power,

etc. considering a set of constraints. However, the final goal of the placement algorithm is to avoid isolated CE issue. The critical region minimization is a sort of necessary condition and it is for sure not sufficient condition to achieve this goal. The minimization considerations are related to critical partitions placing only, as there is only one non-critical partition and it just uses the resources which are still available after critical partitions have been mapped. However, the final goal is to avoid the isolated CEs and the critical region minimization is not a necessary condition for this goal. Thus, in case a CE is not used by non-critical partition due to its isolation, the placement algorithm can attempt a non-minimal placement for critical partitions, trying to make an isolated CE gain a logical connection.

4.2.3 Partitioning Enforcement

The NoC components privatization described in Section 4.2 only provides temporal (and spatial) partitioning under the assumption of fault-free system. Thus, this technique represents only a partial solution as in the context of mixed criticality the partitioning must hold under the software components unexpected behavior (e.g., bug, DoS attack, etc.). The main element of the overall solution proposed in this chapter is the isolation enforcement feature. This feature is implemented by differently for each of the considered architectures and thus will be described in the relative section (Section 4.3).

4.3 NoC Architectures and Partitioning Solution Implementation

This section will present the considered NoC architectures and the relative partitioning enforcement technique. All the considered architectures match the NoC model described in Section 4.2.1, thus only the particularities of each architecture will be presented in this section.

4.3.1 Baseline LBDR

The first considered architecture is the baseline LBDR (B_LBDR) [41], a scalable mechanism for implementing routing algorithms in for 2D architectures. LBDR depends on two fixed sets of bits: connectivity and routing bits. The former describes the underlying topology, whereas the latter defines the allowed/disallowed turns dictated by the routing algorithm.

Connectivity bits describe whether a physical link (a bit for each direction) is active or not. Thus, by un-setting these bits it is possible to create a set of completely isolated domains inside the NoC. Considering the mcs_1_a (Fig. 4.2.a) and the router model as in Fig. 4.1, to implement the partitioning enforcement, the following bits has to be un-set: the S channel connectivity bits of N_0, N_1 and N_2 as well as the W channel connectivity bits of N_7, N_{11} and N_{15}.

Regarding the routing algorithm used for the critical region, it must be chosen among the two described in Section 4.2.1. Although rout_alg_A creates a smaller critical region, the NoC resources it occupies are exactly the same compare to the rout_alg_B (as can be observed in Fig. 4.2). Furthermore, rout_alg_A is more likely to create *dead nodes*, i.e., nodes which cannot use the NoC because all the router resources are reserved by another partition. In other words, dead nodes are logically disconnected from the NoC. An example of dead node is N_3 of mcs_1_a.

The B_LBDR architecture is considered as a baseline architecture, having almost no means to implement the partitioning enforcement. Due to this characteristic, for B_LBDR there is a further cause for dead nodes. Each node located inside a critical partition, which does not belong to that partition, must be powered off. In fact, B_LBDR has no means to deny a node the usage of each and every resource of its router.

4.3.2 Extended LBDR

The second considered architecture is the extended LBDR (E_LBDR), which has been created to overcome the limitations of the previous architecture. The extended routing mechanism is implemented by means of some additional bits to flank the 5 connectivity bits which codify the connection of each router to the N, E, W, S and L channels (already present in B_LBDR). These extra 20 bits denote the

allowed/restricted turns at each router. E_LBDR not only has extra bits to allow a fine-grained control of the CE to the N, E, W and S channels, instead the turn model implementation for the E_LBDR differs as well. B_LBDR considers the turns at one hop away for setting the values of routing bits (8 routing bits per router). For E_LBDR there are the 20 routing bits per router to encode the turns at the current router (from an input port to an output port, including L output port), thus, the 20 bits allow codifying the 90-degree turns (i.e. N2E, N2W, E2N, etc.), the straight paths (i.e. N2S, S2N, etc.) and the paths from/to L port (i.e. N2L, L2N, etc.).

This architecture allows to solve the dead nodes created by B_LBDR because of location inside critical region. Thus, a non-critical node can now be located inside the critical region which allows to overcome the strict partitioning and thus achieve a much better usage of the NoC.

It must be highlighted how the additional bits of E_LBDR are not able to address the dead nodes generated because of all resources of their router have been reserved. Thus N_3 of mcs_1_a will be a dead node also in case E_LBDR is used.

4.3.3 Routing-table Based

In this architecture, for each input port of the router (N, E, W, S and L) a routing table is implemented, which stores the candidate output direction(s) for each destination in the network. Therefore, each table for each input must have 4 entries showing whether the flit should be forwarded to N, E, W or S direction. For the L port, it is handled separately in the routing logic; The information regarding routing table needs to be stored for each destination in the table of the same input. The region separation is implemented by just non supporting any which would violate the partitioning. As the stored bits in routing tables translate into memory elements, it does not scale well with growing network size and number of nodes [102]. However, routing tables allow implementation of any routing algorithm (also non minimal) in any topology.

4.4 QoSinNoC

QoSinNoc is an important part of the overall contribution described in this Chapter. The goal of the framework is to facilitates the comparison between the considered

NoC architectures for a specific set of applications to be implemented. Given a set of applications to be mapped, a set of requirements on the application mapping constrains the goal of the framework is to infer the best setup for the considered architectures and to collect some figures of merit.

Reachability is one of the most important metrics for the partitioning techniques based on resource privatization. In the scope of this dissertation I will define *reachability* metric of N_i as the number of nodes to which N_i is able to send a packet. This metric must not be confused with the *connectivity* metric, which is usually defined as number of nodes to which a given node is physically connected. In fact, the existence of a physical connection between the two nodes is a necessary condition for these nodes to able to send a packet to each other, but it is not a sufficient condition. In fact, a further requirement to allow the traffic flow is the routing algorithm to allow the logical connection for the given physical connection existing between those nodes.

The reachability metric has been introduced to measure the logical connectivity improvement given by the redirection feature. This metric will be only considered for NC_apps, as C_apps already have all the required connectivity (see Section 3.2.2) and in any cannot use redirection feature. On the other hand, NC_apps are BE QoS level and thus can benefit from the extra connectivity given by the redirection feature.

In order to exploit potentialities of LBDR-based and Table-based architectures, QoSinNoC framework optimizes the reachability for the particular considered placement scenario. The existing constrains on the nodes that should be able to communicate with each other are also considered by the framework.

Proof of deadlock-freeness is also provided in [103], which is based on the concept of proving the absence of cycles in routing graph. By traversing the routing graph, it is possible to find the paths from each source to destination. Both in the case of using E_LBDR and also Table-based, since it is assured that no cycles exist in the routing graph corresponding to the chosen turn model, therefore, even non-minimal paths would not lead to any deadlock.

The total power consumption (in mW) consists of static and dynamic power, where the latter consists of internal and switching power. The power results are obtained via Synopsys Design Compiler using AMS 180 nm CMOS technology library. In order to obtain the power results, first each architecture has been simulated under each scenario using a specific packet injection rate (PIR). Simulation traces are collected in form of VCD files. The switching activity of the components and

signals are then stored in SAIF format (switching activity interchange format) which is generated by ModelSim. Further, during the synthesis process, the annotated switching activities (the SAIF files) are fed into the synthesis tool (Synopsys Design Compiler) in order to calculate the final static and dynamic power consumption (in mW).

4.5 Experimental Evaluation

In this section QoSinNoC framework better described by evaluating a set of simple placement scenarios. These placement scenarios are mcs_1_a, mcs_1_b, mcs_2_a and mcs_2_b, presented respectively in Fig. 4.2 and Fig. 4.3. All the considered placement scenarios refer to the same MCS scenario: one critical application (C_app) and one non-critical application (NC_app), where C_app uses exactly two node and NC_app_1 — all the remaining nodes. Both mcs_1_a and mcs_1_b are related to the same nodes placement, however they differ in NoC resources placement (links, FIFOs, etc.), due to the different routing algorithms they use (rout_alg_1 and rout_alg_2 respectively). The two routing algorithms are described in detail in Section 4.2.1. The same considerations are true for mcs_2_a and mcs_2_b.

Thus, the considered scenarios are extremely simple and very similar to each other. This has been done on purpose, to evaluate how small placing differences provide very different figures of merit, even in extremely simple case of only two nodes are used by a unique critical application. Furthermore, to better stress the critical points of the proposed approach, the considered scenarios are far from being best case placement. In fact, mcs_1_a and mcs_1_b are the worst-case scenario (critical nodes) placement scenario as they exhibit the maximum Manhattan distance between their critical nodes. Also, mcs_2_a and mcs_2_b are quasi-worst-case scenario, having the property to create two non-critical regions isolated by the critical one.

As explained in previous sections, the proposed techniques (for both rout_alg_A and rout_alg_A) are not limited to systems with one critical application nor to critical application using only two CE. Instead, an arbitrary number of critical nodes are supported as far as the size of the critical regions and the consequent connectivity reduction are considered. However, to simulate a scenario having more than 2 critical nodes will not add much to the general discussion as the scheduling of a critical

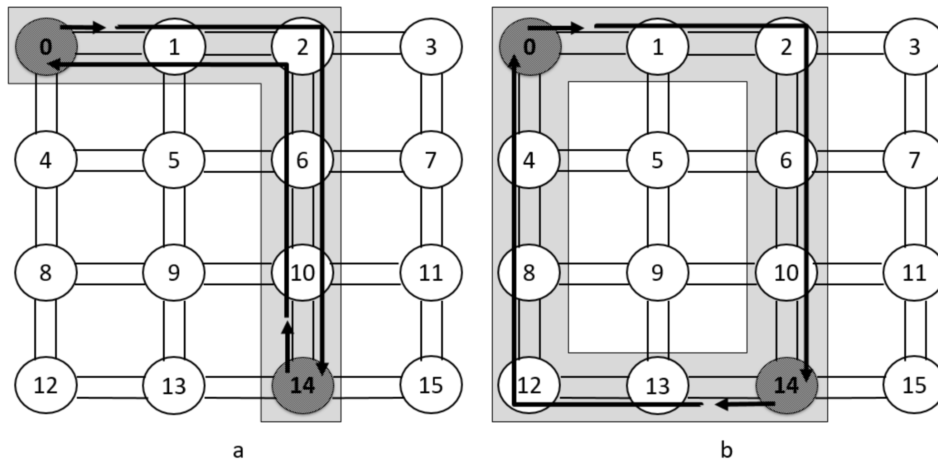

Fig. 4.3 MPSoC with 4x4 mesh topology NoC;
mixed-criticality system example: one C_app, one NC_app;
black arrows: critical traffic paths;
gray-shaded: critical regions (or domains)
a) placing scenario *mcs_2_a*: C_app_1 - N_0 and N_{14}, NC_app_1 - all the remaining nodes, rout_alg_A is used for C_app_1;
b) placing scenario *mcs_2_b*: C_app_1 - N_0 and N_{14}, NC_app_1 - all the remaining nodes, rout_alg_B is used for C_app_1;

application should be designed (best practice) considering the access time for each shared resource in order to avoid intra-partition contention. As interconnection is a shared resource itself, this means that only congestion-free figures make sense for C_apps.

While the congestion-free metrics of the NoC can be computed theoretically, the presence of congestion makes the estimation extremely difficult. To this end, QoSinNoC uses the simulation approach to evaluate the system behavior and collect relevant metrics. The proposed framework allows the generation of traffic that mimics the actual traffic patterns of an application. This is done by fine-grained simulation set-up. The architectures are evaluated by simulation of RTL description of the architectures. The traffic is generated and injected by custom traffic generators which allow to easily configure the synthetic traffic generation. An infinite input buffer at the Network Interface (NI) of each node is assumed in order to maintain the packet injection rate fixed.

Both the considered scenarios were simulated injecting 5000 packets per node. Each node always sent 8-flit packets. The destination of non-critical nodes has been

computed randomly to achieve a random uniform traffic pattern. Table 4.1 compares the placement-independent characteristics of each architecture. Table 4.2 is related to the mcs_1 while the Table 4.3 is related to the mcs_2.

Table 4.1 Architectural only figures of merit (4x4 network).

	Architectures		
	B_LBDR	**E_LBDR**	**Table-based**
Scalability	+	+	-
Non-minimal Path Support	-	-	+
Area (um^2)	1.49 × 10e6	1.54 × 10e6	2.12 × 10e6
Area Overhead	0 %	2.9 %	41.9 %
Critical Path Delay (ns)	9.1	9.0	9.0
Critical Path Delay Overhead	0 %	-0.9 %	-0.9 %

Table 4.2 Figures of merit: mcs_1.

		Architectures		
Metrics	**Routing**	**B_LBDR**	**E_LBDR**	**Table-based**
Non-critical nodes related metrics				
Number of dead nodes	rout_alg_A	5	1	1
	rout_alg_B	10	0	0
Average reachability	rout_alg_A	8.0	8.3	12.0
	rout_alg_B	3.0	7.5	11.9
Saturation throughput (flits/clk cycles/nodes)	rout_alg_A	3.0	3.4	1.1
	rout_alg_B	1.9	3.2	0.8
Critical nodes related metrics				
congestion free latency (clk cycles)	rout_alg_A	30.5	30.5	30.5
	rout_alg_B	30.5	30.5	30.5
Saturation throughput (flits/clk cycles/nodes)	rout_alg_A	0.37	0.37	0.37
	rout_alg_B	0.37	0.37	0.37

Before commenting the figures reported in the Table 4.2 and Table 4.3,it is worth to consider some aspects of the reported metrics. First of all, the number of dead nodes can easily be considered as the most important figure of merit. The reachability metric could be considered the second important, as the throughput and latency are dependent on such metric. Often, high reachability means that also distant nodes can communicate with each other, which increase the congestion and thus makes throughput and latency be worse. However, it should be clear that this will not be a

Table 4.3 Figures of merit: mcs_2.

Metrics	Routing	Architectures		
		B_LBDR	E_LBDR	Table-based
Non-critical nodes related metrics				
Number of dead nodes	rout_alg_A	4	0	0
	rout_alg_B	8	0	0
Average reachability	rout_alg_A	4.2	7.6	13.0
	rout_alg_B	2.3	7.9	12.5
Saturation throughput (flits/clk cycles/nodes)	rout_alg_A	3.7	3.5	1.2
	rout_alg_B	2.3	2.9	1.2
Critical nodes related metrics				
congestion free latency (clk cycles)	rout_alg_A	27.5	27.5	27.5
	rout_alg_B	27.5	27.5	27.5
Saturation throughput (flits/clk cycles/nodes)	rout_alg_A	0.37	0.37	0.37
	rout_alg_B	0.37	0.37	0.37

ceteris paribus comparison (as different reachability implies different traffic patterns), so the throughput and latency figures must be considered accordingly.

From Table 4.2 and Table 4.3 it can be observed how the B_LBDR suffers a doubling of isolated nodes. Indeed, rout_alg_B is not suitable for B_LBDR architecture, at least all the nodes of a critical region are used by that region. Apart B_LBDR, considering the mcs_1, rout_alg_B allows to solve the issue of dead nodes. Thus rout_alg_B is better than rout_alg_A for this placement scenarios. Considering the mcs_2, where there are no dead nodes for both rout_alg_A and rout_alg_B, it can be observed how rout_alg_B provides a reachability metric improvement for E_LBDR, while Table-based has better reachability with rout_alg_A.

4.6 Summary

Considering hardware-implemented temporal isolation of software components running on the same NoC-based MPSoC, this chapter describes: (i) a set of *specific hardware —related techniques* to allow the usage of the related NoC architectures in the context of mixed criticality, and (ii) QoSinNoC framework, which allows an easy comparison between these NoC architectures for a given set of software components to be executed.

The presented techniques, each related to a simple NoC architecture, are derived to address the software components temporal isolation issue. Where the latter is the main issue related to the usage of MPSoC in the context of mixed criticality. Thus, in this chapter these techniques will be sometime referred to as *mixed criticality enforcement techniques*. The proposed techniques are based on NoC resource privatization (thus sharing avoidance) paradigm. The overall solution is based on a one-to-one bound of each NoC internal resource to a software module (i.e., application). Thus, the NoC is partitioned between the applications and the resources are prioritized to avoid any traffic interference. Furthermore, the solution does not allow any inter-application resource sharing. In this way, the temporal (and spatial) partitioning between software modules is achieved by denying any NoC inter-partition interaction.

An important part of the overall contribution (described in this chapter) is QoSinNoC framework, which facilitates the comparison between the considered NoC architectures. QoSinNoC framework can be seen as the union of: (i) a set of considered NoC architectures, (ii) the related techniques to allow their usage in the context of mixed criticality and (iii) a module in charge of computing a set of figures of merit in order to compare which architecture will better fit the implementation of a given set of applications.

The considered NoC architectures as well as the related mixed criticality —enforcement techniques are deliberately as simple as possible. This approach is adopted to make the overall solution be simple, thus, to exhibit a high certification potential. The architectures currently supported by the framework are:

- a basic version of LBDR with no direct support for mixed criticality;
- a modification of LBDR to better support the mixed criticality;
- a basic version of routing table —based routing, to also consider an architecture able to support a non-minimal routing.

The proposed solution also implements spatial partition between critical applications. However, the focus put on the temporal domain as the spatial partitioning is much easier to achieve and does not represent an open issue. Some of the security issues, for instance DoS attack, are solved by the proposed solution as well. However, the security aspects are out of the scope of this dissertation and will not be considered. This partitioning scheme is enforced by physically isolating each critical partition

(i.e., by making impossible any NoC usage that would break the partitioning scheme). This actual isolation is physically implemented by hardware means which are specific to each of the considered NoC architectures. The proposed approach supports an arbitrary high number of levels of criticality as each critical software component is granted with temporal (and spatial) isolation. The proposed approach allows to overcome a strict domain-based partitioning by performing NoC resource reservation at router level, instead of node level. The latter represents an advantage of the state-of-the-art solutions.

The techniques to allow the mixed criticality usage, used by QoSinNoC, are not limited to the particular kind of NoC architecture they were created for. On the contrary, these techniques can be applied for whatever COTS MPSoC as far as the latter presents the key characteristics for their applicability. I.e., the NoC architecture used by that COTS MPSoC should present the similarities with the NoC the technique was developed for (i.e., the NoC considered in QoSinNoC).

Chapter 5

High-level ranking of candidate software susceptibility to soft errors

This chapter describes the third contribution of this dissertation: a technique to address the reliability aspect of the software running on a MPSoC. The reliability definition is given in Section 2.2.1. This attribute of the overall system dependability is directly related to the probability of a failure. A *system failure* (or simply *failure*) is any mismatch in the system actual output with respect to the intended output, where this mismatch concerns both the content of the output and its timing. The reliability aspect is an important aspect, especially for a (safety- or mission-) critical system, as a failure for such system has catastrophic consequences (Section 2.3).

A failure can be provoked by several causes (i.e., faults): from a software bug to a problem inside the manufacturing process of the hardware. Section 2.2.2 provides an exhaustive taxonomy of the faults as well as the details on the process of a fault evolving into a failure. This chapter will focus on a precise type of failures: the radiation-induced failures and the related soft errors (single event upset - SEU). SEU is the most relevant for the radiation induced errors, as it has the highest probability to take place. Section 2.2.2 contains a dedicated subsection, entitled *Radiation-Induced Faults and Errors*, which provides more detail on radiation induced faults and the related errors and failures. The contribution described in this chapter targets the problem of assessing the robustness, against radiation-induces soft errors, of a software component. The particular case is considered: the early design stage of the system, where it exists a function which can be implemented by different candidate

software modules. Thus, this assessment will provide a ranking of these candidates according to their robustness against the considered error. As the early design stage is addressed, the whole analysis is aimed to be as hardware-independent as possible.

The robustness ranking technique presented in this chapter is mainly based on [104], which extends the research proposed in [105]. The latter, in turn is derived from the research published in [106, 107].

The rest of this chapter is structured as follows. Section 5.1 presents some state-of-the-art related the radiation effects evaluation; Section 5.2 presents proposed solution to perform candidate software ranking; Section 5.3 presents the experimental evaluation of the proposed solution; finally, Section 5.4 summarizes the key aspects of the proposed solution;

5.1 Radiation Effects Evaluation: State-of-the-art

Evaluating the dependability characteristics of a software module when no information about the target executing platform is available yet is a challenging task. Considering the early phases of the design process, only a high-level version of the algorithms (e.g., in C or C++) to be considered available, while the target executing platform has still to be decided.

At this early design stage, a possible approach for performing a meaningful reliability analysis of a software component is based on executing it using a meaningful workload, injecting faults and observing the resulting behavior. Where, in case radiation induced faults must be evaluated, the faults model related to radiation effects have to be used.

From a technical point of view, such a kind of analysis typically adopts simulation-based fault injection [42], where bit flips are injected inside the data structures of the program [43–49]. In some cases, other representations of the compression program may also exist (e.g., a Matlab's Simulink model): in these cases, a preliminary analysis can be performed by executing fault injection campaigns on this model [50]. The formal technique [51, 52] can be used as an alternative to the fault injection simulation.

The main issue concerning the aforementioned is the lack of precision. Having no information about the hardware, the injected faults could never precisely model the radiation effects [53].

5.2 Proposed Ranking Technique

5.2.1 The Overall Approach

The proposed technique has been derived to perform a ranking of a set of software modules, candidate to implement a given function of the system (e.g., data compression function). Thus, the proposed ranking technique is intended to use in the early design stage, when for a given function, one possible implementation must be chosen among several candidates. This will give to the system designer information about the reliability dimension of the design space, thus helping the designer in his choice of the most suitable candidate. The overall solution also provides some information to be used during an eventual hardening phase, as an extra feature (naturally implemented by the ranking process).

The proposed ranking technique is implemented at variables-level and requires minimal information about the executing hardware platform. This aspect implies several specific usages and advantages of the proposed technique. First of all, the ranking technique is particularly suitable for COTS components, as their netlist is confidential, thus precluding the usage of low-level approaches. Furthermore, the quasi-independence from the target hardware makes the proposed technique applicable during the early stages of the design cycle.

On the other hand, the high-level approaches are known to not be able to provide high precision and not to be able to exactly capture the actual reliability figures. Indeed, the proposed approach is only applicable for the comparative purposes and only for a set of alternative implementations of a same function.

This comparative characteristic of the proposed approach is the key point to achieve a sufficient level of precision. Given a specific MPSoC (or a simple microprocessor), a low-level approach would be able to capture hardware-specific reliability information. Such information is impossible to be captured by the proposed high-level approach. On the other hand, a very particular case is a comparison

between a set of software modules, running on the same hardware and implementing the same function. The key point of such kind of ranking analysis is the focus on the difference between the reliability figures, instead of the reliability figures itself. This main characteristic of the ranking allows to neglect the common-mode effects, as these are meaningless in comparative terms.

However, for what it concerns the comparison of software modules, running on the same hardware and implementing the same function, this hardware-specific information can be at high extent considered as a common-mode contribution. Thus, a low-level approach will provide information that is mostly negligible for the comparison (i.e., ranking) purposes. The detailed discussion on this topic can be found inside Section 5.3.1.

As no architectural information about the executing hardware is considered during the fault injection phase, the collected information mostly concerns the algorithm- and implementation-specific aspects of the considered software component. The latter are mostly: the fault masking and self-convergence capabilities as well as intrinsic capabilities to detect errors. This aspect of the proposed technique provides the useful for the hardening purposes information, as a secondary and free feature of the overall ranking technique.

As the radiation induced soft errors (in particular SEUs) are considered, the approach mainly targets space and avionic domains. However, as the issue becomes more relevant also for the on-ground applications, the proposed solution could be considered for automotive and other domains as well.

In detail, the proposed technique is based on (i) the execution of the software component and injecting single bit-flip faults in the program variables during its execution, followed by (ii) a post-processing of the obtained figures to obtain a normalized and thus more suitable to for the comparison purposes information. It worth to highlight how the fault injection does not aim the precise reliability figures computation, instead, it aims to achieve a level of precision which is sufficient to perform a comparison.

In order to be ranked by the proposed technique, each software module must be available as source code (C code only) and shall not have any dependency with specific libraries or system calls. These requirements are stemming from the purpose of analyzing the intrinsic — implementation-specific —robustness, without

considering any platform-specific details (e.g., operating system/hardware-specific services);

The scheme of the experimental environment, which implements the proposed technique, is presented in Fig. 5.1. This scheme will be described in detail in Section 5.2.4 and Section 5.2.5.

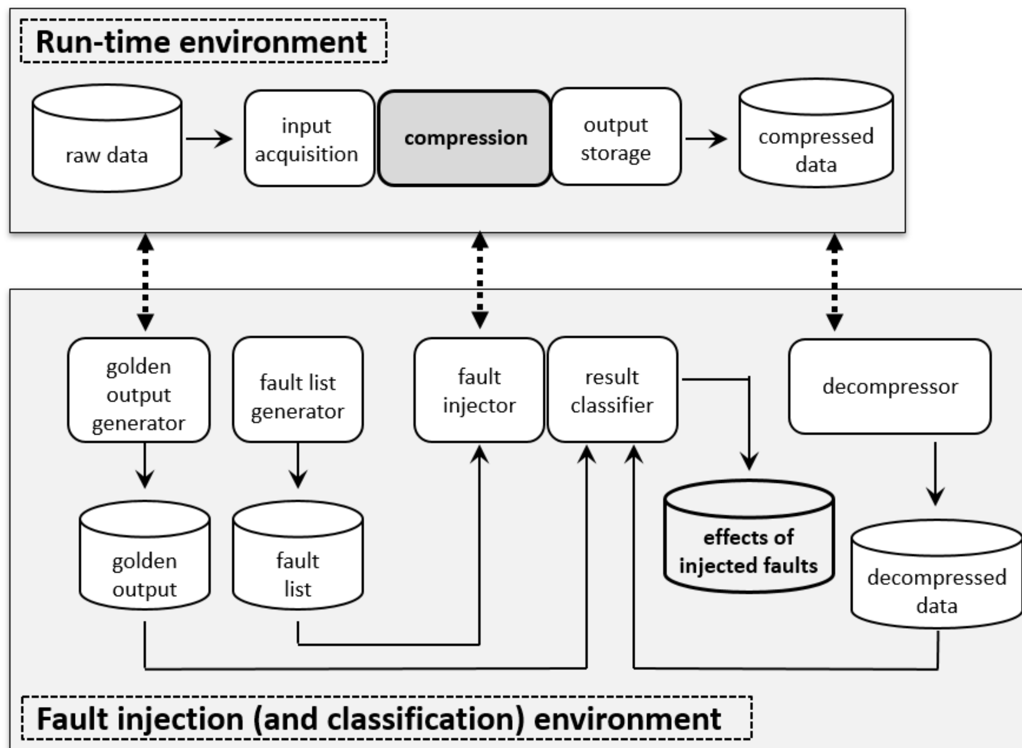

Fig. 5.1 The experimental environment.

To explain the proposed ranking technique, a set of lossless data compression programs will be considered (ranked).

The rest of this section is structured as follows. Section 5.2.2 will present set of lossless compression algorithms to be ranked as well as the realistic workload considered for this purpose. Some of the figures of merit of such programs will be presented as well. Section 5.2.3 will describe a candidate executing hardware platform, considered to evaluate the proposed ranking solution. Section 5.2.4 and Section 5.2.5 explain the experimental environment. Where the former concerns its run-time part, while the latter – fault injection part. Finally, Section 5.2.6 describes

the post-processing of the results obtained by means of fault injection campaign performed by the experimental environment.

5.2.2 Use Case: Sensor Data Lossless Compression

The ranking technique presented in this chapter, has been evaluated considering a set of different (software) implementation for the data compression function. In detail, this compression function is the core for a data logger system to be deployed in a satellite launcher, thus the lossless telemetry sensor data compression is considered.

The different lossless compression programs, used to demonstrate the proposed ranking technique, belong to completely different groups and are based on completely different paradigms. This is a perfect example of the situation; the proposed ranking technique is meant for: one function to be implemented in software and multiple candidates to become that software module.

In order to improve the precision of the considered high-level approach, it is crucial to consider a realistic workload. Thus, this chapter will evaluate the considered software module candidates on actual telemetry data collected during Ariane launcher missions.

This section will first briefly describe the data used as realistic workload for the considered compression programs. Then, the implementation details of these compression programs will be briefly presented. Finally, the characteristics and the figures of merit of the considered programs will be presented.

Considered Workload

As a realistic workload I used two data sets collected during Ariane missions. Pressure sensor data (1,531 samples, 8 bits per sample) and vibration sensor data (3,072 samples, 8 bits per sample).

LZW_12

Dictionary-based techniques [108] take advantage of commonly occurring pattern sequences by using a dictionary in such a way that repeating patterns are replaced by a codeword that points to the index of the dictionary containing the pattern.

In particular, the LZ78 (Lempel-Ziv 1978) technique has been considered. This technique achieves compression by replacing repeated occurrences of data with references to a dictionary that is built based on the input data stream.

The pseudocode of the considered LZW implementation is presented in Algorithm 5.1. The particular implementation is characterized by using 12 bits per LZW code and will be referred to as LZW_12.

Algorithm 5.1 Pseudocode of LZW_12.

```
 1: procedure LZW_12( input_data, output_buffer )
 2: CLEAR hash table
 3: INITIALIZE LZW_dictionary with values from 0 to 255
 4: SET found_string to NULL
 5: SET curr_string_to_encode to NULL
 6:     for each element to encode do
 7:         APPEND element to encode to curr_string_to_encode
 8:         SEARCH (using hash table) curr_string_to_encode inside
    LZW_dictionary
 9:         if previous search successful then
10:             found_string ← curr_string_to_encode
11:         else
12:             WRITE the code of found_string to output_buffer
13:             found_string ← NULL
14:             if LZW_dictionary is full then
15:                 REINITIALIZE LZW_dictionary
16:             end if
17:             code_curr_string_to_encode ← (code_last_element_LZW_dictionary
    + 1)
18:             APPEND curr_string_to_encode to the LZW_dictionary
19:             curr_string_to_encode ← NULL
20:         end if
21:     end for
22:     if found_string not equal to NULL then
23:         WRITE the code of found_string to output_buffer
24:     end if
25:     return (number of codes written to output_buffer)
26: end procedure
```

RICE

Golomb-Rice-based techniques [108] are a subset of Golomb-based techniques, which exploit variable-length codes based on a model of the probability of the data dealt as natural numbers. To code a number according to Golomb paradigm we must find the quotient and remainder of the division by a pre-defined divisor. The quotient is then expressed in unary notation, and the remainder is expressed in truncated binary notation. A Golomb-Rice code is a Golomb code where the divisor is a power of two, enabling an efficient implementation using shift and mask operations rather than division and modulo. The interesting aspect of these techniques is that they are suitable for compressing streams of data, such as those acquired by the telemetry system, without using memory-demanding data structures such as dictionaries, and thus they are suitable when resource-constrained implementations are necessary.

The pseudocode of the considered Golomb-Rice implementation is presented in Algorithm 5.2. In this dissertation, this implementation will be called RICE.

RLE

Run Length Encoding (RLE) technique [108] consists in encoding each sequence of consecutive symbols as the symbol followed by the number of times the symbol is repeated in the sequence. This technique is extremely simple but able to provide good results if the data to compress presents sufficiently long sequences of repeated symbols.

A straightforward implementation of RLE technique is considered, featuring no pre-processing stage. The pseudocode of this implementation is reported in Algorithm 5.2.

Code Size and Memory Footprint

Table 5.1 reports the characteristics of the considered compression software modules in term of lines of code and memory occupation, where:

- *Lines of code*, which is the number of lines of code composing the considered compression programs;

Algorithm 5.2 Pseudocode of RICE.

```
1: procedure RICE( input_data, output_buffer )
2:
3:     WRITE the first element to encode to output_buffer
4:     block_size ← 16
5:     for each block of block_size elements to encode do
6:         // "zeros_block_counter" counts consecutive mapped_diff blocks containing only zeros
7:         zeros_blocks_counter ← 0
8:         start_index ← 0, except for the first block, in that case start_index is set to 1
9:         for each element j of the of the block, from start_index to block_size do
10:            diff_var ← difference between element j and element (j-1), where element -1 corresponds to the last element of the previous block
11:            // COMPUTE the "mapping to positive"
12:            if (diff_var >= 0) then
13:                mapped_diff[j] ← (diff_var*2)
14:            else
15:                mapped_diff[j] ← (-diff_var*2 - 1)
16:            end if
17:            if mapped_diff contains only zeros then
18:                INCREASE zeros_blocks_counter
19:            else
20:                if zeros_blocks_counter not equal 0 then
21:                    WRITE to output_buffer the code related to the value of zeros_blocks_counter
22:                end if
23:                COMPUTE Rice code for the current block mapped_diff[j], with j from start_index to block_size
24:                WRITE to output_buffer the just computed code
25:            end if
26:        end for
27:    end for
28:    if zeros_blocks_counter not equal 0 then
29:        WRITE to output_buffer the code related to the value of zeros_blocks_counter
30:    end if
31:    return (number of codes written to output_buffer)
32: end procedure
```

Algorithm 5.3 Pseudocode of RLE.

```
1:  procedure RLE( input_data, output_buffer )
2:
3:      // "run_length" counts the number of consecutive repetitions of the current
    element to encode
4:      run_length ← 0
5:      for each element to encode do
6:          if current element differs from the previous then
7:              WRITE to output_buffer the current element to encode followed by
    one byte binary coding of run_length
8:              run_length ← 0
9:          else
10:             INCREASE run_length
11:         end if
12:     end for
13:     if run_length not equal 0 then
14:         WRITE to output_buffer the current element to encode followed by one
    byte binary coding of run_length
15:     end if
16:     return (number of codes written to output_buffer)
17: end procedure
```

- *Memory occupation* of the considered compression programs. In this column, we reported only the size of the data structure needed for the compression, while the input/output buffers have been neglected.

Table 5.1 Compression programs characteristics.

Compression program	Lines of code	Memory occupation [B]
RICE	180	64
LZW_12	110	21,091
RLE	20	17

From this column, the much higher footprint of the LZW_12 is evident. Indeed, being based on a dictionary, LZW_12 requires a much higher amount of memory compared to the others;

Table 5.2 reports the figures of merit of the considered compression software modules when the aforementioned telemetry sensor data is considered, in particular compression ratio and compression rate, where:

- *Compression ratio* is defined as number of bytes of the original data set over the number of bytes of the compressed data set;
- *Compression rate* is defined as the number of compressed bytes over the time required for the compression. The time required for the conversion is computed considering the compression performed by the OpenRISC 1200 ISS, running at 100 MHz clock frequency.

Table 5.2 Compression programs figures of merit: compression ratio and compression rate.

	Pressure sensor data			Vibration sensor data			**WEIGHED AVARAGE**		
Figures of merit	LZW_12	RLE	RICE	LZW_12	RLE	RICE	**LZW_12**	**RLE**	**RICE**
Compration ratio	12.6	20.1	25.5	2.4	0.6	2.4	**5.8**	**7.1**	**10.1**
Compression rate [MB/s]	1.4	5.8	1.9	0.9	4.8	0.7	**1.1**	**5.1**	**1.1**

5.2.3 Candidate Executing Hardware Platform

The proposed ranking solution will be evaluated considering OR1200 [109, 110] as a candidate executing hardware. The OR1200 is a 32-bit scalar RISC with Harvard microarchitecture. It features 5 stage integer pipeline, virtual memory support (MMU) and basic DSP capabilities.

5.2.4 Run-time Environment

The run-time environment, in charge of executing the considered software components, has been implemented by an Intel Core i7 powered workstation running a 64-bit Linux distribution. Each software component (to be analyzed) is considered composed of three modules, reported in Fig. 5.1, which are:

- *Input acquisition module*: this module is usually in charge of loading the data to processed, from the file system, to the final memory location. In case and operating system is present, this operation is done exploiting the dedicated

system APIs. However, in the embedded systems domain, a file system is not always present. As this module is not related to the compression itself, it is not considered for the proposed technique purposes. To address the general case and to avoid dependencies with any operating system, the input memory array is pre-initialized with the data to be compressed. Thus, the input acquisition module is made by a function only passing to the processing module a pointer to the data to compress;

- *Processing module*: this module implements the actual compression, by processing the input memory array and producing an output memory array. This module is coded in C language and does not make use of any operating system API. Thanks to this approach, the compression program is highly portable, and can be reused in any target hardware platform (e.g., an embedded processor, as well as an ASIC or FPGA after high-level synthesis);

- *Output module*: this module is usually in charge of downloading the output memory array into the file system. In case and operating system is present, this operation is done exploiting the dedicated system APIs. However, in the embedded systems domain, a file system is not always present. As this module is not related to the compression itself, it is not considered for the proposed technique purposes. In order to address the general case and to avoid dependencies with any operating system, this module is symbolic and has not been actually implemented. I.e., the execution of the compression program terminates when all the compressed data are inside the memory.

5.2.5 Fault Injection Environment

To rank the robustness against the radiation-induced soft errors of each compression program, a high-level fault injection campaign is performed using the selected workload (see Section 5.2.2). The effects of radiations are approximated as single bit flips in the data structures of the compression program (see the related to radiations part of Section 2.2.2).

The proposed solution is aware that the main drawback of any high-level approach is the limited capability to capture the exact behavior of the hardware affected by radiation-induced soft errors [53]. Indeed, the aim of the proposed technique is to perform a ranking of a set of software components, while any usage to compute the

quantitative reliability figures is discouraged. Thus, this technique benefits from the fact that a comparison requires a lower level of precision with respect to the quantitative reliability figures computation. However, the main challenge of the overall technique is indeed to achieve a sufficient precision to allow a meaningful comparison.

The adopted fault injection environment allows to evaluate the effects of radiation induced soft errors, modeled as single bit flips of program variables. The fault injection environment, described in Fig. 5.1, is composed of five modules:

- *Golden output generator*, which is in charge of running the compression program once, without any fault injection. The run-time environment is used to collect the binary file storing the correct compressed data array (i.e., the golden output), to be used as a reference for classifying the effects of the injected faults. The execution time of the compression program is collected as well;

- *Fault list generator*, which is in charge of producing, for each considered program, the list of faults to be injected. The fault list generator produces a predefined number of faults (i.e., the fault list), each defined as the tuple (injection time, identifier name, bitmask), which values are chosen randomly. The injection time is chosen based on the execution time of the compression program. The identifier is randomly chosen based on the set of data structures of the compression program (including both scalar variables, such as temporary variables, indexes, as well as data arrays). The bitmask corresponds to the single bit in the selected identifier that must be bit-flipped at the injection time;

- *Fault injector*, which is a GDB-based script that for each fault starts the execution of the considered compression program in debug mode, advances the execution until injection time, injects the bit-flip inside the identifier to attack, and resumes the execution of the program until its termination. The fault injection makes use of the run-time environment for compression program execution. Fault injection takes place after the input acquisition module completed its activity, and before the compression module completed. As a result, faults are injected only during the execution of the compression program;

- *Decompressor*, which decompresses the compressed data to evaluate the intrinsic masking and detection capabilities. In order to avoid useless computations only corrupted data is decompressed, avoiding decompressing data which is identical to the golden data. This module is implemented outside the run-time environment because the decompression is supposed to take place on earth, thus it is not subject to the radiation-induced soft errors;

- *Result classifier*, which is implemented by a set of BASH scripts, analyzes the termination status of the compression program under fault injection as well as the produced output. In particular, the termination of the program could be normal, or caused by the detection of an exception (e.g., segmentation fault). In case the program terminates normally, the output produced by the program is compared to the golden output. In case the produced output differs from the golden output, the decompressor is used to evaluate whether the data can be still perfectly decompressed or at least the corruption is detected by the decompressor.

Faults are classified (by the result classifier) according to the following categories:

- *Silent*: the faulty execution completes within a pre-defined amount of time and produces either the same compressed data as the golden run or (in case compressed data corruption took place) the decompressor can perfectly decompress the corrupted data;

- *Run-time detected*: the faulty execution triggers some error detection mechanism. Considering both the specificity of the compression algorithms and the nature of soft errors, in case a soft error is detected, the recovery can be done by simply repeating the compression. The sub-classification differs for the two considered levels of fault injection. This category is further detailed in:
  - *Timeout error*: the faulty execution does not complete within a pre-defined amount of time;
  - *Segmentation fault error*, when a memory access violation takes place.

- *Detected by decompressor*: this category groups faults for which the compression execution produced a corrupted compression and no run-time detection of the corruption took place. Thus, the corrupted compression data has been sent to earth. In this category, the decompressor is not able to tolerate the corruption, thus it is not able to obtain the correct data. The data have been permanently

lost and no recovery is possible, still the data corruption has been detected by the decompressor, preventing the corrupted data from being considered as correct. Furthermore, this implementation-specific (or algorithm-specific) detection capability can be exploit during the hardening phase in order to implement run-time detection mechanism;

- *Wrong output*: this category groups faults for which the compression execution had produced a corrupted compression, and no run-time error detection took place. Thus, the corrupted compression data has been sent to earth. In this category, the decompressor is not able to tolerate the corruption, thus it is not able to obtain the correct data. Furthermore, the decompressor was not able to identify the corruption either, thus the corrupted data is considered correct.

5.2.6 Post-processing and Candidate Ranking Process

The proposed approach aims to be as hardware-independent as possible, however, to provide a sufficient level of precision, some target hardware information has still to be considered. This kind of information does not require the actual usage of the target hardware, thus the proposed technique is suitable for the design space exploration, as a set of possible execution platform can be quickly analyses.

Considering the fault classification as in Section 5.2.5, the outcomes considered as a measure of unreliability are *Wrong output* outcomes as well as *Detected by decompressor* outcomes. The latter is considered in this category because the detection is done in a moment when no recovery action can be taken any more. In order to avoid confusion and to keep notation simple, I define a further outcome category *Corrupted* outcome, defined in Equation 5.1, as a sum of the aforementioned two.

$$Corrupted = Wrong\ output + Detected\ by\ decompressor \tag{5.1}$$

To perform a meaningful comparison, the following basic information related to the target executing hardware platform should still be considered: (i) the execution time of the compression and (ii) the number of actually used registers. This information must be known for each compression program and it is required to normalize the results of fault injections. In particular, the execution time must be considered

because programs having longer execution time are likely to experience a higher number of soft errors during their execution [111]. Whereas, programs using less registers have a lower probability to be affected by a soft error during their execution.

However, this information does not require a candidate executing hardware to be actually used to execute the investigated software modules. Indeed, the execution time can be collected by an instruction set simulator, which availability is fairly common. While, the number of used registers can be collected from the compiled code, thus a compiler is sufficient to provide this information.

To consider the different amount of used registers, the normalization of Silent outcome is presented in Equation 5.2, where:

- $S_{program_i}$ is the number of Silent outcomes obtained for the i-th considered program;
- $NUR_S_{program_i}$ is the, normalized per register usage, amount of Silent outcomes, obtained for the i-th considered program;
- $UR_{program_i}$ is the number of actually used registers by the i-th considered program.

$$NUR_S_{program_i} = S_{program_i} \times \frac{MAX_i\{UR_{program_i}\}}{UR_{program_i}} \tag{5.2}$$

To consider the difference in execution times, the normalization of Corrupted outcome is presented in Equation 5.3, where:

- $C_{program_i}$ is the number of Corrupted outcomes obtained for the i-th considered program;
- $NET_S_{program_i}$ is the number of Corrupted outcomes normalized per execution time, obtained for the i-th considered program;
- $ET_{program_i}$ is the execution time of the i-th considered program.

$$NET_C_{program_i} = C_{program_i} \times \frac{ET_{program_i}}{MIN_i\{ET_{program_i}\}} \tag{5.3}$$

5.3 Experimental Evaluation

This section will first describe the environment used to validate the proposed ranking technique (Section 5.3.1). Then, in Section 5.3.2, the proposed technique is used to rank the considered set of compression programs (see Section 5.2.2), considering the candidate executing hardware platform (see Section 5.2.3).

5.3.1 Validation Environment

The validation environment has the same structure as the experimental environment (Fig. 5.1). Its run-time part is the same described in Section 5.2.4, with the exception that it is implemented by an instruction set simulator of the candidate executing hardware, namely OR1ksim [112].

To validate the proposed approach, the results of high-level fault injection have been compared with those gathered using register-level fault injection simulation. In detail, general-purpose registers (GPRs) of a candidate executing platform have been considered. The injection of faults inside the memory has not been considered, as memories (both on chip and external) used in space applications, are typically protected by error detection/correction capabilities [54, 55]. Thus, the memories can be considered as immune to radiation-induced soft errors.

Despite the GPRs-level fault injection simulation does not provide precise figures about the real behavior of the hardware, it is reasonable to assume that their precision allows the comparison among software modules implementing the same function (e.g., data compression). In particular, in [53] the authors provide a detailed analysis of the flip-flop error propagation for a considered in-order processor. According to this analysis, about 18.6% of the injected flip-flop errors lead to a failure which is not related to the register file. This means that 18.6% of radiation induced soft errors cannot be modeled by only injecting faults inside the register file. However, these errors are related to either processor state corruption (e.g., program counter corruption), cache/interrupt controller malfunctioning or memory subsystem malfunctioning. Considered the nature of these errors, it is reasonable to assume that them have a common mode effect for all the considered software components, thus could be neglected for what it concerns the comparison between these software components.

The fault injection environment used to validate the proposed approach is similar to the one described in Section 5.2.5. The differences are the following:

- for what it concerns the *Fault list generator*, a single bit-flip injection in the General-Purpose Registers (GPRs) is used;
- for what it concerns the *Run-time detected* outcome, it is further detailed in:
  - *Timeout error*: the faulty execution does not complete within a pre-defined amount of time;
  - *Bus error*, when the injected fault leads the program to access an invalid memory address;
  - *Alignment error*, when the fault leads the program to access a misaligned memory address;
  - *Illegal instruction*, when the fault leads the processor to access a memory area not containing valid instructions.

5.3.2 Results Analysis

The candidate hardware's compiler has been used to collect the information about the number of registers actually used by the proposed architectures. These figures, reported in Table 5.6, have been used to process (Equation 5.2) the fault injection campaign results gathered by the experimental environment. These normalized results of 100,000 injected faults, are reported in Table 5.4. The results further normalized by execution time, as in Equation 5.3, are reported in Table 5.5. From the figures of Table 5.5, the RLE compression program turns to be the most robust, followed by LZW_12 and RICE. However, the small difference existing between RLE and LZW_12 suggests using the ranking between the two carefully.

Table 5.3 Number of actually used general purpose registers.

	Compression program		
	LZW_12	RLE	RICE
Number of used GPRs	18	13	18

To validate the proposed technique, the validation environment described in Section 5.3.1, has been used to inject 100,000 faults and the collected figures are reported in Table 5.6. Table 5.7 reports figures achieved applying the normalization

Table 5.4 Results of fault injection into program variables, normalized per number of used registers.

Outcome [%]		Pressure sensor data			Vibration sensor data			WEIGHED AVARAGE		
		LZW_12	RLE	RICE	LZW_12	RLE	RICE	**LZW_12**	**RLE**	**RICE**
Silent		99.1	64.3	46.7	94.6	66.7	39.5	**96.1**	**65.9**	**41.9**
Detected at run-time	Timeout error	0.0	0.0	0.0	0.0	0.0	0.0	**0.0**	**0.0**	**0.0**
	Bus error exception	0.1	17.4	5.9	0.1	16.4	5.3	**0.1**	**16.7**	**5.5**
Wrong output		0.1	1.5	5.0	0.8	2.5	0.6	**0.6**	**2.2**	**2.1**
Detected by decompressor		0.7	16.8	42.4	4.5	14.4	54.6	**3.2**	**15.2**	**50.5**

Table 5.5 Results of fault injection into program variables, normalized with respect to both the number of used registers and the execution time.

	LZW_12	RLE	RICE
Wrong output undetected at run-time (Wrong output + Detected by decompressor)	18,083	17,400	250,274

of Equation 5.3 to the figures of Table 5.6. The figures of Table 5.7 show the same ranking obtained by applying the proposed approach (Table 5.5), i.e., RLE is the most robust one, followed by the LZW_12 and finally by RICE. Thus, the proposed high-level approach is considered validated by a more precise one.

5.4 Summary

This chapter describes the third contribution of this dissertation: a technique to address the reliability aspect of software running on a MPSoC. In detail, the reliability is considered for what it concerns the robustness against radiation-induced soft errors. The proposed technique has been derived to perform a ranking of a set of software modules, candidate to implement a given function of the system (e.g., data compression function). Thus, the proposed ranking technique is intended to use in the early design stage, when for a given function, one possible implementation

Table 5.6 Results of fault injection into general purpose registers.

Outcome [%]		Pressure sensor data			Vibration sensor data			WEIGHED AVARAGE		
		LZW_12	RLE	RICE	LZW_12	RLE	RICE	**LZW_12**	**RLE**	**RICE**
Silent		80.8	82.0	49.0	80.5	84.0	53.9	**80.6**	**83.3**	**52.3**
Detected at run-time	Timeout error	1.4	3.7	9.1	1.7	3.5	8.7	**1.6**	**3.6**	**8.8**
	Bus error exception	12.0	5.4	15.3	11.8	5.0	15.0	**11.9**	**5.1**	**15.1**
	Alignment exception	0.4	1.0	0.7	0.4	0.8	1.2	**0.4**	**0.9**	**1.0**
	Illegal instruction exception	0.1	0.0	0.0	0.0	0.1	0.0	**0.0**	**0.1**	**0.0**
Wrong output		1.8	0.9	13.5	2.6	1.1	11.4	**2.3**	**1.0**	**12.1**
Detected by decompressor		3.5	7.0	12.4	3.0	5.5	9.8	**3.2**	**6.0**	**10.7**

Table 5.7 Results of fault injection into general purpose registers, normalized with respect to the execution time.

	LZW_12	RLE	RICE
Wrong output undetected at run-time (Wrong output + Detected by decompressor)	26,172	7,187	108,483

must be chosen among several candidates. This will give to the system designer information about the reliability dimension of the design space, thus helping the designer in his choice of the most suitable candidate. The overall solution also provides some information to be used during an eventual hardening phase, as an extra feature (naturally implemented by the ranking process).

The proposed approach aims to be as hardware-independent as possible, however, to provide a sufficient level of precision, some information about target executing hardware has still to be considered. In detail, the proposed ranking technique is implemented at program-variables-level. This aspect implies several specific usages and advantages of the proposed technique. First of all, the ranking technique is particularly suitable for COTS components, as their netlist is confidential, thus precluding the usage of low-level approaches. Furthermore, the quasi-independence

from the target hardware makes the proposed technique applicable during the early stages of the design cycle.

On the other hand, the high-level approaches are known to not be able to provide high precision and not to be able to exactly capture the actual reliability figures. Indeed, the proposed approach is only applicable for the comparative purposes and only for a set of alternative implementations of a same function.

This comparative characteristic of the proposed approach is the key point to achieve a sufficient level of precision. Given a specific MPSoC (or a simple microprocessor), a low-level approach would be able to capture hardware-specific reliability information, which are impossible to be evaluated by the proposed high-level approach. However, for what it concerns the comparison of software modules (running on that specific hardware), considering the literature on the flip-flop fault injection effects, it is reasonable to consider this hardware-specific information as a common-mode contribution. Thus, the base assumption of the proposed technique is that a low-level approach will provide information that is mostly negligible for the comparison (i.e., ranking) purposes. For what it concerns the fault injection simulation done at program-variables-level, the former assumption has been validated by comparing the achieved ranking to the one derived with the register-level fault injection simulation.

As no architectural information about the executing hardware is considered during the fault injection phase, the collected information mostly concerns the algorithmic- and implementation-specific aspects of the considered software component. The latter are mostly: the fault masking and self-convergence capabilities as well as intrinsic capabilities to detect errors. This aspect of the proposed technique provides the useful for the hardening purposes information, as a secondary and free feature of the overall ranking technique.

As the radiation induced soft errors (in particular SEUs) are considered, the approach mainly targets space and avionic domains. However, as the issue becomes more relevant also for the on-ground applications, the proposed solution could be considered for automotive and other domains as well.

In detail, the proposed technique is based on (i) the execution of the software component and injecting single bit-flip faults in the program variables during its execution, followed by (ii) a post-processing of the obtained figures to obtain a normalized and thus more suitable to for the comparison purposes information. It

worth to highlight how the fault injection does not aim the precise reliability figures computation, instead, it aims to achieve a level of precision which is sufficient to perform a comparison.

The proposed approach aims to be as hardware-independent as possible, however, to provide a sufficient level of precision, some target hardware information has still to be considered. In particular, an estimation of the execution time and the number of used registers allows to better correlate the results of the fault injection campaign to the real behavior of the candidate executing hardware platform. However, this information does not require a candidate executing hardware to be actually used to execute the investigated software modules. Indeed, the executing time can be collected by an instruction set simulator, which availability is fairly common. While, the number of used registers can be collected from the compiled code, thus a compiler is sufficient to provide this information.

The proposed approach is evaluated by ranking a set of lossless compression programs, candidate for the data logging sub-system. To validate the proposed approach, the results of high-level fault injection have been compared with those gathered with register-level fault injection simulation. In detail, general-purpose registers (GPRs) of a candidate executing platform have been considered. The injection of faults inside the memory has not been considered, as memories (both on chip and external) used in space applications, are typically protected by error detection/correction capabilities, and can be considered as immune to radiation-induced soft errors.

Chapter 6

Conclusions and Future Work

The usage of multi-core microprocessors is today normally avoided by the critical application domain. This is particularly true for the avionics domain, where an independent certification entity exists. The main issue related to the multi-core architectures is the *lack of predictability* due to the presence of shared resources. This issue is of paramount importance in a very complex system, which proper functioning must be rigorously demonstrated at each level of abstraction. In fact, an avionic certification process is extremely complex and thus expensive, and although possible, can easily be not feasible from economical point of view.

Driven by the even-growing complexity and number of required applications, aviation domain community constantly increase their attention towards multi-core processor–based MCSs. While some certification is expected by the end of 2019, at the best of author's knowledge no multi-core–based MCS exists yet.

While the certification of a quad-core microprocessor is reportedly close to be achieved, the market is offering much more advanced devices — MPSoC — integrating tens or hundreds of cores on the same chip.

It is expected that one day a multi-core–based MCS will become a common practice. And it is also expected that sometime later, this solution will not be able to provide the required computational power, thus the MPSoC usage will go under avionic community investigation. The same way it happened for multi-core microprocessors, investigated to replace the single-core ones.

The situation in avionic domain, described so far, is not much different from the other critical domain.

In this dissertation I investigate the usage of NoC-based MPSoC in the context of mixed criticality. The particular focus is put upon the lack of predictability issue related to the intrinsic NoC contention, which is identified as the main obstacle for the multi-core usage in the context of critical systems. In fact, this lack of predictability has a direct impact on the functional safety aspect.

In this last chapter, the three contributions of this thesis are first summarized (Section 6.1, Section 6.2 and Section 6.3). Finally, Section 6.4 will present some ideas to extend the presented line of research as such as well as the proposed solutions individually.

6.1 Contribution 1

The first and the main contribution of this dissertation is a technique derived to solve the main issue that prevents the usage of MPSoC in the context of mixed criticality. In detail, I address the software components temporal isolation issue. In fact, the parallel execution of a set of software components, on the same COTS NoC-based MPSoC, represents the main threat for the functional safety of the systems.

The proposed solution is based on resource privatization–based NoC partitioning, where each critical application has a dedicated partition. The proposed solution enforces this partitioning and ensures the absence of both inter-partition NoC implicit contention and inter-partition communication. Thus, each critical partition is isolated from any other partition. The partitioning technique, used by the proposed solution, is based on resource privatization and software-implemented on-line monitoring and enforcing of the aforementioned partition isolation.

The isolation monitoring (and enforcement) task is the core of the proposes technique and it is based on the deterministic (or de facto deterministic) routing used by NoC. Thus, this particular routing characteristic is the main requirement for the target NoC design. The other and only requirements are that both routing algorithm and NoC topology must be known.

The traffic filtering is implemented as purely software module to be inserted inside the RTOS, in order to provide a modular and reusable design.

The proposed approach supports an arbitrary high number of levels of criticality as each critical software component is granted with temporal (and spatial) isolation.

Advantages

There are several advantages of proposed approach with respect to the existing partitioning and resource privatization–based approaches.

The main advantage is the low complexity, for what it concerns both (i) the implementation of the technique itself and (ii) the NoC architecture targeted by the proposed approach. This aspect is crucial as the complexity is the main issue for the certification process, which is mandatory for any critical system.

Furthermore, the proposed solution overcomes the strict domain-based segregation. This means that what is normally defined as a critical region of the NoC, in the proposed solution, can overlap with a non-critical region without any temporal isolation loss. This allows a much flexible and thus more efficient NoC utilization, allowing in turn more application to be mapped on the same MPSoC.

To further improve the NoC utilization, the proposed solution also features a redirection functionality for the non-critical applications. This feature allows to extend the overall routing algorithm of non-critical traffic. The resulting extra connectivity is used to mitigate the connectivity reduction eventually provoked by the proposed resource privatization paradigm.

Limitations

The natural limitation of the proposed solution concerns deterministic or de facto deterministic characteristic, the routing algorithm used by the NoC is expected to exhibit. This requirement is due to the really core functionality of the proposed approach to exploit this particular characteristic. On the other hand, this is not a big limitation, if critical domain is considered. In fact, an adaptive routing algorithm can be easily considered excessively complex from the certification point of view.

The other requirements concerning the routing and topology to be both known, follows the same reasoning. In a context of critical systems, the computing hardware must be sufficiently known. Thus, it is natural to conclude that an MPSoC, which routing algorithm or topology are unknown, cannot be used in context of critical

systems, irrespective of the fact whether the proposed solution is applicable, or it is not.

Another natural limitation of the proposed approach is a potential connectivity reduction cost, which is presented by any resource privatization–based approach. Thus, a particular cure must be put during the placement phase in order to avoid distant nodes and large critical regions.

A limitation of the proposed solution is the absence of inter-partition communication. This limitation comes directly from the decision to have such limitation. The reason, of such choice, is a much simpler certification process, which is a big advantage. On the other hand, the presence of inter-partition communication would make the overall solution more flexible. This aspect will be discussed more in detail in the Section 6.4, which concerns the future work.

6.2 Contribution 2

The second contribution of this dissertation (Chapter 4) is a solution to the same issue addressed the first contribution in Chapter 3. The main difference between the two is that while the first contribution presents a purely software solution targeting fairly generic COTS components, the second contribution presents a set of hardware-specific solutions and a framework to facilitate their comparison.

The addressed issue is the software components temporal isolation one, which is the main issue to prevent the usage of MPSoC in the context of mixed criticality. In fact, the parallel execution of a set of software components, on the same NoC-based MPSoC, represents the main threat for the functional safety of the systems.

Thus, the second contribution of this dissertation presents: (i) a set of *specific-hardware–related techniques* to allow the usage of the related NoC architectures in the context of mixed criticality, and (ii) QoSinNoC framework, which allows an easy comparison between these NoC architectures for a given set of software components to be executed.

The presented techniques, each related to a simple NoC architecture, are derived to address the software components temporal isolation issue. Where the latter is the main issue related to the usage of MPSoC in the context of mixed criticality. Thus, in this chapter these techniques will be sometime referred to as *mixed criticality*

enforcement techniques. The proposed techniques are based on NoC resource privatization (thus sharing avoidance) paradigm. The overall solution is based on a one-to-one bound of each NoC internal resource to a software module (i.e., application). Thus, the NoC is partitioned between the applications and the resources are prioritized to avoid any traffic interference. Furthermore, the solution does not allow any inter-application resource sharing. In this way, the temporal (and spatial) partitioning between software modules is achieved by denying any NoC inter-partition interaction.

An important part of the overall contribution (described in this chapter) is QoSinNoC framework, which facilitates the comparison between the considered NoC architectures. QoSinNoC framework can be seen as the union of: (i) a set of considered NoC architectures, (ii) the related techniques to allow their usage in the context of mixed criticality and (iii) a module in charge of computing a set of figures of merit in order to compare which architecture will better fit the implementation of a given set of applications.

The considered NoC architectures as well as the related mixed-criticality enforcement techniques are deliberately as simple as possible. This approach is adopted to make the overall solution be simple, thus, to exhibit a high certification potential. The proposed solution also implements spatial partition between critical applications. However, the focus put on the temporal domain as the spatial partitioning is much easier to achieve and does not represent an open issue. Some of the security issues, for instance DoS attack, are solved by the proposed solution as well. However, the security aspects are out of the scope of this dissertation and will not be considered. This partitioning scheme is enforced by physically isolating each critical partition (i.e., by making impossible any NoC usage that would break the partitioning scheme). This actual isolation is physically implemented by hardware means which are specific to each of the considered NoC architectures. The proposed approach allows to overcome a strict domain-based partitioning by performing NoC resource reservation at router level, instead of node level. The latter represents an advantage of the state-of-the-art solutions.

The techniques to allow the mixed criticality usage, used by QoSinNoC, are not limited to the particular kind of NoC architecture they were created for. On the contrary, these techniques can be applied for whatever COTS MPSoC as far as the latter presents the key characteristics for their applicability. I.e., the NoC

architecture used by that COTS MPSoC should present the similarities with the NoC the technique was developed for (i.e., the NoC considered in QoSinNoC).

The proposed approach supports an arbitrary high number of levels of criticality as each critical software component is granted with temporal (and spatial) isolation.

Advantages

There are several advantages of proposed approach with respect to the existing hardware-specific partitioning and resource privatization–based approaches.

The main advantage is the low complexity, for what it concerns both (i) the implementation of the technique itself and (ii) the NoC architectures targeted by the proposed approach. This aspect is crucial as the complexity is the main issue for the certification process, which is mandatory for any critical system.

Furthermore, the proposed solution overcomes the strict domain-based segregation. This means that what is normally defined as a critical region of the NoC, in the proposed solution, can overlap with a non-critical region without any temporal isolation loss. This allows a much flexible and thus more efficient NoC utilization, allowing in turn more application to be mapped on the same MPSoC.

Limitations

The natural limitation of each of the proposed mixed criticality enforcement technique is the specific hardware characteristics, exploited by the former.

Another natural limitation of the proposed approach is a potential connectivity reduction cost, which is presented by any resource privatization–based approach. Thus, a particular cure must be put during the placement phase in order to avoid distant nodes and large critical regions.

A limitation of the proposed solution is the absence of inter-partition communication. This limitation comes directly from the decision to have such limitation. The reason, of such choice, is a much simpler certification process, which is a big advantage. On the other hand, the presence of inter-partition communication would make the overall solution more flexible. This aspect will be discussed more in detail in the Section 6.4, which concerns the future work.

6.3 Contribution 3

The third contribution of this dissertation (Chapter 5) is a methodology to address the reliability aspect of COTS MPSoC used for space and avionics systems.

The proposed technique allows to perform a ranking of a set of different software implementations of a given functionality. A typical situation, the proposed technique is intended to be used for, is when different algorithms or even paradigms exist to implement the required function (e.g., data compression function). Thus, the proposed solution is intended to help the designer in the choice of the most suitable (in terms of reliability) software implementation for a given system function, when more candidates for that function exist. The overall solution also provides some information to be used during an eventual hardening phase, as an extra feature (naturally implemented by the ranking process).

The proposed ranking technique is implemented at variables-level and requires minimal information about the computing hardware platform. This aspect implies several specific usages and advantages of the proposed technique. First of all, the ranking technique is particularly suitable for COTS components, as their netlist is confidential, thus precluding the usage of low-level approaches. Furthermore, the quasi-independence from the target hardware makes the proposed technique applicable during the early stages of the design cycle.

On the other hand, the high-level approaches are known to not be able to provide high precision and not to be able to exactly capture the actual reliability figures. Indeed, the proposed approach is only applicable for the comparative purposes and only for a set of alternative implementations of a same function.

This comparative characteristic of the proposed approach is the key point to achieve a sufficient level of precision. Given a specific MPSoC (or a simple microprocessor), a low-level approach would be able to capture hardware-specific reliability information, which are impossible to be evaluated by the proposed high-level approach. However, for what it concerns the comparison of software modules (running on that specific hardware), this hardware-specific information can be at high extent considered as common-mode contribution. Thus, a low-level approach will provide information that is mostly negligible for the comparison (i.e., ranking) purposes.

As no architectural information about the computing hardware is considered during the fault injection phase, the collected information mostly concerns the algorithmic- and implementation-specific aspects of the considered software component. The latter are mostly: the fault masking and self-convergence capabilities as well as intrinsic capabilities to detect errors. This aspect of the proposed technique provides the useful for the hardening purposes information, as a secondary and free feature of the overall ranking technique.

As the radiation induced soft errors (in particular SEUs) are considered, the approach mainly targets space and avionic domains. However, as the issue becomes more relevant also for the on-ground applications, the proposed solution could be considered for automotive and other domains as well.

In detail, the proposed technique is based on (i) the execution of the software component and injecting single bit-flip faults in the program variables during its execution, followed by (ii) a post-processing of the obtained figures to obtain a normalized and thus more suitable to for the comparison purposes information. It worth to highlight how the fault injection does not aim the precise reliability figures computation, instead, it aims to achieve a level of precision which is sufficient to perform a comparison.

The proposed approach is evaluated by ranking a set of lossless compression programs, candidate for the data logging sub-system. To validate the proposed approach, the results of high-level fault injection have been compared with those gathered with register-level fault injection simulation. In detail, general-purpose registers (GPRs) of a candidate executing platform have been considered. The injection of faults inside the memory has not been considered, as memories (both on chip and external) used in space applications, are typically protected by error detection/correction capabilities, and can be considered as immune to radiation-induced soft errors.

Advantages

As the proposed solution is implemented at high level its main advantage it cost, as the techniques implemented at lower levels of abstraction are more expensive both in terms of time and money.

Limitations

The proposed technique only performs a ranking of a set of candidates for the same function. Furthermore, this ranking is just an indication as the overall precision is not estimated.

6.4 Future Work

Considering this thesis line of research as such, a possible future work consists in putting the focus on the automotive domain. In fact, although considering critical systems in general, this thesis' focus mainly on space and especially on avionic domains. On the other hand, automotive industry presents similar — yet different — requirements, thus the applicability to the automotive domain should be explicitly investigated.

This future work is justified by the ever-growing need of performance the automotive domain ex experiencing. While the avionics and space industry have SWaP reduction as their main driving factor (towards multi-core microprocessor adaptation), the automotive industry has its own factors as well. These factors are the evolution of infotainment functionalities but first of all the rise of autonomous driving vehicles. The latter will require a huge computational power and will be safety critical, thus a perfect match for an MPSoC-based MCS. Furthermore, avionics and space domains are particularly conservative, and the new technology adaptation is particularly slow for them. This means that probably the first usage of MPSoC in the context of mixed criticality will be done in automotive domain instead.

Apart from the future work concerning this thesis line of research as such, the rest of this section will list possible future work which is specific to each of the presented contributions.

For what it concerns the first contribution, presented in Chapter 3, several possible extensions and future activities can be identified.

1. *Inter-partition communication implementation*. The current solution does not allow inter-partition communication. The reason for such limitation is the easiest certification process. However, it could be possible that the extra flexibility, given by the inter-partition communication, will worth the extra

complexity given by its implementation. A possible future work can explore the cost of the mechanisms able to provide such inter-partition communication.

2. *Tolerance to hardware faults*. The solution proposed in Chapter 4 was specifically developed to address the partitioning issue related to the MPSoC usage, and in particular the temporal isolation between software components running of such system. However, the proposed approach appears promising from the (hardware) fault tolerance point of view, thus this aspect could worth a further investigation.

3. *Testing of the proposed solution on a COTS MPSoC*. The next step for the proposed solution validation is to identify a COTS MPSoC and to test the proposed solution on such hardware platform.

For what it concerns the second contribution, presented in Chapter 4, several possible extensions and future activities can be identified. As both the first and the second contributions target the same issue (i.e., temporal isolation of the software components) the future activities are somehow similar.

1. *Inter-partition communication implementation*. The current solution does not allow inter-partition communication. The reason for such limitation is the easiest certification process. However, it could be possible that the extra flexibility, given by the inter-partition communication, will worth the extra complexity given by its implementation. A possible future work can explore the cost of the mechanisms able to provide such inter-partition communication.

2. *Further NoC architectures*. The set of considered architectures can be extended with further architectures, e.g.., source routing -based. Different form the mesh topologies can be investigated as well.

3. *Tolerance to hardware faults*. The solution proposed in Chapter 4 was specifically developed to address the partitioning issue related to the MPSoC usage, and in particular the temporal isolation between software components running of such system. However, the proposed approach appears promising from the (hardware) fault tolerance point of view, thus this aspect could worth a further investigation.

For what it concerns the last contribution of this dissertation, presented in Chapter 5, several possible extensions and future activities can be identified as well.

1. *More precision: hardware timer–based system.* So far, the proposed solution was implemented on a common workstation computer, were the time of injection was affected by the operating system services. A possible improvement to be investigated is the creation of a dedicated processor-based system featured with a hardware timer. Such system will thus allow to achieve a much better precision in fault injection process, thus will increase the precision of the overall solution.

2. *Not only compression function: verification on other functions.* The technique presented in Chapter 5 was derived to be applied to a set of software components, which all implement the same function, which are candidates to be actually used by the system to implement that function. However, so far only the compression function has been evaluated. Thus, the future work should evaluate the proposed solution applicability in general case, by evaluating more functionalities.

3. *More candidate executing hardware platforms.* More candidate executing hardware platforms evaluated by the proposed technique will better validate the general applicability of the proposed solution

4. *More precise techniques for validation.* So far, the proposed technique was only validated by fault injection inside the processor register file. A future work will consider more precise fault injection techniques to better validate the precision of the proposed technique.

References

[1] IFIP Working Group on Dependable Computing and Fault Tolerance. IFIP WG 10.4. Dependability: Basic Concepts and Terminology, 1990.

[2] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[3] International Roadmap Committee. International technology roadmap for semiconductors, 2013.

[4] CBS News. Toyota "Unintended Acceleration" Has Killed 89. http://www.cbsnews.com/news/toyota-unintended-acceleration-has-killed-89/, 2010. Accessed: 2019-01-04.

[5] Wikipedia. 2009–11 Toyota vehicle recalls. https://en.wikipedia.org/wiki/2009%E2%80%9311_Toyota_vehicle_recalls, 2013. Accessed: 2019-01-04.

[6] Junko Yoshida. Toyota case: Single bit flip that killed. http://www.eetimes.com/document.asp?doc_id=1319903, 2013. Accessed: 2019-01-04.

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

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

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

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

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

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

[13] Paul Parkinson David Radack, Harold G. Tiedeman. Civil Certification of Multi-core Processing Systems in Commercial Avionics. Technical report, Rockwell Collins, 2018.

[14] Alexandre Esper, Geoffrey Nelissen, Vincent Nélis, and Eduardo Tovar. An industrial view on the common academic understanding of mixed-criticality systems. *Real-Time Systems*, 54(3):745–795, 2018.

[15] 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 *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–5. IEEE, 2014.

[16] Ermis Papastefanakis, Xiaoting Li, and Laurent George. Deterministic scheduling in network-on-chip using the trajectory approach. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 60–65. IEEE, 2015.

[17] Qin Xiong, Fei Wu, Zhonghai Lu, and Changsheng Xie. Extending real-time analysis for wormhole nocs. *IEEE Transactions on Computers*, 66(9):1532–1546, 2017.

[18] Stefano Esposito, Massimo Violante, Marco Sozzi, Marco Terrone, and Massimo Traversone. A novel method for online detection of faults affecting execution-time in multicore-based systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(4):94, 2017.

[19] Adam Kostrzewa, Sebastian Tobuschat, and Rolf Ernst. Self-aware network-on-chip control in real-time systems. *IEEE Design & Test*, 35(5):19–27, 2018.

[20] Benoît Dupont De Dinechin, Duco Van Amstel, Marc Poulhiès, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[21] Georgia Giannopoulou. *Implementation of Mixed-Criticality Applications on Multi-Core Architectures*. PhD thesis, ETH Zurich, 2017.

[22] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24. IEEE, 2016.

[23] Quentin Perret, Pascal Maurere, Eric Noulard, Claire Pagetti, Pascal Sainrat, and Benoit Triquet. Temporal isolation of hard real-time applications on many-core processors. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–11. IEEE, 2016.

[24] Mohammad Fattah, Amir-Mohammad Rahmani, Thomas Canhao Xu, Anil Kanduri, Pasi Liljeberg, Juha Plosila, and Hannu Tenhunen. Mixed-criticality run-time task mapping for noc-based many-core systems. In *2014 22nd Euromicro international conference on parallel, distributed, and network-based processing*, pages 458–465. IEEE, 2014.

[25] Tilera Corporation. Tile Processor Architecture Overview for the TILEPro Series. http://www.mellanox.com/repository/solutions/tile-scm/docs/UG120-Architecture- Overview-TILEPro.pdf. Accessed: 2019-01-02.

[26] 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.

[27] Mikael Millberg, Erland Nilsson, Rikard Thid, and Axel Jantsch. Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 2, pages 890–895. IEEE, 2004.

[28] Tobias Bjerregaard. The mango clockless network-on-chip: Concepts and implementation. *IMM, Danmarks Tekniske Universitet*, 2005.

[29] Théodore Marescaux and Henk Corporaal. Introducing the supergt network-on-chip; supergt qos: More than just gt. In *2007 44th ACM/IEEE Design Automation Conference*, pages 116–121. IEEE, 2007.

[30] Sebastian Tobuschat and Rolf Ernst. Efficient latency guarantees for mixed-criticality networks-on-chip. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 113–122. IEEE, 2017.

[31] Alan Burns, James Harbin, and Leandro Soares Indrusiak. A wormhole noc protocol for mixed criticality systems. In *2014 IEEE Real-Time Systems Symposium*, pages 184–195. IEEE, 2014.

[32] Leandro Soares Indrusiak, James Harbin, and Alan Burns. Average and worst-case latency improvements in mixed-criticality wormhole networks-on-chip. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 47–56. IEEE, 2015.

[33] Boris Grot, Joel Hestness, Stephen W Keckler, and Onur Mutlu. Kilo-noc: a heterogeneous network-on-chip architecture for scalability and service guarantees. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 401–412. ACM, 2011.

[34] Kees Goossens, John Dielissen, and Andrei Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *IEEE Design & Test of Computers*, 22(5):414–421, 2005.

[35] Hamidreza Ahmadian and Roman Obermaisser. Time-triggered extension layer for on-chip network interfaces in mixed-criticality systems. In *2015 Euromicro Conference on Digital System Design*, pages 693–699. IEEE, 2015.

[36] Arkadiy Morgenshtein, Avinoam Kolodny, and Ran Ginosar. Link division multiplexing (ldm) for network-on-chip links. In *2006 IEEE 24th Convention of Electrical & Electronics Engineers in Israel*, pages 245–249. IEEE, 2006.

[37] Francisco Triviño, José L Sánchez, Francisco J Alfaro, and José Flich. Network-on-chip virtualization in chip-multiprocessor systems. *Journal of Systems Architecture*, 58(3-4):126–139, 2012.

[38] Thomas Hollstein, Siavoosh Payandeh Azad, Thilo Kogge, and Behrad Niazmand. Mixed-criticality noc partitioning based on the nocdepend dependability technique. In *2015 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–8. IEEE, 2015.

[39] Serhiy Avramenko and Massimo Violante. RTOS Solution for NoC-Based COTS MPSoC Usage in Mixed-Criticality Systems. *Journal of Electronic Testing*, pages 1–16, 2019.

[40] Serhiy Avramenko, Siavoosh Payandeh Azad, Stefano Esposito, Behrad Niazmand, Massimo Violante, Jaan Raik, and Maksim Jenihhin. Upgrading QoSinNoC: Efficient Routing for Mixed-Criticality Applications and Power Analysis . In *26th IFIP/IEEE International Conference on Very Large Scale Integration and System-on-Chip design*, Verona, 2019. IEEE.

[41] José Flich Cardo. *Cost Effective Routing Implementations for On-chip Networks*. PhD thesis, Universitat Politècnica de València, 2010.

[42] Régis Leveugle, A Calvez, Paolo Maistri, and Pierre Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 502–506. European Design and Automation Association, 2009.

[43] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[44] Alfredo Benso, Stefano Di Carlo, Giorgio Di Natale, Luca Tagliaferri, and Paolo Prinetto. Validation of a software dependability tool via fault injection experiments. In *Proceedings Seventh International On-Line Testing Workshop*, pages 3–8. IEEE, 2001.

[45] Alfredo Benso, Alberto Bosio, Stefano Di Carlo, and Riccardo Mariani. A functional verification based fault injection environment. In *22nd IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2007)*, pages 114–122. IEEE, 2007.

[46] Weining Gu, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Error sensitivity of the linux kernel executing on powerpc g4 and pentium 4 processors. In *DSN*, page 887, 2004.

[47] Guilin Chen, Gu Chen, Mahmut Kandemir, Narayanan Vijaykrishnan, and Mary Jane Irwin. Object duplication for improving reliability. In *Asia and South Pacific Conference on Design Automation, 2006.*, pages 6–pp. IEEE, 2006.

[48] Daniel Chen, Gabriela Jacques-Silva, Zbigniew Kalbarczyk, Ravishankar K Iyer, and Bruce Mealey. Error behavior comparison of multiple computing systems: A case study using linux on pentium, solaris on sparc, and aix on power. In *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 339–346. IEEE, 2008.

[49] Keun Soo Yim, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Measurement-based analysis of fault and error sensitivities of dynamic memory. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 431–436. IEEE, 2010.

[50] H Hakobyan, P Rech Ufrgs, M Sonza Reorda, and Massimo Violante. Early reliability evaluation of a biomedical system. In *2014 9th International Design and Test Symposium (IDT)*, pages 45–50. IEEE, 2014.

[51] Vishal Chandra Sharma, Arvind Haran, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. Towards formal approaches to system resilience. In *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*, pages 41–50. IEEE, 2013.

[52] Maha Kooli and Giorgio Di Natale. A survey on simulation-based fault injection tools for complex systems. In *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6. IEEE, 2014.

[53] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A Abraham, and Subhasish Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Proceedings of the 50th Annual Design Automation Conference*, page 101. ACM, 2013.

[54] Michel Pignol. Dmt and dt2: Two fault-tolerant architectures developed by cnes for cots-based spacecraft supercomputers. In *12th IEEE International On-Line Testing Symposium (IOLTS'06)*, pages 10–pp. IEEE, 2006.

[55] 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):2687–2694, 2015.

[56] BA Nayfeh and K Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[57] Samuel H Fuller and Lynette I Millett. Computing performance: Game over or next level? *Computer*, 44(1):31–38, 2011.

[58] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[59] Vikas Agarwal, MS Hrishikesh, Stephen W Keckler, and Doug Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 248–259. ACM, 2000.

[60] David A Patterson. Latency lags bandwith. *Communications of the ACM*, 47(10):71–75, 2004.

[61] ARM. AMBA advanced extensible interface (AXI) protocol specification. https://www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications, 2011. Accessed: 2019-01-04.

[62] William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[63] Érika Cota, Alexandre de Morais Amory, and Marcelo Soares Lubaszewski. *Reliability, Availability and Serviceability of Networks-on-chip*. Springer Science & Business Media, 2011.

[64] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1):1, 2006.

[65] Christopher J Glass and Lionel M Ni. The turn model for adaptive routing. *Journal of the ACM (JACM)*, 41(5):874–902, 1994.

[66] Ge-Ming Chiu. The odd-even turn model for adaptive routing. *IEEE Transactions on parallel and distributed systems*, 11(7):729–738, 2000.

[67] Sheng Ma, Libo Huang, Mingche Lai, and Wei Shi. *Networks-on-chip: From Implementations to Programming Paradigms*. Morgan Kaufmann, 2014.

[68] Luis Miguel Pinho, Eduardo Quinones, and Andrea Marongiu. *High-Performance and Time-Predictable Embedded Computing*. River Publishers, 2018.

[69] Jose Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection networks*. Morgan Kaufmann, 2003.

[70] Kees Goossens, John Dielissen, Jef van Meerbergen, Peter Poplavko, Andrei Rădulescu, Edwin Rijpkema, Erwin Waterlander, and Paul Wielage. Guaranteeing the quality of services in networks on chip. In *Networks on chip*, pages 61–82. Springer, 2003.

[71] Santanu Chattopadhyay and Santanu Kundu. *Network-on-Chip: The Next Generation of System-on-Chip Integration*. CRC press, 2014.

[72] Intel. Arteris FlexNoC. http://www.arteris.com/, 2006. Accessed: 2019-01-04.

[73] Giovanni De Micheli and Luca Benini. Networks on chips: 15 years later. *Computer*, (5):10–11, 2017.

[74] Intel. Product Change Notification 116378 - 00. https://www.intel.co.uk/content/www/uk/en/products/processors/xeon-phi/xeon-phi-processors.html, 2016. Accessed: 2019-01-04.

[75] Intel. Product Change Notification 116378 - 00. qdms.intel.com/dm/i.aspx/9C54A9A7-BF37-4496-B268-BD2746EA54D3/PCN116378-00.pdf, 2018. Accessed: 2019-01-04.

[76] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, et al. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE micro*, 22(2):25–35, 2002.

[77] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1137–1142. ACM, 2012.

[78] Julien Mottin, Mickael Cartron, and Giulio Urlini. The sthorm platform. In *Smart Multicore Embedded Systems*, pages 35–43. Springer, 2014.

[79] Olga Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. *Software-implemented hardware fault tolerance*. Springer Science & Business Media, 2006.

[80] Neil R Storey. *Safety critical computer systems*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[81] Kenneth A LaBel, Charles E Barnes, Paul W Marshall, Cheryl J Marshall, Allan H Johnston, Robert A Reed, Janet L Barth, Christina M Seidleck, Sammy A Kayali, and Martha V O'Bryan. A roadmap for nasa's radiation effects research in emerging microelectronics and photonics. In *2000 IEEE Aerospace Conference. Proceedings (Cat. No. 00TH8484)*, volume 5, pages 535–545. IEEE, 2000.

[82] NASA. Draft Single Event Effects Specification. https://radhome.gsfc.nasa.gov/radhome/papers/seespec.htm, 2015. Accessed: 2019-01-04.

[83] Serhiy Avramenko, Stefano Esposito, and Massimo Violante. Rice coding-based lite compression algorithm with controlled error for sensor data. In *2018 5th IEEE International Workshop on Metrology for AeroSpace (MetroAeroSpace)*, pages 6–10. IEEE, 2018.

[84] 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.

[85] SAE ARP4754. Certification considerations for highly-integrated or complex aircraft systems. *SAE International*, 1996.

[86] SAE ARP4754A. Guidelines for development of civil aircraft and systems. *SAE International*, 2010.

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

[88] FAA Order. 8040.4: Safety risk management (1998).

[89] FAA System Safety Handbook, Chapter 2: System Safety Policy and Process. Technical report, FAA, 2000.

[90] Leanna Rierson. *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance*. CRC Press, 2013.

[91] Bob Green, Joseph Marotta, Brian Petre, Kirk Lillestolen, Richard Spencer, Nikhil Gupta, Daniel O'Leary, Jason Dan Lee, John Strasburger, Arnold Nordsieck, et al. Handbook for the selection and evaluation of microprocessors for airborne systems. *FAA, Tech. Rep.*, 2011.

[92] Xavier Jean, Marc Gatti, Guy Berthon, and Marc Fumey. Mulcors-use of multicore processors in airborne systems. *EASA, Tech. Rep.*, 2012.

[93] Lloyd Condra, Gary Horan, and other. Afe 75 cots aeh issues and emerging solutions. *FAA, Tech. Rep.*, 2014.

[94] Laurence H Mutuel, Xavier Jean, Vincent Brindejonc, Anthony Roger, Thomas Megel, and E Alepins. Assurance of multicore processors in airborne systems. *FAA, Tech. Rep.*, 2017.

[95] Serhiy Avramenko, Stefano Esposito, and Massimo Violante. Efficient Software-Based Partitioning for Commercial-off-the-Shelf NoC-based MP-SoCs for Mixed-Criticality Systems. In *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design*, pages 189–194, Platja d'Aro, 2018. IEEE.

[96] Serhiy Avramenko, Stefano Esposito, and Massimo Violante. RTOS for mixed criticality applications deployed on NoC-based COTS MPSoC. In *2018 IEEE 19th Latin-American Test Symposium*, pages 1–6, Sao Paulo, 2018. IEEE.

[97] Justin Littlefield-Lawwill and Larry Kinnan. System considerations for robust time and space partitioning in integrated modular avionics. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pages 1–B. IEEE, 2008.

[98] Leonidas Kosmidis, Cristian Maxim, Victor Jegu, Francis Vatrinet, and Francisco J Cazorla. Industrial experiences with resource management under software randomization in arinc653 avionics environments. In *Proceedings of the International Conference on Computer-Aided Design*, page 108. ACM, 2018.

[99] Suzana Milutinovic, Enrico Mezzetti, Jaume Abella, Tullio Vardanega, and Francisco J Cazorla. On uses of extreme value theory fit for industrial-quality wcet analysis. In *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–6. IEEE, 2017.

[100] Adam Kostrzewa, Selma Saidi, and Rolf Ernst. Dynamic control for mixed-critical networks-on-chip. In *2015 IEEE Real-Time Systems Symposium*, pages 317–326. IEEE, 2015.

[101] Serhiy Avramenko, Siavoosh Payandeh Azad, Stefano Esposito, Behrad Niazmand, Massimo Violante, Jaan Raik, and Maksim Jenihhin. QoSinNoC: Analysis of QoS-Aware NoC Architectures for Mixed-Criticality Applications. In *2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, pages 1–7, Budapest, 2018. IEEE.

[102] Samuel Rodrigo Mocholí. *Cost Effective Routing Implementations for On-chip Networks*. PhD thesis, Universitat Politècnica de València, 2010.

[103] Siavoosh Payandeh Azad, Behrad Niazmand, Karl Janson, Thilo Kogge, Jaan Raik, Gert Jervan, and Thomas Hollstein. Comprehensive performance and robustness analysis of 2d turn models for network-on-chips. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE, 2017.

[104] Serhiy Avramenko, Matteo Sonza Reorda, Massimo Violante, and Görschwin Fey. RTOS Solution for NoC-Based COTS MPSoC Usage in Mixed-Criticality Systems. *Journal of Electronic Testing*, 33:53–64, 2017.

[105] Serhiy Avramenko, Matteo Sonza Reorda, Massimo Violante, and Görschwin Fey. Analysis of the effects of soft errors on compression algorithms through fault injection inside program variables. In *2016 17th Latin-American Test Symposium*, pages 14–19, Foz do Iguacu, 2016. IEEE.

[106] Serhiy Avramenko, M Sonza Reorda, Massimo Violante, Görschwin Fey, J-G Mess, and R Schmidt. On the robustness of dct-based compression algorithms for space applications. In *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–2. IEEE, 2016.

[107] Tino Flenker, Jan Malburg, Görschwin Fey, Serhiy Avramenko, Massimo Violante, and Matteo Sonza Reorda. Towards making fault injection on abstract models a more accurate tool for predicting rt-level effects. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 533–538. IEEE, 2017.

[108] David Salomon. *A guide to data compression methods*. Springer Science & Business Media, 2013.

[109] Damjan Lampret and Julius Baxter. Openrisc 1200 ip core specification (preliminary draft), 2014.

[110] openrisc 1200. https://opencores.org/projects/or1k_old/openrisc 1200. Accessed: 2019-01-04.

[111] George A Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I August, and Shubhendu S Mukherjee. Design and evaluation of hybrid fault-detection systems. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 148–159. IEEE Computer Society, 2005.

[112] Jeremy Bennett. Or1ksim user guide. *Embecosm*, 2008.