Real-time and reliable design for safety-critical embedded systems

Angeliki Kritikakou

UNIVERSITÉ DE RENNES 1

HABILITATION À DIRIGER DES RECHERCHES

(HDR)

UNIVERSITY OF RENNES 1

DOCTORAL SCHOOL N° 601
Mathematics, Science and Technology of Computer Science and Communication
Domain: *Computer Science*

By

Angeliki KRITIKAKOU

Real-time and reliable design for safety-critical embedded systems

HDR presented and defended in Rennes, on 28 November 2022
Research center: Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA)

Reviewers:

Alberto BOSIO	Professor, École Centrale de Lyon, France
Lirida Alves DE BARROS NAVINER	Professor, Telecom ParisTech, France
Jari NURMI	Professor, Tampere University, Finland

Examinators:

Smail NIAR	Professor, Université Polytechnique Hauts-de-France, France
Olivier SENTIEYS	Professor, University of Rennes 1, France
Dimitrios SOUDRIS	Professor, National University of Athens, Greece

Abstract

Embedded systems in safety-critical domains, such as avionics, space, automotive, health-care etc., require hard real-time and reliable application execution. As applications are becoming more complex, their computational demands scale rapidly. To address these demands, architectures with multiple processing elements (cores) and application specific hardware accelerators are required. Multicore architectures are able to concurrently execute a high volume of applications, while hardware accelerators are tailored to the needs of the application. To provide hard real-time guarantees, the Worst-Case Execution Time (WCET) has to be considered during system analysis and design. However, WCET are overestimated due to application and hardware complexity. Furthermore, reliable execution is under threat due to the increased fault susceptibility of modern electronic systems, such as manufacturing process variation, aging and soft errors. Typical reliability solutions, that rely on full hardware or software redundancy, usually entail significant cost, latency and resource overheads, which are often not suitable for critical embedded systems. To cost-effectively address these reliability threats, the system must be properly analysed and enhanced with effective fault tolerance means. Meanwhile, the energy consumption of embedded systems has become crucial and energy efficient electronic devices should be designed. Platforms have been enhanced with Dynamic Voltage and Frequency Scaling (DVFS), which scales down the processor supply voltage and frequency, whenever possible. However, such approaches usually have a negative impact on execution time and reliability. To provide real-time, reliable and low energy execution over multicore architectures, efficient analysis and deployment approaches, along with run-time adaptation capabilities, are required.

In this context, a set of design-time task deployment approaches have been proposed, under real-time and energy budget constraints. A decomposition-based algorithm has been designed to provide the optimal solution for independent Imprecise Computation (IC) tasks on symmetric multiprocessors and extended for platforms with DVFS capabilities. We have leveraged our decomposition-based solution for dependent IC tasks and heterogeneous multicore platforms and proposed an accelerated, but still optimal, version. Optimal and heuristic approaches have been proposed, considering task migration on asymmetric multicore platforms. Furthermore, optimal and heuristic task deployment approaches have been designed, under real-time and reliable constraints for multicore systems with DVFS. Our fault-tolerant task deployment method jointly optimizes frequency assignment, task allocation, task scheduling, and task duplication for three DVFS schemes, i.e., task-level, processor-level and system-level, for independent and dependent tasks. Last, we have taken into account the inter-processor communication and proposed a task deployment process for multicore platforms with Network-on-Chip (NoC) and DVFS, where routing path selection is also taken into account. Moreover, a novel heuristic method is proposed to enhance scalability, achieving good solutions with low computation time.

To analyse the reliability of complex hardware designs, we proposed a cross-layer reliability

analysis framework, from the semiconductor layer up to the application layer, for transient faults caused by single radiation particles. We combined statistical analysis with single-cycle gate-level fault injection and microarchitecture-level fault injection, and explored the impact of faults to the application execution. Furthermore, run-time hardware mechanisms have been proposed to deal with faults on heterogeneous Very Large Instruction Word (VLIW) processors and NoC. For VLIW processors, the proposed mechanism performs instruction triplication considering short-term transient faults. We leveraged this mechanism for permanent faults, where instruction triplication and re-scheduling is applied taking into account the status of the Function Units (FUs). To reduce the performance degradation due to the instruction level fault tolerance, dynamic instruction re-scheduling is applied based on the status of the faulty FU in a coarse-grained way and a fine-grained way. For NoC, we proposed a technique for fault mitigation of multiple permanent faults, where flits at the subflit scale are re-organized in order to move the fault impact on the Low Significant Bits (LSBs). Furthermore, a redundancy approach is presented to handle critical data. To reduce hardware overheads, a region-based version is proposed, which protects regions, instead of routers.

Last, a set of run-time approaches have been proposed to reduce the pessimism introduced by the WCET overestimations. More precisely, when interference-sensitive WCET (isWCET) estimations are used during system design, they are only valid for the specific schedule solution they have been computed for. To support a safe adaptation of interference-sensitive schedules, we proposed a run-time approach that enables parallel execution of the control phases on each core with a fine-grained protection. Our second contribution comes from the observation that by enforcing the partial order of tasks, we limit the performance improvement that can be achieved through run-time adaptation. To further improve performance gains, we leverage our approach with a safe relaxation of the partial order of tasks. Furthermore, existing approaches are based on WCET estimations obtained during design-time, and thus, they are not able to take advantage of the actual execution progress of the tasks. To deal with this limitation, we proposed an approach that computes dynamically new safe estimations of the WCET during execution, based on the task progress. The updated WCET estimations are used to derive the available time-slack and postpone mode switch in mixed critical systems. The proposed approach has been leveraged in order to dynamically decide when to invoke the controller, reducing the time overhead, further increasing the gains.

As future research directions, we will focus on providing the means to design, in a near-optimal and efficient way, real-time and reliable embedded systems for safety-critical domains, with unreliable components, under multiple reliability threats. Our first direction is the analysis of the timing impact of transient faults occurring on cores. To achieve that, we will leverage our reliability analysis framework to incorporate the application timing behaviour and to include the impact of interference on shared resources. We will enhance WCET estimations with fault awareness and design low cost fault tolerance techniques to protect the system. Our second direction concerns the design of real-time and reliable heterogeneous systems and specialized hardware accelerators, which is the next promising architecture to deal with the increasing demands for high computation capabilities in timely manner. We will design hardware accelerators for WCET-aware and fault-aware systems to extend homogeneous multicore systems towards domain specific heterogeneous multicore architectures. To achieve that, we will adapt reliability and WCET analysis frameworks and design real-time and fault tolerance techniques for hardware accelerators, such as accelerators dedicated for Artificial Intelligence (AI). Although soft errors have been considered as the most important ones, until recently, with the further ongoing reduction of transistors size, system aging is becoming more and more sensitive to the workload. Different cores are subjected to different amount of stress as a result of varying workloads, leading to aging imbalance among cores. We

will focus on dedicated cross-layer fault-aware and WCET-aware approaches to efficiently deal with workload-dependent aging faults for safety-critical systems. Furthermore, the systems are susceptible to multiple types of reliability threats, potentially correlated with each other. Therefore, we will leverage the proposed approaches to consider multiple sources for reliability threats. Last, but not least, the continuous decrease of technology size has pushed CMOS devices to their limits, suffering from high static power consumption, reduced reliability, high cost and scaling issues. Future computing systems will exploit emerging technologies and novel computing paradigms, such as Processing In Memory (PIM). In order to be used in safety-critical domains with real-time and reliable guarantees, we will propose approaches to provide timing and reliability analysis and Design Space Exploration (DSE) for these emerging technologies and computing paradigms, especially when used as shared resources.

Contents

1 Introduction and Motivation **1**
1.1 Context . . . 1
1.2 State-of-the-Art (SoA) and positioning of current contributions . . . 3
1.2.1 WCET-Aware (WA) techniques . . . 4
1.2.2 Fault-Aware (FA) techniques . . . 7
1.2.3 WCET-aware (WA) and Fault-Aware (FA) techniques . . . 10
1.3 Conclusions . . . 11

2 WCET-aware task deployment for multicore architectures **13**
2.1 Interference-pessimistic design-time mapping for IC tasks . . . 13
2.1.1 Context . . . 13
2.1.2 State-of-the-Art . . . 14
2.1.3 Contributions . . . 15
2.1.4 System model . . . 15
2.1.5 General problem formulation . . . 16
2.1.6 Optimal decomposition-based method . . . 17
2.1.7 Accelerated optimal decomposition-based method . . . 18
2.1.8 Heuristic methods . . . 19
2.1.9 Evaluation . . . 19
2.2 Interference-controlled run-time adaptation of *is*WCET time-triggered task execution 23
2.2.1 Context . . . 23
2.2.2 State-of-the-Art . . . 24
2.2.3 Contributions . . . 25
2.2.4 System model . . . 26
2.2.5 Enforcing partial order through fine-grained protection mechanism . . . 26
2.2.6 Relaxing partial order mechanism . . . 28
2.2.7 Evaluation . . . 29
2.3 Risk-permissive run-time adaptation of task execution in mixed-critical systems . . . 33
2.3.1 Context . . . 33
2.3.2 State-of-the-Art . . . 33
2.3.3 Contributions . . . 35
2.3.4 System model . . . 36
2.3.5 Design time analysis for high criticality tasks . . . 36
2.3.6 Run-time control mechanism . . . 38
2.3.7 Evaluation . . . 42
2.4 Conclusions . . . 44

3 Fault-aware techniques for hardware design **47**
3.1 Run-time instruction re-scheduling for VLIW processors . . . 47
3.1.1 Context . . . 47
3.1.2 State-of-the-Art . . . 48
3.1.3 Contributions . . . 49
3.1.4 System model . . . 50
3.1.5 Fault model . . . 50
3.1.6 Dynamic instruction replication and scheduling mechanism . . . 51
3.1.7 Cluster-based instruction replication and scheduling mechanism . . . 53
3.1.8 Coarse-grained and fine-grained dynamic instruction scheduling mechanism . 53
3.1.9 Evaluation . . . 54
3.2 Run-time data shuffling for NoC . . . 57
3.2.1 Context . . . 57
3.2.2 State-of-the-art . . . 57
3.2.3 Contributions . . . 58
3.2.4 System model . . . 59
3.2.5 Fault model . . . 59
3.2.6 Basic Bit-Shuffling method . . . 60
3.2.7 Region-based Bit-Shuffling method . . . 60
3.2.8 Evaluation . . . 61
3.3 Cross-layer reliability analysis for complex hardware designs . . . 63
3.3.1 Context . . . 63
3.3.2 State-of-the-Art . . . 64
3.3.3 Contributions . . . 65
3.3.4 Fault models through Technology and Circuit Analysis . . . 65
3.3.5 Error patterns through Gate-Level Analysis . . . 66
3.3.6 Vulnerability metrics through Microarchitecture-Level Analysis . . . 67
3.3.7 Evaluation . . . 68
3.4 Conclusions . . . 71

4 WCET- and fault-aware task deployment for multicore architectures **73**
4.1 Design-time mapping under different DVFS schemes . . . 73
4.1.1 Context . . . 73
4.1.2 State-of-the-Art . . . 74
4.1.3 Contributions . . . 75
4.1.4 System model . . . 75
4.1.5 General problem formulation . . . 76
4.1.6 Optimal solution . . . 76
4.1.7 Heuristic methods . . . 76
4.1.8 Evaluation . . . 77
4.2 Design-time mapping considering NoC routing . . . 81
4.2.1 Context . . . 81
4.2.2 State-of-the-Art . . . 81
4.2.3 Contributions . . . 82
4.2.4 System Model . . . 82
4.2.5 Problem Formulation . . . 84

4.2.6 Optimal approach 84
4.2.7 Heuristic method 84
4.2.8 Evaluation 85
4.3 Conclusions 87

5 Perspectives **89**
5.1 Impact of hardware faults on timing behavior 91
5.2 Real-time and reliable AI hardware accelerators 94
5.3 Workload-dependent aging and multiple reliability threats 98
5.4 Real-time and reliable emerging technologies 101
5.5 Conclusions 103

List of Figures

2.1 The structure of optimal decomposition-based approach for dependent tasks and AMP platforms with DVFS. . . . 18
2.2 The structure of accelerated optimal decomposition-based approach for dependent tasks over AMP platforms with DVFS. . . . 19
2.3 The structure of heuristic approach for independent tasks and SMP platforms with DVFS. . . . 20
2.4 The structure of heuristic approach for dependent tasks and AMP platforms. . . . 20
2.5 QoS and computation time gain of B&B, GA, OJTM, and HJTM with M, N and η varying. . . . 22
2.6 QoS and computation time gain of B&B, GA, JDQT, and AJDQT with M, N and η varying. . . . 23
2.7 Task execution considering *is*WCET . . . 25
2.8 a) Global [244] and b) fine-grained [C18] protection mechanisms to enforce partial order. . . . 25
2.9 Example of *is*RA-FG enforcing operation for four tasks on two cores (Curly brackets: ready vector, parentheses and arrows: notification vector). . . . 27
2.10 Example of *is*RA-FG relaxion operation for four tasks on two cores (Curly brackets: ready vector, parentheses and arrows: notification vector). . . . 29
2.11 Average performance gain of *is*RA-FG and *is*RA-GLO compared to TT execution, among all cores and experiments, for all configurations and benchmarks. . . . 31
2.12 Average performance gain of *is*RA-DYN compared to *is*RA-FG, among all cores and experiments, for all configurations, benchmarks and timing variability. . . . 32
2.13 Motivational example. . . . 35
2.14 Schematic representation of grammar rules . . . 37
2.15 Illustration example where CFG is obtained from C code. . . . 37
2.16 Run-time behavior among the master, high and low criticality tasks. . . . 39
2.17 Comparison of RWCET static and dynamic approaches w.r.t. the gain of the execution time of the low criticality tasks compared to the isolated execution and the number of active points. . . . 44

3.1 a) Assembly instructions and register dependencies. Corresponding execution by the b) compiler and b) DIRS approach [C12, J26] for an 4-issue VLIW. . . . 49
3.2 Execution based on a) compiler, b) DIRS [C12, J26], c) DIRS-CG [C11], d) instruction re-execution at a new slot [238] and e) DIS [C17, C15]. . . . 50
3.3 Original VLIW processor. . . . 51
3.4 VLIW datapath enhanced with a) DIRS [C12, J26] and b) DIS [C15, C17]. . . . 52
3.5 a) Coarse-grained and b) fine-grained decomposition of FUs. . . . 54

3.6 Performance degradation under DIRS-CB and DIRS-CNB. 55
3.7 Performance degradation, compared to fault-free execution, taking into account the status of FUs in a coarse-grained [C15] and fine-grained [C11] way. 56
3.8 NoC extended with the BiSu method. 58
3.9 Illustration of the R-BiSu technique for different region sizes. 59
3.10 Reliability efficiency and area overhead of BiSu technique and the Hamming code. . 61
3.11 Reliability efficiency and area overhead comparison. 62
3.12 Area and MSE Pareto front. 63
3.13 FLODAM cross-layer reliability analysis flow. 66
3.14 RISC-V core with 5-stage pipeline [213]. 68
3.15 SET distribution normalized to cell area and input sizes 69
3.16 Gate-level results of the RISC-V execution stage. 70
3.17 Vulnerability metrics. 70

4.1 Feasibility for all DVFS schemes for optimal solutions. 79
4.2 Energy consumption gain for all DVFS schemes for optimal solutions. 79
4.3 Computation time for all DVFS schemes for optimal approaches. 80
4.4 Feasibility for all DVFS schemes for optimal and heuristic approaches. 80
4.5 Energy consumption gain for all DVFS schemes for optimal and heuristic approaches. 81
4.6 Computation time for all DVFS schemes for optimal and heuristic approaches. . . . 81
4.7 An example of NoC-based multicore system. 83
4.8 The structure of heuristic approach. 85
4.9 Energy consumption of optimal approach compared to a) a single-path approach, and b) having the objective of minimizing energy consumption. 86
4.10 Comparison of optimal and heuristic approaches: a) feasibility, b) energy consumption, and c) computation time. 87

List of Tables

1.1 Classification of representative SoA approaches 5

2.1 Classification of representative task deployment approaches 15
2.2 Summary of task deployment problem formulations. 17
2.3 Summary of experimental set-up for task deployment approaches 21
2.4 Comparison of representative *is*WCET approaches 24
2.5 TMS platform and benchmark characteristics. 30
2.6 Benchmark timing variability 30
2.7 WCET controller overhead of *is*RA-GLO, *is*RA-FG and *is*RA-DYN approaches (cycles). 33
2.8 Comparison with representative risk-permissive approaches 34
2.9 Risk-permissive run-time adaptation controller time overhead (in cycles) 44

3.1 Comparison with representative SoA fault tolerant VLIW approaches. 48
3.2 Area and critical path delay overhead. 56
3.3 Comparison with representative SoA fault tolerant NoC approaches. 57
3.4 Comparison with representative reliability analysis approaches. 64
3.5 Relative area of RISC-V pipeline stages. 68
3.6 Time of gate-level analysis. 69
3.7 Time of microarchitecture-level analysis. 71

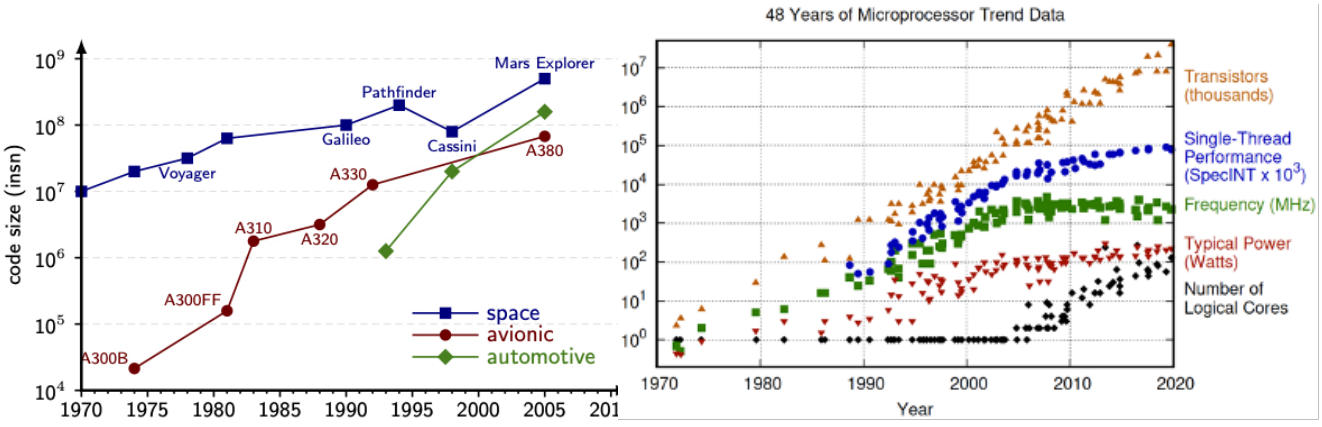
4.1 Representative DVFS task deployment approaches targeting energy minimization. . 74
4.2 Summary of problem formulations. 76
4.3 Summary of experimental set-up for real-time and reliable DVFS task deployment . 78
4.4 Classification of representative task deployment approaches 82
4.5 Summary of problem formulation in [C25]. 84

Chapter 1

Introduction and Motivation

1.1 Context

Industries are specialized in domains that rely 95% on embedded systems [67]: fly-by-wire systems are used in avionics [193] (Airbus, Thales, ONERA, etc.), navigation systems in space [193] (ESA, DLR, CNES, etc.), automatic braking systems in automotive [177] (Peugeot, Volkswagen, Scoda, etc.), insulin pumps in the health-care [157] (Siemens Healthineers, Philips Healthcare, etc.), embedded vision systems in surveillance [251] (Ivisys, Datalogic, IRIDA-Labs etc). Embedded applications are complex, imposing constraints to the systems that execute them [B2]. Particularly, *hard real-time* and *reliable* application execution, that delivers results of acceptable quality and under a maximum *energy budget*, must be guaranteed [192]. As applications are becoming more complex, their computational demands scale rapidly. As shown in Figure 1.1a, the code size of automotive, space and avionics applications has increased significantly within thirty years, e.g. more than three orders of magnitude in avionics [35], while future automotive and avionic applications will require higher computing resources [34]. To address these demands, architectures with multiple processing elements (cores) and application specific hardware accelerators are required. As shown in Figure 1.1b, a significant increase in the number of logical cores is observed in recent hardware platforms. Multicore and manycore architectures are able to concurrently execute a high volume of applications [136], while hardware accelerators are tailored to the needs of the application [110].



(a) Code increase in safety-critical industry [35].

(b) Evolution of platform complexity [223].

Hard real-time guarantees are required for critical embedded systems, and thus, the worst case of the system execution has to be considered during system analysis and design [151]. Overall, the real Worst Case Execution Time (WCET) of an application is generally unknown, and thus, an estimation of the WCET must be obtained. Such a WCET estimation has to safely upper bound the execution time of the application, and thus, it naturally overestimates the real WCET [179]. A main reason for the WCET overestimation is the unpredictable timing behavior of the system, originated from both application complexity and hardware platform complexity.

Regarding application complexity, the application code can have multiple different execution paths with different instructions and memory accesses, leading to different execution times, complicating WCET analysis. For instance, the final targets of computed branches, indirect addressing and dependencies on input parameters cannot be resolved during static timing analysis [179], whereas no guarantee exists that all executions paths can be tested when measurement-based WCET approaches are used [4]. Regarding hardware complexity, modern architectures have been enhanced with dynamic, history-based, hardware components, to improve average performance, e.g., cache memories and branch predictors [73]. However, such components have variable timing behaviour, leading to further inflation of the WCET estimations. Furthermore, modern hardware architectures have several cores and share resources among them, e.g. communication networks, memory hierarchy and controllers. Parallel execution of applications on the same platform may lead to concurrent accesses to shared resources [224]. These concurrent accesses add timing delays (interference), highly affecting applications' timing behaviour in a non-deterministic manner. This behavior introduces additional uncertainties that further increase the WCET overestimation, since the interference effects in time have to be safely bounded.

Overall, the timing behavior of multiple execution paths, dynamic hardware components and shared hardware resources has to be upper bounded in order to be safe, leading to overly pessimistic WCET estimations. For instance, WCETs estimations with interference on shared resources can be $\times 7$ larger than the WCETs estimations without interference [C8, 246]. This WCET overestimation leads to under-utilisation of the system and to the "one-out-of-m processors" problem [127], where the additional processing capacity, provided by multicore architectures, is negated by the WCET pessimism. As a result, sequential execution on a single core may provide better timing guarantees than any parallel execution, seriously undermining the advantages of using architectures with multiple cores. To reduce the WCET pessimism, run-time approaches are a promising solution, which take advantage of the information that becomes available only during the real system execution.

Reliable execution is under threat due to the increased fault susceptibility of modern electronic systems. Reliability threats, such as manufacturing process variation, aging and soft errors, depend on transistors size and are expected to significantly increase with transistors shrinking [241]. Nanoscale transistors are difficult to precisely manufacture, leading to erroneous variation in transistor physical and electrical parameters. Aging refers to frequency degradation, induced by shifting transistor threshold voltage [206]. Industrial standards mainly rely on introducing a safe margin (guard-band margins), in voltage or frequency, to mitigate process variation and aging [206, 43]. Soft errors occur due to environmental conditions, such as high temperature peaks and high-energy electromagnetic radiation [206] and their mitigation is very challenging due to their random and transient nature [167]. Due to this unreliable nature of ultra-scaled electronic systems, combined with the high density of modern architectures, the susceptibility of multicore architectures towards multiple reliability threats is inevitable [206] and faults will be occurring even under normal operation [130], which was not the case with the technology used a decade ago [98].

Typical reliability solutions, that rely on full hardware or software redundancy, usually entail

significant cost, latency and resource overheads, which are often not suitable for critical embedded systems, which are resource constraint. Hardware redundancy introduces significant overheads in chip area and energy consumption, whereas software redundancy accounts for notable performance degradation and acts agnostically to the underlying hardware [206]. To cost-effectively address these reliability threats, the system must be properly analysed in order to identify the most vulnerable software and hardware parts [206], and enhanced with selective fault tolerance and self-adaptation [43].

Meanwhile, the *energy consumption* of embedded systems has become a crucial factor. The amount of electronic devices is increasing day by day. Combined with the increased complexity of applications and the computation capabilities of modern hardware platforms, the amount of consumed electricity of electronic devices is significantly increased, negatively impacting the environment. To reduce this negative impact, energy efficient electronic devices should be designed and e-waste should be minimized [13].

Several approaches have been established to maximize system energy efficiency. The platforms have been enhanced with Dynamic Voltage and Frequency Scaling (DVFS), which is an adaptive management technique that optimizes energy consumption by simultaneously scaling down the processor supply voltage and frequency, during execution [274]. Acceptable errors in the application results are introduced using approximation for computation and communication [7] in domains that can tolerate approximated results, such as image processing, data mining, machine learning, etc. However, such approaches usually have a negative impact on execution time and reliability, e.g., reducing supply voltage/frequency increases the execution time and the transient fault rate [284].

To provide real-time, reliable and low energy execution over multicore architectures, efficient analysis and deployment approaches, along with run-time adaptation capabilities, are required. However, designing real-time, reliable and energy-efficient multicore embedded systems is perplexing and tedious process, as it consists of several interdependent NP-hard problems [B2] in a large multi-dimensional design space [124], with constraints and trade-offs among execution time, reliability, and energy efficiency. In this context, applying exhaustive Design Space Exploration (DSE) is prohibited. Trial-and-error approaches, based on the designers' experience, usually require several design iterations, being time-consuming without any upfront guarantee that the system will satisfy the application constraints [J6]. DSE methodologies are required with reduced exploration time leading to near-optimal solutions. Furthermore, as design-time approaches cannot take advantage of the information created during the system execution, they have to be combined with run-time approaches in order to provide further improvements.

1.2 State-of-the-Art (SoA) and positioning of current contributions

In this section we describe representative SoA approaches with respect to real-time, reliable and energy efficient embedded systems and position our contributions with respect to existing works. Table 1.1 classifies representative approaches from the SoA under the following categories:

- Regarding the focus of the approaches:
  - *WCET-Aware (WA)*: Is the system under study considering the worst-case during design and execution?

- *Fault-Aware (FA)*: Is the system under study susceptible to hardware faults? Hardware faults can impact the memories, the cores or the interconnections of the target platform. Their impact can be *permanent* or *transient*. Such hardware faults can impact the *functional behavior* and the *timing behavior* of the applications. Functional behavior refers to *denial of service*, i.e., no outcome is generated because the application is hanged or crashed, and to *binary correctness*, i.e., the application outcome is different than expected [212]. Timing behaviour refers to an application execution time that is different compared to the fault-free execution. Note that, a denial of service due to application hang is identified when the application execution time exceeds a, usually high, threshold.

- Regarding the computing platform:
  - *Single-Core (SC)* or *Multi-Core (MC)*: The computing platform under study consists of a single core (or a standalone hardware accelerator) or multiple cores?
  - *HW design (SC):* Does the work designs any customized hardware component for the computing platform, i.e., *Memory (M)*, *Core (C)*, *Hardware Accelerator (A)*, or it extends a given computing platform with additional hardware *Control Mechanisms (CM)?*

- Regarding the approach:
  - *Design Space Exploration (DSE)*: Is any DSE adopted to drive the design of the system ?
  - *Run-time (RT) adaptation*: Is any run-time adaptation employed ?
  - *Design layer*: At which design layer the DSE or RT-adaptation is applied, i.e., *Application Layer (AL)*, *Deployment Layer (DL)*, and *Hardware Layer (HL)*? Note that, DL can refer to tasks, data or instructions depending on the approach (e.g., task mapping to cores, data mapping to registers, memory and buffers, or instruction mapping to function units).

1.2.1 WCET-Aware (WA) techniques

These techniques derive mainly from the real-time community, having the requirement of timing guarantees based on WCET estimations, and the majority considers considers fault-free computing architecture. WCET-aware approaches that consider hardware faults are presented in Section 1.2.3. This section presents some representative WA techniques that: a) *guarantee real-time system execution*, and b) *estimate WCET*, whereas further approaches can be found in surveys [151, 73, 72].

a) Guarantee real-time system execution: These WA approaches can be further classified to the following main categories: *i) Interference-pessimistic*, that assume that interference will always occur at any access to shared resources, *ii) Interference-free*, that employ upfront spatial-temporal isolation to prohibit the occurrence of interferences, *iii) Interference-controlled*, that restrict the number of allowed interferences, so as to obtain tighter WCET, and *iv) Risk-permissive*, that allow design decisions that may lead to timing violations, such us allowing interferences to occur or using less pessimistic WCET estimations, but watch at run-time for risks that can lead to timing violations, and take actions in order to mitigate them, if needed.

Table 1.1: Classification of representative SoA approaches

Reference	Context		Platform		HW design				DSE			RT adapt		
	WA	FA	SC	MC	M	C	A	CM	AL	DL	HL	AL	DL	HL
[288]	✓		✓		✓	✓								
[266]	✓		✓		✓					✓				✓
[191]	✓		✓					✓		✓				
[22, 41, 24, 68, 185, 250]	✓		✓							✓			✓	
[23, 40, 229, 111, 26, 86]	✓		✓										✓	
[152, 209]	✓			✓										
[214, 215, 245, 264, 285]	✓			✓						✓				
[C10, J15, J16, C16]	✓			✓						✓				
[J14, W1, C7, C8, C9]	✓			✓									✓	
[C18, C21, 179, 282]	✓			✓									✓	
[75, 12, 162, 246, 244]	✓			✓						✓			✓	
[281, 180, 142, 25]	✓			✓						✓			✓	
[184]	✓			✓	✓	✓				✓				✓
[261]	✓			✓	✓	✓			✓	✓				✓
[C13]	✓			✓	✓	✓			✓	✓				
[5, 240]	✓			✓										
[254, 104, 78]	✓		✓											
[36]		✓	✓							✓				
[129, 112]		✓	✓					✓		✓				
[C11, C12, C15, C17, J26, 135, 238]		✓	✓					✓		✓			✓	
[232, 235, 233, 59, 60, 249, 128, 174]		✓	✓					✓					✓	
[20]		✓	✓					✓						
[234]		✓	✓			✓				✓			✓	✓
[194, 120, 77]		✓	✓				✓							
[216]		✓	✓				✓			✓				✓
[158, 161, 143, 175]		✓		✓						✓				
[31]		✓		✓							✓			
[154, 8, J23, C24, C22, PP5, 56, 280]		✓		✓				✓					✓	
[88, 63, 125, 83]		✓		✓				✓					✓	✓
[82, 166, 173]		✓		✓				✓						
[243]		✓		✓		✓							✓	
[C26, C20, 117, 208, 168, 123, 265] [269, 47, 93, 236, 262]		✓												
[119, 115, 188, 9, 53]	✓	✓	✓							✓			✓	
[145]	✓	✓	✓					✓					✓	
[230]	✓	✓		✓									✓	
[C19, C23, J25, J24, PP7, 276, 113, 231]	✓	✓		✓						✓				
[189, 197, 96, 283, 103, 101, 275, 90]	✓	✓		✓						✓			✓	
[227]	✓	✓		✓									✓	
[48]	✓	✓		✓				✓					✓	
[105, 248, 52, 50, 51]	✓	✓	✓											
[247]	✓	✓		✓										
[3, 106]	✓	✓	✓					✓						✓

i) Interference-pessimistic: These approaches consider the worst-case interference at each access to a shared resource. This category also includes approaches that do not explicitly discuss about interference, but since they imply real-time constraints, the WCET estimation has to be pessimistic in order these approaches to be able to provide timing guarantees. The majority of these approaches focus on task scheduling usually using priority-based heuristics and schedulability analysis. For instance, a work-conserving greedy scheduling method is proposed for dependent tasks on heterogeneous multicore platforms [264] and a scheduling method using priorities based on the critical path, the early predecessors and the path length, is proposed for dependent tasks on homogeneous platforms [285].

The majority of interference-pessimistic approaches focus on task deployment and schedulability analysis mainly considering precise tasks sets. Few approaches consider imprecise computation tasks sets and mainly focus on in-time execution under a given architecture configuration. We extend the SoA by proposing a set of optimal and heuristic design-time task deployment approaches for imprecise tasks in order to improve the QoS, not only under real-time constraints, but also under energy budget constraints on multicore architectures with DVFS capabilities [C10, J15, J16, C16], described in Section 2.1.

ii) Interference-free: These approaches upfront isolate the shared resources among critical tasks and between critical and less critical tasks. For instance, a scratchpad memory is designed for critical tasks offering spatial isolation to non-overlapping addresses, a scheduling algorithm is proposed to assign tasks to partitions and a run-time scratchpad partitioning is applied between two successive critical tasks [266]. FlexPRET [288] processor provides hardware-based isolation to critical tasks, while a thread scheduler enables the assigning of idle cycles to low criticality tasks in a round-robin fashion. MERASA [184] is a multicore architecture based on simultaneous multithreading cores, providing full isolation within a core and time bounded interaction between critical tasks of different cores. A bank remapping unit is proposed to support dynamic cache partitioning. ParMERASA [261] is based on MERASA multicore architecture and aims at WCET-aware application parallelization. The PRedictable Execution Model (PREM) [191, 215] executes the tasks in three phases (read data, execute, write results), and provide contention-free execution by allowing parallel execution only between communication and computation. A hardware controller buffers incoming packets from the network [191]. This category is currently not our focus.

iii) Interference-controlled: The knowledge of the mapping solution enables the computation of a better upper bound on the number of interferences a task can have. As a result, a tigher WCET is computed, which is, however, interference-sensitive (*is*WCET), i.e., it is valid only for the given task mapping solution. For instance, the analysis of [209] estimates tighter WCETs by computing the interference from tasks running in parallel under a given schedule. Algorithmic optimisations and task deployment is explored over the invasIC multicore platform and the corresponding *is*WCETs are computed through an iterative method in ARGO [C13]. Task deployment and *is*WCETs are computed using an ILP method in [214] and a two-step approach in [244, 246, 245]. Tighter WCETs are derived by exploiting the knowledge of memory budget assignments to the cores [152]. A preliminary analysis of the task resource usages allows off-line partitioning of the resources. A monitor observes at run-time the real task resource usages, and suspends the task that overtakes its allocated capacity [180]. The extension of [179] allows dynamic changes in the resource partitioning, when resources are under-utilized. In [281], the budget of each task with respect to the number of memory accesses is decided in order to improve schedulability, combined with a controller to prioritize memory accesses. The extension [282] regulates the best-effort task accesses, when possible.

The majority of *is*WCET approaches are applied at design-time and follow a time-triggerred execution. We extend the SoA by proposing a set of run-time approaches to improve the system performance by allowing an earlier task execution, while preserving the timing guarantees of a given interference-sensitive mapping solution [C18, C21], described in Section 2.2.

iv) Risk-permissive: These approaches are based on run-time mechanisms to watch for timing violations during execution. The majority of approaches focuses on timing violations due to underestimation of the task WCET. Different WCET static analysis is employed for different levels of criticality, and, thus, different WCETs exist for a single task. As the criticality level is increased [41], the WCET has a greater and safer value. Such approaches are usually based on the notion of temporal isolation among tasks of different criticalities, where tasks of lower criticality should not adversely affect tasks of higher criticality. Task mapping is performed in order to meet the real-time requirements at each criticality level. In a two level mixed-criticality scheduling [12] and its extension to several levels [162] for multicore architectures, the low criticality WCET is optimistically assigned at each task. At run-time, if the execution of a high criticality task exceeds its low criticality WCET, a switch occurs to high criticality mode. The low criticality tasks are usually dropped [142, 22, 23, 41], their priority or execution time requirements are reduced [24], or their periods are extended [40]. Other strategies explore slack to postpone mode switch [229, 68, 75, 111, 26, 185] and to switch back to low criticality mode [12, 162, 86]. Several global and partitioned scheduling algorithms of this category are presented in [25]. Similar approaches exist for single cores, e.g., decreasing computation demands of lower criticality task by increasing their service intervals, i.e., periods [250] and considering several criticality levels [86].

Existing approaches are based on design-time estimations of the WCET. We extend the SoA by computing new estimations of the remaining WCET at observation points during execution [J14, C9, C8, C7, W1], i.e., the WCET required from an observation point until the critical task ends, without executing low criticality tasks. The system starts execution with high criticality and low criticality tasks. At run-time, a safe condition uses the remaining WCET to verify whether the concurrent execution is still allowed to continue or the low criticality tasks should be suspended. More details are provided in Section 2.3.

a) Estimate WCET: WCET estimation is a key element to provide timing guarantees in the real-time domain. WCET estimation is performed through safe measurements, based on application execution, or static analysis of the programs. For instance, a number of static analysis methods have been proposed, such as [254, 104], focusing on caches, and measurement-based approaches, such as [240, 78, 5]. A more detailed description of WCET estimation methods and tools is available in surveys, such as [268, 73]. WCET approaches without faults are currently not our focus.

1.2.2 Fault-Aware (FA) techniques

This section presents some representative FA techniques, whereas surveys exist for error detection and correction [94, 242, 196] and reliability-aware design [71]. We focus on techniques mainly introduced by the embedded system design community, with particular interest on approaches that address reliability issues on the computation on processing elements and communication over Network-On-Chips (NoC). Therefore, approaches that propose information redundancy to deal with errors in the memories are not included in this short SoA. Furthermore, we are not interested in hardening techniques which increase the size of transistors, since this type of techniques reduces the system frequency, and thus, limits the performance capabilities of the processor [79]. We present two main categories: *a) FA approaches that enhance the system reliability*, and *b) FA approaches*

that analyse the system reliability. Note that, the majority of fault tolerant techniques usually focus on the impact of faults on the functional behavior and do not consider worst-case aspects.

a) Enhance system reliability: These approaches can be further classified depending on whether they are applied for the processing elements or the data traversing the NoC. Note that, approaches for memory are not the focus of this research. The first category can be further refined on *i) Instruction-level* approaches, applied in a fine-grained manner considering the internal components of a processor, and *ii) Task-level* approaches, applied in a coarse-grained manner among processors. NoC approaches are categorized as *iii) Packet-level*. Each category can be further refined regarding the application of the fault tolerant approach, i.e., *i) Full*, providing fault tolerance to all instructions, tasks, or packets, and *ii) partial*, that provide fault tolerance to a subset of instructions, tasks or packets. We have proposed a detailed classification of fault tolerant approaches in [J21].

i) Instruction-Level Fault Tolerance (ILFT) can be achieved through software, a.k.a., instruction replication or re-execution, and through hardware, i.e., resources replication.

Instruction replication/re-execution can been full or partial and it can performed by software, hardware and hybrid approaches. Instruction replication is usually applied over platforms that have several function units, such as Very Long Instruction Word (VLIW) processors. Full instruction redundancy replicates all instructions. For instance, the compiler duplicates all instructions and inserts comparison instructions [36]. A hybrid approach replicates instructions using the compiler and inserts a hardware mechanism to perform the comparison [238]. A full hardware mechanism dynamically duplicates the instructions and applies re-execution in case of an error [233]. Partial instruction redundancy replicates a subset of the instructions. For instance, the compiler does not replicate control-flow and store instructions [158] and selects which instructions to duplicate based on the instruction fault masking capability [161]. Hybrid approaches exist, e.g., the compiler maximizes the number of duplicated instructions under a bound [112, 129], combined with hardware-based comparison. The compiler selects the instructions to be duplicated based on temporal and physical vulnerabilities. This selection is encoded in the instructions, which is decoded by a hardware mechanism that performs the instruction duplication at run-time [135]. Hardware approaches replicate an instruction when the coupled issue is idle [232] or based on an ILP threshold [235] for coupled VLIWs, and on the size of the additional queues in a configurable VLIW [234].

Software approaches increase the code size and memory requirements, whereas existing hardware approaches are applicable on homogeneous VLIW processors. We extend the SoA by a set of hardware mechanisms for heterogeneous VLIW, which fully triplicate instructions and re-schedule faulty instructions, while performing instruction scheduling at run-time using priority scheduling algorithms [J26, C11, C12, C15, C17], described in Section 3.1.

Resource replication to support ILFT is performed in hardware inside the processor and it can be full or partial. Full resource replication replicates all instances of the resource type to be protected. For instance, function units of a processor are duplicated [174] and two pipelines execute instructions in lock-step with time diversity [243]. Partial resource replication can be achieved by selective fault tolerance approaches, e.g., the most radiation-sensitive circuit gates are identified and replicated [216], reduced duplication is applied based on structural susceptibility analysis, logical weight of the arithmetic circuit outputs and approximated structure [77].

Note that, *hybrid approaches* also exist, where instruction replication/re-execution and resource replication are combined. For instance, when enough resources do not exist to execute twice the instructions, spare FUs are used [59, 60]. Resource replication and hybrid approaches have not been up to now our focus.

ii) Task-Level Fault Tolerance (TLFT) can be achieved through software, using task replication or

re-execution, and through hardware, i.e., core replication.

Task replication/re-execution can been full or partial, whereas task re-execution can be performed with or without check-pointing. For instance, N instances of the same task are created and guarded within a fork and a voter task, and the tasks are re-mapped during execution, when faults occur on a core [154], whereas the number of task instances is decided based on a variation-aware core and task vulnerability scheme [143]. Other approaches take into account the communication cost during task replication and mapping [175].

Core replication supports TLFT in hardware by providing additional, either identical or diverse, cores/hardware designs dedicated for the redundant execution of the tasks. For instance, three cores are used to execute the same workload and compare their final results [249], two processor units are fully synchronized executing identical instruction streams [128], and complete hardware design replication occurs in [120], where an identical copy of a circuit is inserted and the its output is compared with the original one. Diverse core replication can be achieved through reduced precision redundancy inserting the same, but lower precision, copies of arithmetic circuits [194], and use of smaller in-order cores [8].

TLFT approaches that do not take into account real-time aspects are excluded from our focus.

iii) Packet-Level Fault Tolerance (PLFT) can be achieved by reconfiguratian and redundancy. Through reconfiguration, the routing path is changed to avoid faulty paths or faulty region [88, 63], spare resources are used to replace the faulty ones [125, 83, 31]. Through redundancy, the circuit is replicated [82, 166] and extra information is inserted, e.g., additional bits inside messages using Error-Correcting Codes (ECCs) [56, 280]. Few approaches reduce the impact of faults for Network-on-Chip (NoC), e.g., by assigning the Most Significant Bits (MSBs) on the borders of the bus [173].

The majority of existing approaches focus on mitigating the impact of faults, while a few approaches focus on reducing the fault impact when approximations are acceptable. However, existing fault correction approaches cannot efficiently address several permanent faults on NoC, due to their high hardware costs, while design-time approaches cannot deal with new occurring faults. We extend the SoA with a low cost hardware mechanisms that performs shuffling of the bits inside the packet flits at run-time, in order to reduce the impact of multiple faults [C22, C24, J23, PP5]. Further details are given in Section 3.2.

b) Analyse system reliability: Reliability analysis can be performed by injecting faults into the system and observing its behavior. Fault injection can be done by hitting the real system with a radiation beam and by simulating the impact of injected faults. Approaches for reliability analysis through simulation either focus on analysing the radiation impact at the lower hardware design layers, i.e., technology and circuit layers or inject faults at higher hardware and application layers to characterize the system execution. Radiation analysis at technology and circuit layers can take into account the circuit layout, the fabrication technology, the radiation and operational environments [208, 117]. For reliability analysis at higher layers, fault injection is performed at different abstraction layers. For instance, a fast, but less accurate, fault injection at application level [168]. Hardware fault injection approaches insert single-bit faults [265, 123] and multiple-bit flips [269, 93] in sequential logic and in combinational logic [47, 236]. Few works consider both categories to analyse the radiation impact on the system execution, e.g., considering single-bit faults in memory components [262]. However, faults in smaller sequential and combinational logic cannot be neglected and should be included in the reliability analysis.

Existing reliability analysis frameworks are based either on low hardware levels to characterize the impact of radiation to the transistors and cells, without analysing the propagation of faults to the system execution, or on vulnerability analysis at higher hardware and software layers usually assuming uniform distribution on fault models. We extend the SoA by proposing a cross-layer reliability analysis framework for multiple-bit faults in sequential and combinational logic, based on realistic fault models for radiation, which are propagated at the gate, microarchitecture and application level [C20, C26], described in Section 3.3.

1.2.3 WCET-aware (WA) and Fault-Aware (FA) techniques

This section presents some representative WAFA techniques, whereas further approaches can be found in surveys, such as [45]. Existing approaches mainly focus on typical hardware faults, i.e., soft errors and permanent faults. We present two main categories: a) WAFA approaches that *enhance the system reliability under real-time constraints*, and b) WAFA approaches that *estimate the WCET under hardware faults*.

a) Enhance system reliability under real-time constraints: To deal with typical hardware faults, the majority of real-time approaches focus on task-level fault tolerance taking into account their overhead in time, implemented through task-replication/re-execution potentially with task migration, and thought execution using lock-step redundant cores.

Task replication/re-execution is applied fully or partially in WAFA techniques. For instance, full task replication techniques are combined with typical schedulability analysis [115, 188, 9, 32] and probabilistic worst-case schedulability analysis for active and passive replicas [189]. A set of schedules is synthesized off-line, using task re-execution as a fault tolerant method. Based on the occurrence of faults, the scheduler selects at run-time which schedule to apply [119]. Re-execution with checkpointing is used in [53]. Task check-pointing and roll-back recovery for transient faults combined with task migration for permanent faults is achieved through heuristics [230] and tabu search optimization [231] to maximize the probability of meeting the deadlines of soft tasks under transient faults. An evolutionary algorithm considers task replication and re-execution creating static schedules under a maximum amount of transient faults for real-time systems [113].

Partial task replication is achieved through a reliability-aware co-synthesis framework, which performs allocation and scheduling by selective task duplication based on the task criticality rank [276]. On top of the real-time constraints, task replication/re-execution approaches exist that focus on minimizing the energy consumption, e.g., scheduling recovery tasks to be executed if an error is detected [197, 96, 283], deciding and scheduling the number of task replicas to meet reliability constraints [103, 101, 275, 90] and taking into account communication cost [48].

Existing WAFA task replication approaches focus on meeting real-time constraints and reliability constraints. We extend the SoA by performing partial task replication based on the reliability constraint of tasks, considering transient faults, and explore the impact of different common DVFS schemes for shared-memory in multicore platforms [C19, C23, J25, J24, PP7] and NoC-based multicore platforms [C25], with the goal of minimizing the energy consumption, described in Chapter 4.

Core redundancy with lock-step execution is usually used in WAFA approaches. The replicated cores can execute the tasks in tight lock-step, where cores execute the same instruction and compare the outputs cycle by cycle, and loose lock-step, where redundant execution is not synchronised on the different cores [20]. For instance, tasks are executed at different times on redundant cores and the progress is compressed into an external state, called fingerprint, and at regular intervals, the original and redundant fingerprints are compared, providing a bounded error detection latency [145].

Core redundancy can be applied also at system level, e.g., redundant multiprocessor system-on-chip are used and interconnected via a dependable and redundant off-chip net [227]. Currently, we have not worked on this aspect yet.

b) Estimate WCET under hardware faults: Existing approaches focus on hardware faults in cache memories, while the rest of the architecture is assumed fault-free [45]. The goal is to estimate the timing impact to WCET, by accounting for the hardware degradation due to the presence of faults. For instance, the probability of an SRAM cell to be faulty is used to evaluate the additional cache misses and taken into account during WCET estimation [105], while a measurement-based approach provides the WCET impact, when cache lines are disabled due to permanent faults [248]. Approaches extend the aforementioned works to incorporate the timing impact of inserted fault tolerant techniques to detect, correct or mitigate faults [52, 50, 51, 247]. Other approaches focus on mitigating the hardware degradation, due to occurring faults, using redundant hardware. As a result, the timing impact of faults on WCET is mitigated and the timing characteristics of hardware are maintained, leading to WCET estimations close to fault-free WCET estimations, despite the presence of faults. For instance, timing analysis is provided considering a reliable victim cache to replace faulty entries [3], an extra reliable cache way per set and a shared reliable buffer [106]. Although our research up to now has not addressed this category, we will tackle hardware faults in processing elements, as described in our near-future perspectives in Chapter 5.

1.3 Conclusions

This research focuses on the domain of designing real-time and reliable embedded systems. We presented the main challenges and presented an overview of the state-of-the-art, regarding WCET-aware and fault-aware approaches, applied at design-time and at run-time at different abstraction layers, and positioned our contributions. The next three chapters will describe in more details our contributions. More precisely, the proposed WA approaches are described in Chapter 2, the contributions regarding FA approaches in Chapter 3, and the proposed WAFA approaches in Chapter 4. Chapter 5 will present the main limitations of existing works, motivate our future work and describe the future research directions.

Chapter 2

WCET-aware task deployment for multicore architectures

This chapter summarises our contributions focusing on real-time systems on multicore architectures with hard timing guarantees, and thus, the system execution must guarantee that tasks are completed before their respective latency requirements (a.k.a. deadlines). More precisely, section 2.1 presents our contributions regarding optimal and heuristic design-time approaches to deploy Imprecise Computation (IC) tasks, characterized with pessimistic WCET regarding interferences, over multicore architectures, under energy supply and hard real-time constraints. These works have been performed during the Postdoctoral period of Lei Mo. Section 2.2 describes the contributions regarding run-time approaches to adapt a time-triggered schedule, which has been created considering WCET that are interference sensitive (*is*WCET) for hard real-time applications. Section 2.3 presents our contributions regarding run-time approaches that allow interferences to occur and adapt the task execution if a risk exist, through system mode switch, for mixed-critical systems. These works have been proposed during my post-doctorate period and tenured-track period in TARAN. Section 2.4 summarises the aforementioned contributions.

2.1 Interference-pessimistic design-time mapping for IC tasks

2.1.1 Context

In several real-time application domains, less accurate results, computed before the deadline, are preferable than accurate, but too late, results [17]. This is due to the fact that a real-time application has to provide a result before its deadline. When not enough time is available, approximate results are acceptable, as long as the minimum acceptable Quality-of-Service (QoS) is satisfied and the results are provided in time [65]. For instance, in audio and video streaming, frames with a lower quality are better than missing frames. In radar tracking, an estimation of target's location in time is better than an accurate location computed too late. In control loops, an approximate result, produced by a control law, is more preferable as long as the controlled system, e.g., cruise control system, remains stable. In these domains, a task can be logically decomposed into a mandatory subtask and an optional subtask [17, 277]. This decomposition is typically modeled by the IC task model [144]. The mandatory subtask must be completed before the deadline in order to generate the minimum acceptable quality, i.e., the baseline QoS. The optional subtask refines the obtained result in order to increase the baseline QoS. For instance, in a real-time video application, at each

period, an imperfect, but acceptable, quality image is initially produced from the received data. Then, this image can be further refined depending on the available resources [17]. Similar applications can be found in many other areas, e.g., mobile target tracking, real-time heuristic research and control engineering [277, 171]. In order to compute the increase in QoS that an optional subtask provides, each task is associated with a QoS function. The most realistic and typical functions for QoS modeling are the linear [219, 279, 277, 286] and the general concave [278, 66, 218, 16, 17] functions, since they can adequately capture the behavior features of many application areas.

At the same time, the system energy consumption has become an important concern. To enable energy efficiency, the platforms have been enhanced with the capability of dynamically scaling their voltage and frequency during execution to balance system performance and energy savings [274].

In this context, the longer an optional subtask is executed, the higher QoS is achieved. However, more energy is consumed and more time is required for the optional subtask execution. Furthermore, executing a task with lower processor voltage and frequency leads to a reduction of the energy consumption and an increase of the execution time of the task. In order to maximize the quality of the application results, and at the same time meet the constraints on energy consumption and real-time execution, proper task deployment approaches are required that can efficiently exploit both the platform characteristics and the application's tolerance to approximate results.

2.1.2 State-of-the-Art

The majority of the deployment approaches on multicore processors focuses on precise computation tasks [139, 54]. Usually these approaches aim at minimizing the energy consumption under real-time constraints. On the contrary, deployment approaches that focus on tasks that tolerate approximation, such as IC tasks [144], aim at maximizing the Quality of Service (QoS) under energy and/or real-time constraints. The way IC tasks are deployed on a multicore platform is decided mainly by three deployment factors. The first factor is the task mapping, which refers to both the task allocation (on which core each task is executed) and the task scheduling (when each task starts execution). The second factor is the decision of the voltage and frequency of the core when it runs a specific task. The third factor is the adjustment of the optional part of each task. Table 2.1 provides several representative IC task deployment approaches from the literature and positions our contributions. Overall, task deployment approaches can be classified based on whether: 1) the tasks are Dependent (Dep.) or Independent (Indep.), 2) the platform is characterised as Symmetric Multicore Processor (SMP) or Asymmetric Multicore Processor (AMP), 3) whether DVFS, task allocation to processors (Alloc.) or task migration (Migr.) is under study, 4) whether Real-Time (RT) or Energy (E) constraints are taken into account, and 5) the solution method is optimal (O) or heuristic (H). Note that, AMPs provide heterogeneity to deal with application diversity, as they consist of cores that differ in microarchitectural features, such as pipeline design, issue/fetch width, and cache hierarchy, and not merely in frequency/voltage, as SMPs [159].

The majority of IC deployment approaches on multicore architectures propose heuristics, while they do not consider concurrently the three aforementioned deployment factors. Regarding independent IC tasks, some approaches define upfront the frequency of the processors, and solve the task allocation and the optional part adjustment, e.g., through a heuristic where the two problems are solved in sequence for AMP [286, 267] and an optimal approach for SMP [17]. Regarding dependent IC tasks, some approaches assume that the task allocation is given upfront, and thus, address only the task scheduling, frequency assignment and optional part adjustment for SMP [279] and AMP [277]. Except single objective optimization, other existing approaches have a bi-objective optimization where the energy consumption is minimized and the QoS is maximized,

Table 2.1: Classification of representative task deployment approaches

Reference	Task				Platform			Constraints		Solution	
	Dep.	Indep.	Alloc.	Migr.	AMP	SMP	DVFS	RT	ES	O	H
[286, 267]		✓	✓		✓			✓	✓		✓
[17]		✓	✓			✓		✓		✓	
[171]		✓	✓			✓	✓	✓			✓
[277]	✓				✓		✓	✓	✓		✓
[279]	✓					✓	✓	✓	✓		✓
[156]	✓	✓	✓			✓	✓	✓			✓
[C10]		✓	✓			✓		✓	✓	✓	
[J15]		✓	✓			✓	✓	✓	✓	✓	✓
[J16]	✓		✓		✓		✓	✓	✓	✓	
[C16]	✓		✓	✓	✓			✓	✓	✓	✓

but without restrictions on the energy supply, e.g., using genetic algorithms and particle swarm optimization for independent and dependent tasks [156] and heuristic approaches for independent tasks on SMP [171]. Overall, the QoS-aware task mapping for multicore platforms is still an open issue, since there is no optimal polynomial-time solution [171].

The aforementioned QoS-aware task mapping methods, except [17], employ heuristics to find the near-optimal solutions. Although heuristics can provide feasible solutions in a short amount of time, they do not provide any bounds on solution quality, and are sensitive to changes in the problem structure and parameters.

2.1.3 Contributions

We propose a decomposition-based approach able to provide the optimal solution, with reduce solution time than typical optimal solvers, for independent IC tasks executed on a SMP [C10]. To further improve the energy efficiency, the aforementioned approach has been enhanced with DVFS capabilities [J15]. Furthermore, a heuristic is proposed to obtain near-optimal results with a negligible solution time and less sensitivity to the problem parameters. As a next step, we have extended our decomposition-based solution for dependent IC tasks and heterogeneous multicore platforms and proposed an accelerated, but still optimal, version of our decomposition-based method [J16]. Last, optimal and heuristic approaches considering task migration are proposed in order to take advantage of AMPs, such as the big.LITTLE platform [C16]. Note that, the extension of the optimal deployment methods from dependent to independent tasks and from SMPs to AMPs is not straightforward, as additional usually non-linear constraints are introduced into the problem.

In the next paragraphs, we summarize the task and platform models considered in our contributions and provide a general description of the proposed problem formulations and solutions. For the exact mathematical formulations for each problem, please refer to the corresponding articles [C10, J15, J16, C16].

2.1.4 System model

We consider an interference-pessimistic set of IC tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_i, \ldots, \tau_N\}$. Each task τ_i is described by a tuple $\{o_i, M_i, O_i, d_i\}$, where O_i is the upper bound on the number of possible optional cycles, i.e., $0 \leq o_i \leq O_i$. Therefore, the total length of task τ_i, measured in execution cycles, is $M_i + o_i$. As we require to guarantee the deadline of the task, the M_i and O_i are measured in Worst Case Execution Cycles (WCEC) [278], assuming the worst case for interferences. Each task has

a relative deadline d_i before which both the mandatory and the optional subtasks of τ_i must be completed. The relative deadline is defined as the time interval between the start of a task and the time instance when the deadline occurs [116]. Without loss of generality, we consider that tasks are released at the time 0. For dependent tasks [J16, C16], the task set is modeled by a Directed Acyclic Graph (DAG) called $G(V, E)$, where vertexes V denote the set of tasks to be executed, while edges E describe the data dependencies between the tasks. A task starts its execution when all its preceding tasks have been completed. Thus, the tuple is extended with t_i^s, which represents the start time of task τ_i. Note that t_i^s is an unknown variable, which is determined by task deployment decision. The deployment is performed within a scheduling horizon H. We use the linear function model, i.e., the system QoS increases uniformly with the optional subtask execution.

We consider a multicore platform with M processors $\{\theta_1, \ldots, \theta_k, \ldots, \theta_M\}$. Each processor θ_k $(1 \leq k \leq M)$(in [C10, C16]) is characterized by a given supply voltage and frequency pair (v_k, f_k). Furthermore, for asymmetric platforms in [C16], the M processors are divided into R clusters $\{\mathcal{W}_1, \ldots, \mathcal{W}_R\}$. Each cluster $\mathcal{W}_r$ consists of a set of symmetric processors, which have the same frequency characteristics (e.g., minimal, maximal and operating frequencies). When DVFS is taken into account in [J15, J16], each processor θ_k has L different Voltage/Frequency (V/F) levels $\{(v_1, f_1), \ldots, (v_L, f_L)\}$. Task-level DVFS is considered [278, 55], i.e., the frequency of a processor stays constant for the entire execution of a task, whereas it can be modified among task executions. When the processors are symmetric [C10, J15], the WCET of task τ_i, when it is executed at frequency f_k on processor θ_k, is given by $\frac{M_i+o_i}{f_k}$. When the processors are asymmetric [J16, C16], the parameter $\lambda_{ik} \in (0, 1]$ [139] is introduced to describe the execution efficiency of processor θ_k, when it executes task τ_i. Therefore, the WCET of task τ_i with frequency f_k on processor θ_k is $\frac{M_i+o_i}{\lambda_{ik} f_k}$.

Processors can operate in two modes: *idle* and *active*. A processor is said to be in the active mode, if the processor currently executes a task. Otherwise, it is in the idle mode. The power consumption of a processor θ_k is expressed as $P_k^c = P_k^s + P_k^d$, $1 \leq k \leq M$, where $P_k^s = C_k^s v_k^{\rho_k}$ is the static power of the processor ready to execute (being either on the active or idle mode), $P_k^d = C_k^d f_k v_k^2$ is the dynamic power of task execution, and C_k^s, ρ_k and C_k^d are constants depending on processor type. It is assumed that when a processor has no task to execute, it goes into idle mode. The transition time and energy overhead is considered very small compared to the time and energy required to execute a task and are assumed to be incorporated into the execution time and energy of the task. This power consumption model is widely adopted by existing works [139, 54].

Furthermore, the system is energy-constrained in the sense that it has a fixed energy budget E_s that cannot be replenished during the scheduling horizon H [286]. Taking the available energy E_s into account, the system operation can be divided into three states: 1) *Low*: the supplied energy E_s is insufficient to execute all the mandatory cycles $\{M_1, \ldots, M_N\}$, 2) *High*: the supplied energy E_s is sufficient to execute all the mandatory and optional cycles $\{M_1 + O_1, \ldots, M_N + O_N\}$, and 3) *Medium*: all the mandatory cycles are ensured to finish, while not all the optional cycles can complete their executions. We focus on the medium state.

2.1.5 General problem formulation

Given a set $\mathcal{T}$ of IC tasks, our goal is to map $\mathcal{T}$ on M processors, such that the overall system QoS is maximized, under task real-time and energy budget constraints. To achieve that, we determine 1) which processor should the tasks be executed on (task-to-processor allocation), and 2) how many optional cycles should be executed (optional task adjustment). Furthermore, when we study platforms that support DVFS, we also define 3) what frequency should be used for the tasks (frequency-to-task assignment). When we study set of dependent tasks, we define 4) when should

the task start (task scheduling). Last, when task migration is allowed, the migration of task is performed among different clusters, as the processors in the same cluster are symmetric. Overall, a processor is able to execute one task at a time instance (task non-overlapping constraint), the tasks should finish before their deadline (real-time constraint) and consume no more than the available budget (energy budget constraint). The above task mapping problems have multiple constraints, and in order to formulate and solve them we have to combine binary and continuous variables. Table 2.2 summarizes the constraints and the variables used in each contribution, along with the type of the proposed problem formulation and the proposed solutions. As Mixed-Integer-Non-Linear Programming (MINLP) problems are difficult to solve, we equivalently transform them to Mixed-Integer-Linear Programming (MILP). We adopt the idea of variable replacement [54] to eliminate the nonlinear items related to the optional cycles and the frequency assignment in [J15, J16] and the task migration in [C16]. To achieve that we introduced continuous auxiliary variables to replace the non-linear items. Then, the relevant constraints are modified accordingly. By doing so, the MINLP is transformed to an equivalent MILP.

2.1.6 Optimal decomposition-based method

We propose an optimal algorithm to solve our problem formulation when binary and continuous variables are coupled with each other linearly. The key idea comes from Benders decomposition, which is an effective method for solving MILP with guaranteed global optimality [28, 198]. It is based on the fact that if binary variables are determined, the problem will reduce to a Linear Programming (LP) problem, which has a simpler structure, and thus, it is easier to solve. Therefore, the overall idea is that, instead of considering all the variables and the constraints simultaneously, we decompose the problem into a Master Problem (MP) and a Slave Problem (SP), which are solved iteratively through a feedback loop. By doing so, the computational complexity of the solution is significantly reduced [28], even if the initial problem is non-convex. More precisely, the MP is an ILP that accounts for all binary variables along with the corresponding part of the objective function and the relevant constraints. Furthermore, it includes a set of constraints called Benders cuts, which derive from the SP. The SP is an LP that includes all the continuous variables and the associated constraints. The SP solution provides additional feasibility and infeasibility constraints that narrow down the feasible region of MP binary variables and are incorporated in the MP through the Benders cuts.

Initially, we solve the MP and obtain a lower bound Q_l for the optimal of the initial problem

Table 2.2: Summary of task deployment problem formulations.

Constraints	**Binary**	**Continuous**	[C10]	[J15]	[J16]	[C16]
Task-to-processor allocation	✓		✓	✓	✓	✓
Optional task adjustment		✓	✓	✓	✓	✓
Frequency-to-task assignment	✓			✓	✓	
Task dependencies	✓	✓			✓	✓
Task migration	✓					✓
Task non-overlapping	✓	✓	✓	✓	✓	✓
Real-time	✓	✓	✓	✓	✓	✓
Energy budget	✓	✓	✓	✓	✓	✓
Type			MILP	MINLP	MINLP	MINLP
Solution			O	O+H	O+acc.O	O+H

objective function Q^* along with a set of values for the binary variables. Then, we substitute these values into the SP and solve the dual of the SP (DSP) to obtain an upper bound Q_u for optimal value of the initial problem objective function. Based on the solution of the DSP, a new Benders cut constraint is generated as follows: if DSP has a bounded (unbounded) solution, a feasibility (infeasibility) constraint is generated. This new constraint is added into the MP, and a new iteration is performed to solve the updated MP, and then, the new SP. The iteration process stops when the gap between the upper and lower bounds is smaller than a predefined threshold. Figure 2.1 schematically describes the optimal algorithm proposed to solve the problem of task deployment for dependent task over AMP platforms with DVFS in [J16].

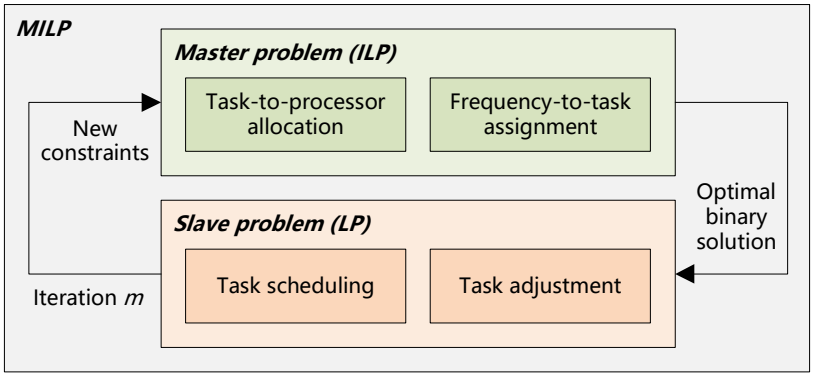


Figure 2.1: The structure of optimal decomposition-based approach for dependent tasks and AMP platforms with DVFS.

2.1.7 Accelerated optimal decomposition-based method

Although the solution provided by the previous method is optimal, this method cannot be used to efficiently solve large problem sizes, because the MP is an ILP and at each iteration, a new feasibility constraint or infeasibility constraint is added into the MP. With an increasing number of iterations, both the computational complexity and the size of MP increase.

In order to circumvent these difficulties, we propose an accelerated version to reduce the computational complexity of the optimal decomposition-based approach, which is dominated by the cost of solving the MP at each iteration. To achieve that, we relax the binary variables to be continuous variables with ranges in $[0, 1]$. The relaxed MP is an LP problem, since all the variables are continuous. Note that, if the DSP is solved with the solution of the relaxed MP, the DSP may be infeasible, since the relaxed MP solution may not be binary. To solve this problem, the solution of the relaxed MP is rounded to the nearest binary solution that is feasible to the MP. Such a binary solution can be found by heuristics, e.g., using the feasibility pump method [85]. Based on the structure of the relaxed MP and the SP, we design a distributed solution based on two-layer subgradient algorithm to solve these problems. Positive Lagrange multipliers are introduced and the dual function is obtained. Then, a two iteration process is applied, where the inner-layer iteration updates the variables under the given Lagrange multipliers and the outer-layer iteration updates Lagrange multipliers under the given variables. With the iterations between the outer-layer and the inner-layer, the Lagrange multipliers statistically converge to the optimal values when step size is a small enough value. Figure 2.2 schematically describes the accelerated version for dependent tasks and DVFS over AMP platforms with DVFS proposed in [J16].

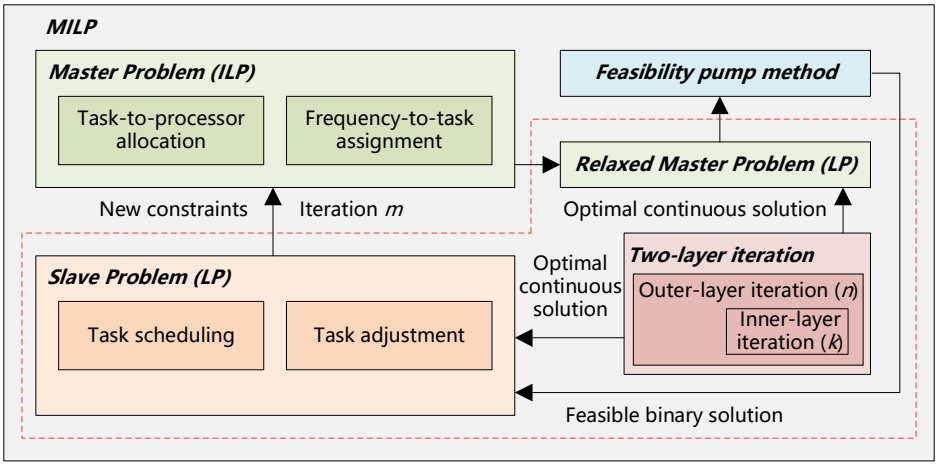


Figure 2.2: The structure of accelerated optimal decomposition-based approach for dependent tasks over AMP platforms with DVFS.

2.1.8 Heuristic methods

As binary and continuous variables are highly coupled with each other in the problems under study, it is difficult to design an efficient heuristic. When the problem formulation is changed, existing heuristics usually have to be redesigned, such as [286, 171]. Inspired by the structure of the optimal algorithm and its accelerated version, we propose two novel heuristics, where the complexity is reduced by i) removing the iteration process for SMP [J15], and ii) reducing the number of iterations for AMP [C16]. Note that, the proposed heuristics are a reduced version of the optimal approach, and thus, they can be applied to similar MILP problems without redesigning.

The heuristic in [J15] exploits the fact that the most complex steps are task-to-processor and frequency-to-task decisions. If the value of these binary variables is determined, the problem reduces to a LP problem. Therefore, the proposed heuristic initially solves the Frequency-to-task Assignment Problem (FAP) to obtain a feasible solution. Since the mandatory subtasks must be always executed, we initially consider only the frequency assignment of the mandatory subtasks and our aim is to minimize the energy consumption. Having determined the frequency-to-task decision, we obtain the execution time of mandatory subtasks. Then, we find a feasible solution for the Task Allocation Problem (TAP), with the goal of minimizing the maximum task execution time among processors. In the final step, based on frequency-to-task assignment decision and task-to-processor allocation decision, we solve the Optional Task Adjustment Problem (OTAP) under real-time and the energy budget constraints. The structure of the proposed heuristic is depicted in Figure 2.3 for independent tasks over SMPs platforms with DVFS.

In [C16], the heuristic (named HDA) has a structure similar to the optimal one, except that the optimal MP solution is replaced by a feasible MP solution (as in the accelerated version). Furthermore, the iteration stops when the SP obtains a bounded solution for first time. The structure of the proposed heuristic is depicted in Figure 2.4 for dependent tasks over AMPs.

2.1.9 Evaluation

This section presents the main results to evaluate the behavior of the proposed approaches, whereas the complete evaluation can be found in [C10, J15, J16, C16]. We present the QoS and computation

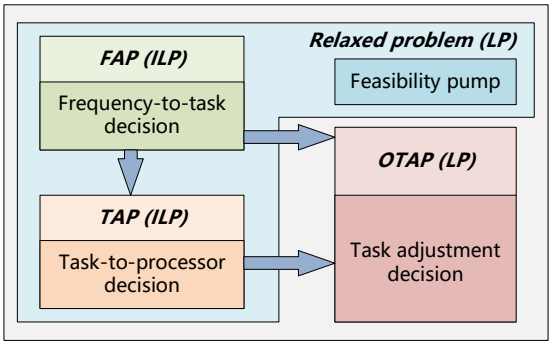


Figure 2.3: The structure of heuristic approach for independent tasks and SMP platforms with DVFS.

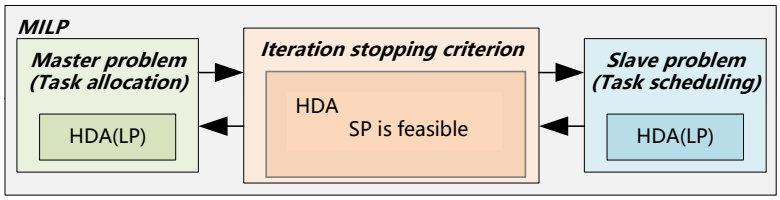


Figure 2.4: The structure of heuristic approach for dependent tasks and AMP platforms.

time of the proposed optimal approach (named OJTM) and heuristic (named HJTM) for independent tasks over SMP and the optimal approach (named JDQT) and accelerated version (named AJDQT) for dependent tasks oven AMP. Furthermore, we compare with optimal approaches, i.e. Branch and Bound method (B&B) [39, 183], which is known to provide optimal solution for the MILP problem, and stochastic approaches, i.e. Genetic Algorithm (GA) [237]. The simulations are performed on a laptop with quad-core 2.5 GHz Intel i7 processor and 16 GB RAM, and the algorithms are implemented in Matlab 2016a.

Experimental set-up

Table 2.3 summarizes the set-up. Note that using different values for the set-up will only modify the parameters, and not the problem structure. Thus, the proposed methods are still applicable.

Platform: The processor model used for the multicore platform in the experiments is based on 70 nm technology, where the accuracy of the parameters' values has been verified by SPICE simulations [55]. The processor operates at five voltage levels in the range of $[0.65\ V, 0.85\ V]$ with a step of 50 mV (i.e., $L = 5$). The power of the processor in the idle state is set to $P_0^s = 80\ \mu W$, while the corresponding frequency f_l, the dynamic power P_l^d, and the static power P_l^s under different voltage levels are shown in Table 2.3. The number of processors (i.e., M) is tuned from 4 to 10 with a step of 2.

Benchmarks: The IC task set is created by randomly generating IC task graphs with a total number of task N equal to 10 up to 50 tasks. The WCEC of the mandatory part M_i and the maximum optional part O_i of a task τ_i are assumed to be within the range $[4 \times 10^7, 6 \times 10^8]$ [286], provided from the execution of MiBench and MediaBench benchmark suites [203].

Constraints: Regarding real-time constraints, for independent tasks, the relative deadline for each task is $d_i = \min_{\forall l \in \mathcal{L}}\{\frac{M_i+O_i}{f_l}\}$ and the scheduling horizon H is assumed to be proportional to the

average processor workload, $H = \left\lceil \frac{N}{M} \right\rceil \frac{\sum_{i=1}^{N} d_i}{N}$. For dependent tasks, the relative deadline of a task τ_i is assumed to be in the range $[\underline{d}_i, \overline{d}_i]$, where $\underline{d}_i = \min_{\forall l \in \mathcal{L}}\{\frac{M_i+O_i}{f_l}\}$ and $\overline{d}_i = \max_{\forall l \in \mathcal{L}}\{\frac{M_i+O_i}{f_l}\}$ are the minimum time and the maximum time required to execute the cycles for the mandatory part and the maximum optional part, respectively. The hyper-period of the tasks is $H = \max_{\forall i \in \mathcal{N}}\{D_i\}$, where D_i is the absolute deadline of a task τ_i, $D_i = \hat{t}_s^i + d_i$, where $\hat{t}_s^i$ is the temporary start time. The temporary start time is computed by setting the temporary end time of task τ_i equal to the temporary start time of task τ_j (i.e., $\hat{t}_e^i = \hat{t}_s^i + d_i = \hat{t}_s^j$), if task τ_i precedes task τ_j. If τ_i is the first task, we set $\hat{t}_s^i = 0$. Regarding energy supply constraint, since system is in the medium energy state, the energy supply is set to $E_s = \eta E_h$, where $E_h = MHP_0^s + \sum_{i=1}^{N}[\min_{\forall l \in \mathcal{L}} \frac{M_i+O_i}{f_l}(P_l^s + P_l^d - P_0^s)]$ is the minimum energy required to execute $\{M_1 + O_1, \ldots, M_N + O_N\}$ cycles. The energy efficiency factor η is tuned from 0.8 to 0.9 with a step of 0.05.

Objective function: The objective function of the problem under study is given by a linear function of the QoS tasks, adopted from [286, 267].

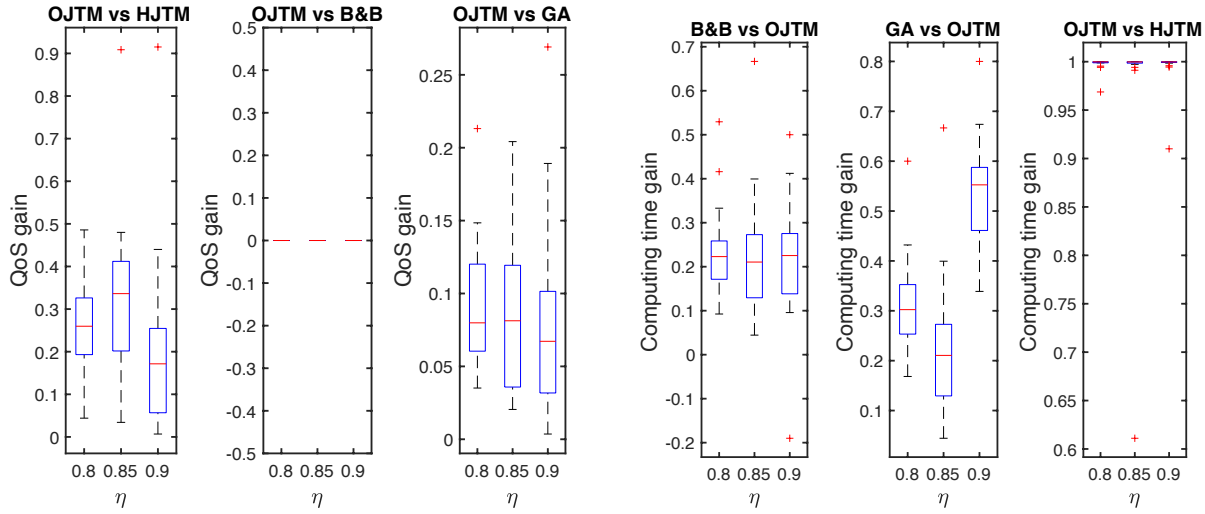
Table 2.3: Summary of experimental set-up for task deployment approaches

<table>
<tr><td colspan="6">Processor θ_k characteristics</td></tr>
<tr><td>v_l (V)</td><td>0.65</td><td>0.7</td><td>0.75</td><td>0.8</td><td>0.85</td></tr>
<tr><td>f_l (GHz)</td><td>1.01</td><td>1.26</td><td>1.53</td><td>1.81</td><td>2.10</td></tr>
<tr><td>P_l^d (mW)</td><td>184.9</td><td>266.7</td><td>370.4</td><td>498.9</td><td>655.5</td></tr>
<tr><td>P_l^s (mW)</td><td>246</td><td>290.1</td><td>340.3</td><td>397.6</td><td>462.7</td></tr>
<tr><td>P_0^s (μW)</td><td colspan="5">80</td></tr>
<tr><td>Task τ_i characteristics</td><td colspan="5">$M_i, O_i \in [4 \times 10^7, 6 \times 10^8]$</td></tr>
<tr><td rowspan="2">Real-time constraint</td><td colspan="2">Independent tasks</td><td colspan="3">Dependent tasks</td></tr>
<tr><td colspan="2">$d_i = \min_{\forall l \in \mathcal{L}}\left\{\frac{M_i+O_i}{f_l}\right\}$
$H = \left\lceil \frac{N}{M} \right\rceil \frac{\sum_{i=1}^{N} d_i}{N}$</td><td colspan="3">$d_i \in [\min_{\forall l \in \mathcal{L}}\{\frac{M_i+O_i}{f_l}\}, \max_{\forall l \in \mathcal{L}}\{\frac{M_i+O_i}{f_l}\}]$
$H = \max_{\forall i \in \mathcal{N}}\{D_i\}$</td></tr>
<tr><td>Energy supply constraint</td><td colspan="5">$E_h = MHP_0^s + \sum_{i=1}^{N}\left[\min_{\forall l \in \mathcal{L}} \frac{M_i+O_i}{f_l}\left(P_l^s + P_l^d - P_0^s\right)\right] \quad E_s = \eta E_h$</td></tr>
<tr><td>Objective function</td><td colspan="5">$\sum_{i=1}^{N} g_i(o_i) = \sum_{i=1}^{N} o_i$</td></tr>
</table>

Independent IC tasks over SMP

Figure 2.5a shows the statistical property of the QoS gain between OJTM and HJTM, B&B and GA. The QoS gain of an approach i compared to an approach j is given by $\frac{Q_i(M,N,\eta)-Q_j(M,N,\eta)}{Q_i(M,N,\eta)}$, where $Q_i(M, N, \eta)$ is the QoS achieved by approach i and $Q_j = (M, N, \eta)$ is the QoS achieved by approach j under given M, N and η parameters, respectively. We present the box plot of the QoS gains obtained under all M and N values for a given η. On each box, the central mark indicates the median, and the bottom and top edges of the box indicate the 25^{th} and 75^{th} percentiles, respectively. The whiskers extend to the most extreme data points that are not considered outliers and the outliers are plotted individually using the '+' symbol. From the obtained results, we observe that the solutions given by B&B and OJTM are same, and thus, OJTM also finds the optimal solution. Furthermore, OJTM achieves higher QoS (26.3% in average) than HJTM, and OJTM achieves higher QoS (7.6% in average) than GA. Although GA solves complex non-linear programming problem (non-convex), such as MINLP, the solution optimality is hard to guarantee.

Figure 2.5b depicts the computation time gain between OJTM and HJTM, B&B and GA. Similar as before, we denote $T_i(M, N, \eta)$ and $T_j(M, N, \eta)$ as the computation time of an approach i and j, respectively, under given M, N and η parameters. The computation time gain of an approach



(a) QoS (b) Computation time

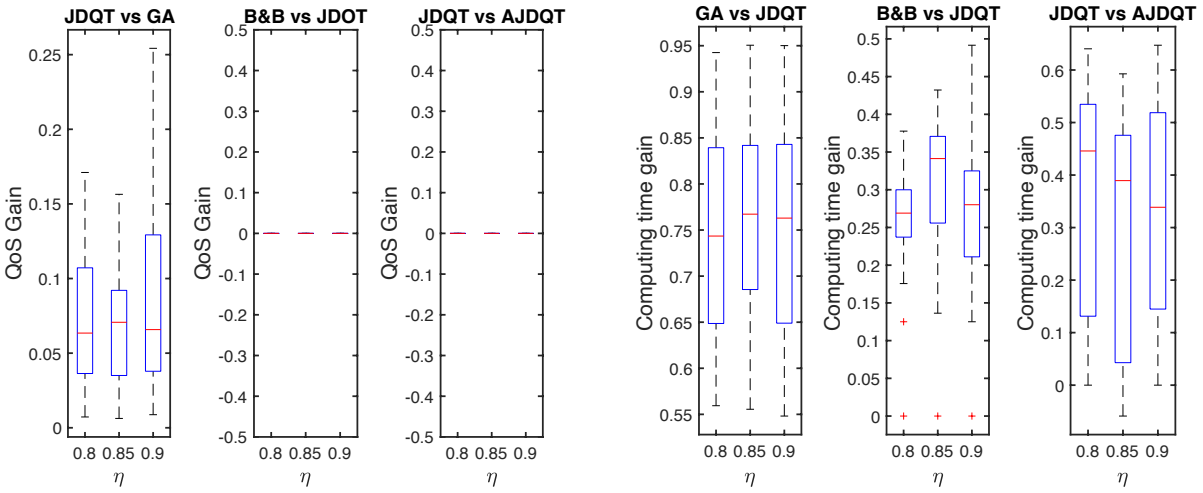
Figure 2.5: QoS and computation time gain of B&B, GA, OJTM, and HJTM with M, N and η varying.

i compared to an approach j is given by $\frac{T_i(M,N,\eta)-T_j(M,N,\eta)}{T_i(M,N,\eta)}$. Overall, OJTM takes a shorter computation time than B&B (22.6% in average) and GA (35.6% in average). The computational complexity of an optimization problem increases significantly with the number of variables and constraints. Although B&B can optimally solve MILP problems for large problem sizes, it explores a large number of nodes to find the optimal solution. Compared with OJTM, the GA structure is more complex, as in each iteration GA generates new populations, through several procedures, such as selection, reproduction, mutation and crossover. Transforming MINLP problem to a MILP simplifies the structure of the problem, and, thus, the optimal solution is easier to find. Furthermore, solving iteratively smaller problems, i.e., the MP and the SP, is more efficient than solving a single large problem. This result agrees with the comparison of [204]: the decomposition-based method is faster than the B&B for lager problem instances. HJTM can find a feasible solution within a negligible computation time compared with OJTM.

Dependent IC tasks over AMP

Figure 2.6a compares the QoS gain of JDQT with B&B and GA, computed as before. We observe that JDQT achieves higher QoS (6.8% on average) than GA and it has the same QoS with B&B. Furthermore, the accelerated version AJQDT has the same QoS with the optimal JDQT.

Figure 2.6b compares the computing time of JDQT, B&B, GA and AJDQT. JDQT takes a shorter computing time than B&B (27.8% on average) and GA (73.2% on average). AJDQT takes 32.69% on average less time than JDQT. As M and N increased, the computing time of JDQT, B&B, and GA grows, since more variables and constraints are involved in the problem, and thus, the problem size is enlarged, compared to AJDQT.



(a) QoS (b) Computation time

Figure 2.6: QoS and computation time gain of B&B, GA, JDQT, and AJDQT with M, N and η varying.

2.2 Interference-controlled run-time adaptation of *is*WCET time-triggered task execution

2.2.1 Context

In multicore architectures, several resources are shared among the cores, such as memories and interconnects. The concurrent accesses to these resources must be arbitrated, introducing non-deterministic variations to the access times. This behavior is called timing interference. The amount of interference a task may suffer during execution depends on the number of accesses to shared resources and on which tasks are running in parallel. As shown in Table 2.4, approaches bound the number of allowed memory access in order to bound the interference. For instance, offline partition the capacities of shared resources among tasks [180, 152] and cores [281]. A monitor observes at run-time the task resource usages and suspends the tasks/cores that overtake the allocated capacity. Extensions of these approaches allow dynamic changes in the resource partitioning, when resources are underutilized [179, 282]. Parallel tasks are defined by the task scheduling and allocation. During the WCET estimation of a task either: i) the worst-case interference is used, i.e. all accesses of a task to a shared resource are assumed to conflict with all other cores, or ii) the task scheduling and allocation is known and it is used to provide essential information regarding which tasks are scheduled in parallel, allowing more accurate estimation of interference. The first approach is valid for any task scheduling and allocation. However, the pessimism introduced in the WCET estimation (by being unaware of the task scheduling and allocation solution) can potentially negate the performance benefit coming from the parallel execution of the tasks on multi-cores [127] or even make the problem infeasible, if the system becomes unschedulable. Such WCETs estimations with interference can be seven times larger than the corresponding estimations without interference, both experimentally measured [C7, 153] and analytically computed [246, 245]. The second approach uses the information obtained by the task scheduling and allocation solution to compute interference-sensitive WCET (*is*WCET), which are lower than the pessimistic WCET of the first approach [214, 245, C13, 209], as they account for the interference only of the tasks scheduled in parallel. However,

the *is*WCETs are schedule-dependent, and thus, they are valid only for the schedule solution they have been estimated for.

In this context, in order to guarantee an execution within the available time, the *is*WCET schedule and allocation solution, used to compute the *is*WCET, has to be maintained during execution. Otherwise, additional interferences may occur, which have not been accounted for during task scheduling and allocation.

Table 2.4: Comparison of representative *is*WCET approaches

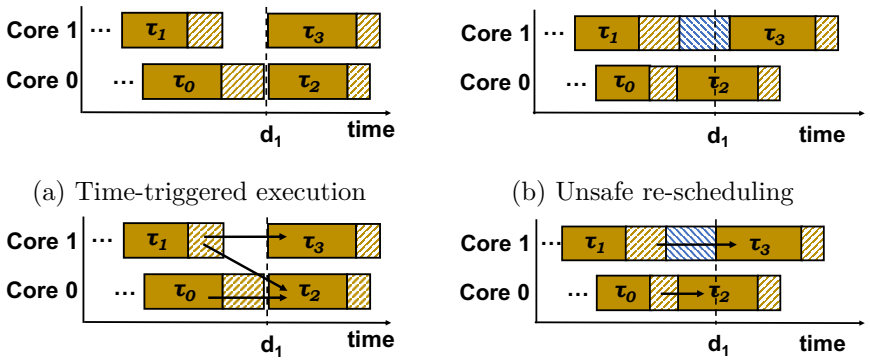
Ref.	Schedule			Interference bounds		
	TT	**Static**	**Dynamic**	**Parallel tasks**	**Static bounds**	**Dynamic bounds**
[180]	✓				✓	
[179]	✓					✓
[281, 152]		✓			✓	
[282]		✓				✓
[214, 245, 209]	✓			✓		
[244, 246]		✓		✓		
[C18]		✓		✓		
[C21]			✓	✓		

2.2.2 State-of-the-Art

In order to maintain the *is*WCET schedule during execution, the majority of existing approaches use time-triggered execution, where the tasks are executed exactly at their start time assigned in *is*WCET schedule. For instance, time-triggered approaches analyse the tasks scheduled in parallel to obtain *is*WCET [214, 245, 209]. Figure 2.7a illustrates the time-triggered execution of the task scheduling and allocation solution for four tasks $\tau_0, \tau_1, \tau_2, \tau_3$, which access a shared resource, e.g., the main memory. The delays, that each task suffers due to the interference caused by the task running in parallel, are denoted with light stripped boxes. Although time-triggered execution is time-safe, it prohibits any improvement on performance, since the tasks can start only at the time instant that has been defined by the time-triggered schedule at design-time. However, performance improvement can create slack, which can be used to increase the QoS or execute other best-effort applications. For example, in cruise control systems, the created slack can be used to further improve quality of the result produced by the control law, whereas in satellite systems less essential functions, such as scientific instrument data collection, can be activated [84].

To enable any performance improvement, adaptation is required during execution. This can be achieved by using information of the task actual execution time, which becomes available as the execution progresses. However, any adaptation occurring at run-time must be safe, i.e., either no additional interference should be allowed by the run-time adaptation or any additional allowed interference must be guaranteed to be safe. Otherwise the system execution becomes unsafe, as shown in Figure 2.7b. Task τ_0 finishes earlier than the time instance given by the *is*WCET time-triggered schedule. If run-time adaptation re-schedules τ_2 at an earlier start time, it can cause additional interferences to τ_1 (depicted by the striped blue box). These additional interferences increase the *is*WCET of τ_1, making the *is*WCET schedule invalid, and τ_1 may violate its deadline.

The work in [244, 246] performs run-time adaptation of *is*WCET time-triggered schedules, by allowing tasks to be executed earlier-than-originally scheduled, preserving the timing guarantees. This is achieved by inserting extra scheduling dependencies to the task scheduling and allocation



(a) Time-triggered execution

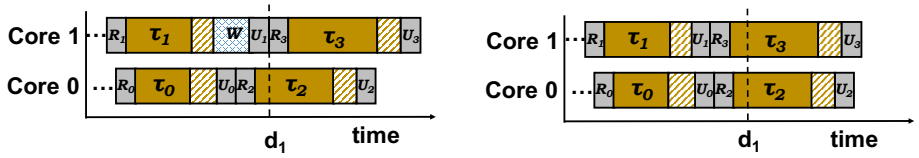
(b) Unsafe re-scheduling

(c) Enforced partial order

(d) Relaxed partial order

Figure 2.7: Task execution considering *is*WCET

solution, depicted by the arrows in Figure 2.7c, to prohibit any additional task overlapping in case of run-time re-scheduling. In this way, the partial order of task execution is enforced at run-time and no increase in the interferences can occur, maintaining the *is*WCET estimations valid. The run-time adaptation is achieved through controllers that monitor the task execution on each core and share information regarding the status of cores. Before executing task τ_i, the controller checks if the scheduling dependencies are met, and thus, the task is ready for execution. This corresponds to the ready phase $\mathcal{R}_i$ of the controller of task τ_i, depicted in Figure 2.8a. When the task finishes execution, the controller updates a status array and notifies the rest of the controllers that the task has finished its execution. In Figure 2.8a, this corresponds to the update phase $\mathcal{U}_i$ of the controller of task τ_i. However, this approach requires a global centralized synchronisation mechanism to protect the information shared among the cores. Such a centralized mechanism executes the requests from different cores in sequential order, which inserts waiting time in order to gain access in the protection mechanism, as depicted by the rectangle W in Figure 2.8a.



(a) Global

(b) Fine-grained

Figure 2.8: a) Global [244] and b) fine-grained [C18] protection mechanisms to enforce partial order.

2.2.3 Contributions

Our first contribution [C18] is to remove the limitations of global centralized mechanisms by proposing an interference-sensitive adaptation approach capable of fine-grained protection. The proposed approach enables parallel execution of the control phases on each core, whenever possible, via fine-grained protection of the shared variables, eliminating any unnecessary blocking, as shown in Figure 2.8b. In this way, tighter WCET of the run-time adaptation controller and overall better run-time execution of tasks are achieved.

Our second contribution [C21] comes from the observation that by enforcing the partial order of tasks, we limit the performance improvement that can be achieved through run-time adaptation. As depicted in Figure 2.7c, the run-time mechanism with enforced partial order can allow an earlier execution of successor tasks (τ_2 and τ_3), only when all their predecessor tasks have finished (τ_0 and τ_1). Let's assume that τ_0 started earlier-than-originally scheduled and some time slack has been created at run-time, due to early termination of a task. This time slack can be exploited to further improve performance. As depicted in Figure 2.7d, if the additional *is*WCET, due to the interferences inserted by a new task running in parallel (e.g., by running τ_2 in parallel with τ_1), is less than the time slack, then the partial order of tasks can be safely relaxed, and thus, τ_2 can be executed in parallel with τ_1. Therefore, we propose an interference-sensitive run-time adaptation approach that safely relaxes the partial order of tasks. The actual execution time of tasks across cores is explored to allow concurrent tasks to sustain more interference, than the one computed during the *is*WCET schedule, as long as the timing guarantees are preserved. Compared to existing approaches, the proposed approach is capable of exploiting the run-time variability due to a shorter task execution compared to the *is*WCET schedule computed offline. This run-time variability is created due to i) lower interference occurred during execution than the maximum possible interference, used to offline compute the *is*WCET schedule, and ii) the executed path is different than the worst-case path of the task, used to compute the *is*WCET schedule. The next sections describe the main notations of the proposed approaches, while all details, definitions and mathematical formulations can be found in [C18, C21].

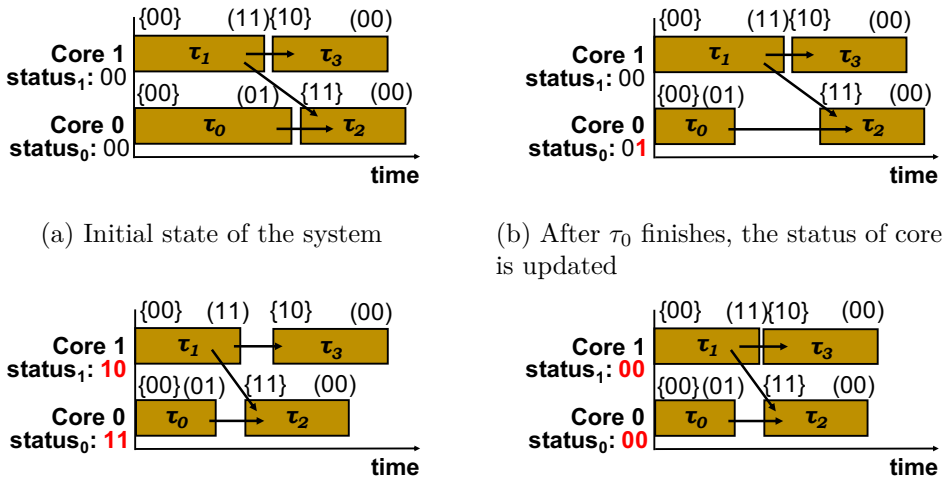
2.2.4 System model

Let T denote the set of tasks of an application to be executed on the set of cores $\mathcal{M}$ of the target platform. The input to the proposed mechanism is a time-triggered schedule that provides the start and end times of the tasks and their allocation to cores. Such time-triggered schedule can be constructed by a scheduling algorithm that provides timing guarantees, applied offline using the *is*WCET of the task-set T. The tasks of T can be dependent, or independent, and are periodically executed in a non-preemptive manner. Since the proposed approach acts upon the time-triggered schedule, any limitation stems from the task model and the scheduling algorithm used offline to derive the time-triggered schedule. A time-triggered schedule is considered safe, iff it satisfies the system-defined timing constraints, i.e., each task deadline and/or a global deadline must be met. Given a safe time-triggered schedule, a set of scheduling dependencies E_{isRA} enforce the partial order of the tasks in the time-triggered schedule, s.t. a task τ depends on the tasks $\{\tau'\}$ that finished immediately before it on all cores $\mathcal{M}$.

2.2.5 Enforcing partial order through fine-grained protection mechanism

To obtain a safe and efficient fine-grained synchronization approach, parallel execution of control phases of different cores must be allowed, whenever it is possible, without creating any concurrency issues. To achieve that, we propose a control mechanism that is executed independently on each core and for each task. The task execution is extended with two control phases, namely ready and update. To achieve fine-grained synchronisation, each core must have its own status vector (of size $|\mathcal{M}|$), where each bit of the status vector corresponds to a core. The status vector of a core represents the notifications received from other cores at any time instance.

Ready phase: During the ready phase, the controller waits for the task to become ready, i.e., all previous tasks have finished, and thus, its dependencies are met. To implement the ready phase,



(a) Initial state of the system

(b) After τ_0 finishes, the status of core 0 is updated

(c) After τ_1 finishes, core 1 notifies core 0, which re-schedules τ_2

(d) Core 1 re-schedules τ_3

Figure 2.9: Example of *is*RA-FG enforcing operation for four tasks on two cores (Curly brackets: ready vector, parentheses and arrows: notification vector).

a ready vector (of size $|\mathcal{M}|$) is required for each task τ. Each bit in the ready vector represents the core k on which the incoming edge of the scheduling dependencies originates from. The ready vectors are created offline for each task τ, based on the dependency relation E_{isRA}, and they are not modified during execution. For instance, in Figure 2.9a, the ready vector of task τ_2 is $\{11\}$, since it is has to wait for i) task τ_1 running on core 1 and ii) task τ_0 running on core 0, to finish before being executed. The ready vector of task τ_1 is $\{00\}$, since no dependency exists from another task. The controller reads the ready vector of the task τ to be executed next. Then, it has to gain access to the critical section of the status vector through the protection mechanism related to the core k. Once it has been granted access to its status vector, it checks if all task dependencies are already met. If this is true, the task τ can be executed. For instance, tasks τ_2 and τ_3 in Figure 2.9c are considered ready, since the corresponding bits of the status vectors of core 0 and core 1 are set and the status vectors are equal with the ready vectors. Before advancing to the execution phase, the controller resets the bits indicated by the ready vector of task τ in its status. This is illustrated in Figure 2.9d, where the corresponding bits in the status vectors are reset and tasks τ_2 and τ_3 are executed. Then, the protection mechanism is released and the phase finishes. Otherwise, the process is repeated, until the task becomes ready.

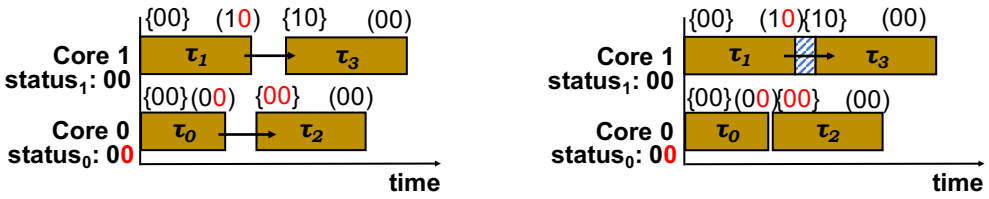
Update phase: During the update phase, the controller notifies all relevant cores that the task has finished execution. A core k' is called relevant for any task τ executed on core k, when there exists an outgoing edge from task τ towards a task τ' on core k'. To implement the update phase, a notification vector (of size $|\mathcal{M}|$) is required for each task τ that describes the relevant cores. The notification vectors are created offline for each task τ, and they are not modified during execution. For instance, in Figure 2.9a, the notification vector of task τ_1 is (11); when it finishes execution, it has to notify task τ_2 running on core 0 (bit 0) and task τ_3 running on core 1 (bit 1). After the task τ on core k completes its execution, the controller has to update the status of all the relevant cores. To do so, it initially reads the notification vector of τ. For each core i, it checks whether the core should be notified, and thus, its status should be updated. For each such core, the controller tries

to gain access to the critical section through the core's protection mechanism. If access is granted, the controller verifies if the previously occurred update of the core k has been already consumed by core i. If this is true, the k-th bit in the status of core i is set, indicating that the dependency from core k has been met. The corresponding bit in the notification vector is cleared to show that the controller has already updated the core. For instance, in Figure 2.9b the controller of core 0 updates its own bit after task τ_0 finishes. In Figure 2.9c, the controller of core 1 updates the corresponding bit in the status of core 0 after task τ_1 finishes.

2.2.6 Relaxing partial order mechanism

To enable a safe partial order relaxation during execution, the previous control mechanism is extended with the relax phase and the global time-slack computation. When a task is not ready, the control mechanism tries to relax the partial order of the tasks, when it is safe. To achieve that, a global slack is computed, which is the minimum time-slack among all cores. The time-slack of a core is given by the amount by which the execution of its tasks has been sped up. Speed-up occurs when the actual execution of a task is shorter than its *is*WCET. The partial order is allowed to be modified, if the introduced interference by any new task, running in parallel, is less than this global slack. The interference that a task τ can cause to and sustain from is upper bounded by $\iota_{max}(\tau)$.
Relax phase: To achieve partial order relaxation, the controller of core k gains access to its critical section and clears the k-th bit of the notification vector for each predecessor task τ', to indicate that the dependency has been removed, as illustrated in Figure 2.10a. In order to reflect these changes to its own status and ready vectors, it stores which dependencies have been removed in a local variable. A dependency from a predecessor task τ' on the same core k is met, i.e., the notification from core k has already occurred. Hence, the local variable is initialized with all bits set, except the k-th bit. For the same reason, the k-th bit of that task's τ' notification vector is not reset. Finally, the controller resets all the bits of its status and ready vector that were modified by the relaxation process, according to the local variable, and tests if the task is now ready. Figure 2.10b shows the partial order relaxation between task τ_1 and τ_2 for the running example (initial steps are the same as Figure 2.9a and Figure 2.9b). Notice that, data-dependencies are preserved, thus ensuring proper ordering of data-dependent tasks.
Global slack: The time-slack of a core is the difference between the actual response time $R(\tau)$ of a task τ and its end time $\epsilon(\tau)$. As the actual response time $R(\tau)$ is not known a-priori, we use a safe slack approximation, i.e., $\sigma_\tau = \beta(\tau) - t$ with $\max_{\tau' \in pred(\tau)} R(\tau') \leq t \leq \beta(\tau)$, where t is any time instance between the instance when the task becomes ready, i.e., all its predecessors have finished, and the time-triggered start time $\beta(\tau)$. This approximation is safe, since the time-slack after the execution of the task is greater or equal to the slack created before the execution of the task. This safe approximation enables an efficient computation of the global slack, i.e., the minimum slack of all cores, at any time instance t, in a distributed manner, without requiring any sort of synchronisation or explicit exchange of information among cores. This is achieved by subtracting the current time instance t from the minimum start time of all active tasks. To avoid inter-core information exchange, a global array is used to store the start time of the active task on each core and a global variable obtains the minimum value of the array. The start time of an active task is updated every time a core has to execute a new task. As soon as a new task is active, the controller of core k stores the old start time in a local variable and updates the start time of its active task with the new one. If its old start time is equal to the minimum value of the array, it means that this controller was the owner of the minimum value, and thus, it has to recalculate the new minimum of

the global array. Otherwise, it delegates this computation to the controller that is the owner of the minimum value of the array. Note that, accessing the global variable without protection can result in missing a write from another core. This means that the controller uses an older value, which is smaller than the new one. This only results in smaller global slack computation, and thus, only missed opportunities of relaxation. This is deliberately done so, in favor of run-time performance.



(a) After τ_1 finishes, core 1 notifies core 0, which re-schedules τ_2

(b) Core 1 re-schedules τ_3

Figure 2.10: Example of *is*RA-FG relaxion operation for four tasks on two cores (Curly brackets: ready vector, parentheses and arrows: notification vector).

2.2.7 Evaluation

This section presents the evaluation of the proposed approaches. We compare the performance gain and overhead of the approaches that enforce the partial order of the tasks, i.e., the proposed fine-grained approach (named *is*RA-FG) and the global centralised approach (named *is*RA-GLO) with the time-triggered execution, and the proposed fine-grained approach that relaxes the partial order of the tasks (named *is*RA-DYN) compared to *is*RA-FG. The performance gain is given by the observed execution time of the tasks allocated on a core (a.k.a. makespan). Hence, to attribute any observed makespan decrease as a gain for a run-time approach, any system parameter, that may lead to timing variability at run-time, should be controlled and explored independently, whenever possible. These parameters are mainly the interferences, the different execution paths of the benchmarks and the impact of caches. Therefore, we initially explore the timing variability that each benchmark can have, when executed on the platform. Here we show the results for three execution configurations, where two, four and eight benchmark instances are running on two, four and eight cores, respectively. Each experiment has been executed twenty consecutive iterations. We provide the average performance gain of an approach j compared to an approach i, i.e., $\frac{Makespan_i - Makespan_j}{Makespan_i}$, for all cores and experiments per configuration. The complete evaluation can be found in [C18, C21]. Note that, during the experiments, we observed no timing violations according to the time-triggered solution.

Experimental Setup

Platform & implementation: A real multi-core COTS platform, i.e., the TMS320C6678 chip (TMS in short) of Texas Instrument [253] is used. The platform characteristics are depicted in Table 2.5. All mechanisms have been implemented as a bare-metal library, with low-level functions for the controller phases using TMS hardware semaphores.

Benchmarks: To experimentally evaluate the approach, we have conducted experiments using three different applications with respect to the number of tasks, WCET, and Worst Case Resource

Table 2.5: TMS platform and benchmark characteristics.

DSP char/stics	Instr/cycle	Freq.	L1P	L1D	L2
	8	1GHz	32KB	32KB	512KB
No. DSPs	8	**NoC**	TeraNet (11 cycles)		
Shared L3	4MB SRAM	**DDR3**	512 MB	**Sem.**	32 cycles

	No. tasks	Seq. WCET (cycles)	No. WCRA
DCT	32	981,120	69,808
MERGE	47	669,026	55,415
FFT	47	275,891	41,981

Table 2.6: Benchmark timing variability

Interference variability

Caches	Path	DCT	MERGE	FFT
Disabled	**Best-Path**	32.13%	46.67%	55.03%
	Worst-Path	44.80%	43.78%	52.70%
Enabled	**Best-Path**	0.43%	0.91%	3.58%
	Worst-Path	0.32%	0.91%	3.29%

Cache variability (No interferences)

Path	DCT	MERGE	FFT
Best-Path	73.83%	69.03%	69.40%
Worst-Path	76.57%	68.60%	69.38%

Path variability (No interferences)

Caches	DCT	MERGE	FFT
Disabled	46.65%	12.84%	0.15%
Enabled	40.51%	14.69%	0.46%

Accesses (WCRA) taken from the StreamIT benchmarks [255]: i) Discrete Cosine Transformation (DCT), ii) Mergesort (MERGE), and iii) Fast Fourier Transformation (FFT).

WCET and WCRA acquisition: Since no existing static WCET analysis tool supports the TMS platform, a measurement-based approach has been used to acquire the WCET of each task. Obtaining safe and context-independent measurements requires to eliminate the sources of timing variability[78], by disabling data-caches, removing interference (i.e., the task is executed alone on one core) and providing input data to enforce the worst-case path. To perform our measurements on the real platform, we used the local timer of the core. To increase the reliability of the measurements, we have followed the approach of multiple executions. Each task has been executed 50 times, which has been shown to provide a small standard deviation [164], and maintained the largest observed value. The application has been compiled with -O0, i.e., no optimizations, in order to obtain the WCRA of each task by the produced binary. Table 2.5 depicts the overall WCET, WCRA and number of tasks of each benchmark, used to obtain the time-triggered near-optimal solutions.

Data-placement: The controller data are placed on the on-chip Multicore Shared SRAM Memory (MSM), while application data are placed in the off-chip main memory (DDR3), ensuring that the controller does not interfere with the task's execution.

TT-schedule: The input of all approaches is the offline *is*WCET schedule generated by [246].

Characterization of timing variability

We tune caches, different execution paths and interference independently in order to characterize its impact to the timing variability per benchmark. To compute the timing variability, the execution time of the best observed case and the worst observed case are compared. Table 2.6 shows the timing variability due to caches and diverse paths (computed without any interference), and the timing variability due to interferences, when all cores are running the same benchmark. We observe the impact of caches is quite high for all benchmarks, with 71.14% on average, whereas the impact of different execution paths depends on the benchmark type. The impact of the interferences is important (45.85% on average), with disabled caches. With enabled caches, the interference impact is reduced, since the cache sizes are large enough, and thus, keep the benchmark data locally.

Fine-grained compared to global centralised protection mechanisms

From the experiments, we observed that the behavior of *is*RA-FG is similar, in terms of minimum, maximum and average makespan, for all cores for all benchmarks, under any timing variability. Note that, this behavior of *is*RA-FG is due to the fixed partial task order and motivates the use of an approach that can explore the variability occurring at run-time, such as *is*RA-DYN. Therefore, this section we present the behavior of *is*RA-FG with 0% timing variability.

Figure 2.11 shows the average performance gain among all cores and experiments, for all configurations and benchmarks, for *is*RA-FG and *is*RA-GLO approaches compared to the TT schedule. Overall, the *is*RA-FG achieves performance improvements compared to the *is*RA-GLO. The gains of the *is*RA-FG compared to *is*RA-GLO are increased with increasing the number of cores, for all benchmarks. The minimum gain is observed when only 2 cores are used, i.e., for DCT benchmark where the proposed approach achieved 47.84% and *is*RA-GLO 47.07% smaller makespan than TT-schedule. As the number of cores increases, more tasks are executed in parallel. As a result, the probability of having requests for accessing the global protection mechanism by more than one core is increased, increasing the overhead of the global protection mechanism compared to the fine-grained mechanism. The maximum gains have been observed for FFT with 8 cores, where *is*RA-FG achieves 40.46% and *is*RA-GLO 5.38% smaller makespan than TT-schedule.

Relaxation compared to enforced partial order of tasks

This section presents the behavior of *is*RA-DYN with respect to the timing variability, due to interferences, caches, and multiple execution paths of the benchmarks. We have performed experiments, where we insert at each benchmarks a timing variability from 0% up to 40%, on average. Figure 2.12 depicts the average performance gains of *is*RA-DYN compared to *is*RA-FG.

The configuration with 0% timing variability is the worst set-up for *is*RA-DYN, since the timing variability of the benchmarks is eliminated as much as possible. To achieve that, the same execution path is used among executions and caches are disabled. However, it is not possible to eliminate the

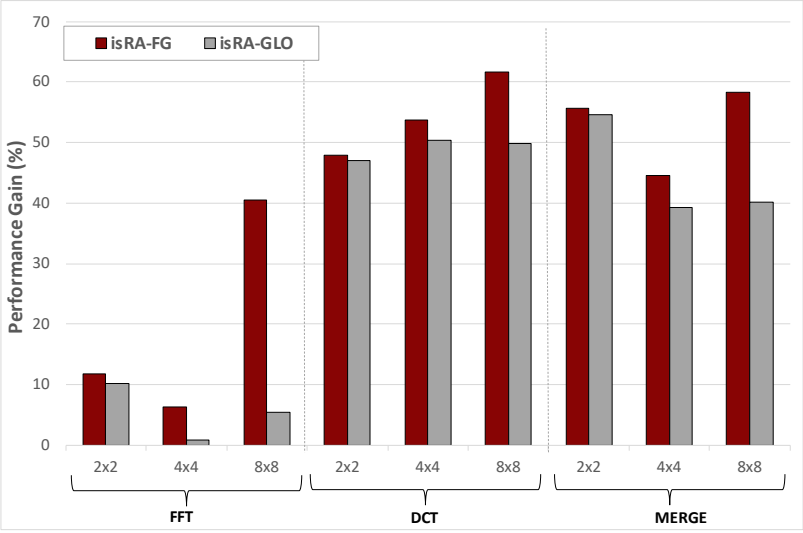


Figure 2.11: Average performance gain of *is*RA-FG and *is*RA-GLO compared to TT execution, among all cores and experiments, for all configurations and benchmarks.

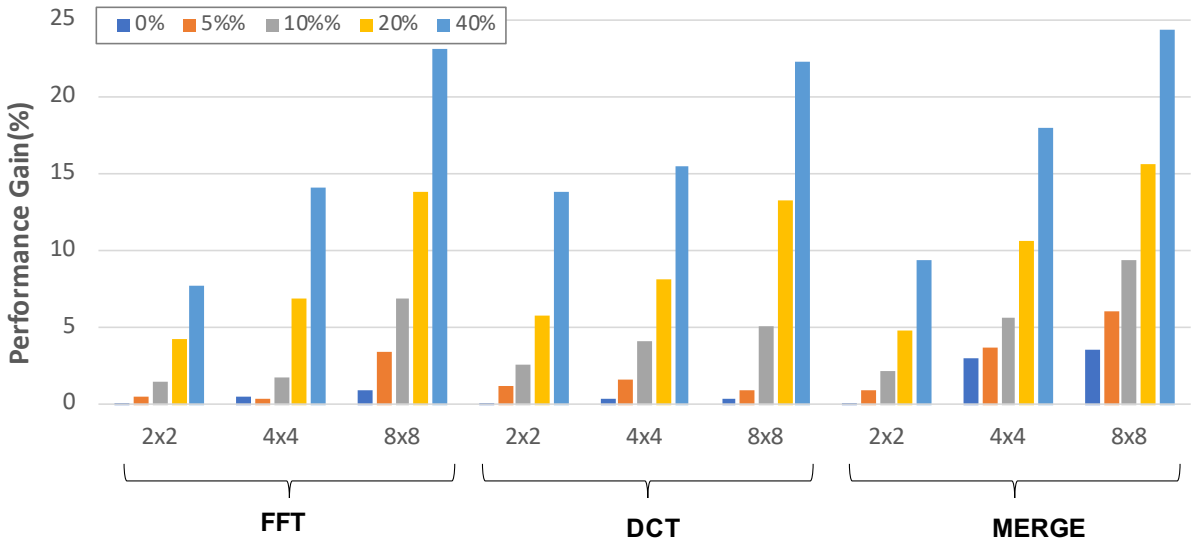


Figure 2.12: Average performance gain of *is*RA-DYN compared to *is*RA-FG, among all cores and experiments, for all configurations, benchmarks and timing variability.

interference occurring from the parallel execution of tasks. For all the experiments, we observe that the behaviour of *is*RA-DYN improves over the behavior of *is*RA-FG, in all cores, as the number of cores increases. For two cores, *is*RA-DYN provides a small gain (from 0.08% for MERGE up to 0.22% for FFT, with an average of 0.145% among all benchmarks). The low gain of two cores is due to the low interference occurring during execution in combination with a bit higher run-time overhead of *is*RA-DYN, due to the relax phase, compared to *is*RA-FG. As the number of cores is increased, the occurring interference is increased, and thus, the gain is higher. As the only source of timing variability is interference, the gain of *is*RA-DYN verifies that the proposed approach is capable of exploring the occurring interference during execution, compared to *is*RA-FG.

To quantitatively characterize the behavior of the *is*RA-DYN, when other sources of timing variability occur on top of the interferences, we insert an average variability of 5%, 10%, 20% and 40% in the WCET of the benchmarks (WCRA remains unchanged). Overall, *is*RA-FG fails to take advantage of timing variability during execution, due to its fixed partial order policy. On the contrary, *is*RA-DYN provides higher gains as the variability is increased. For two cores, as the timing variability is increased, the gains also increase. Considering all benchmarks, we observe an average gains of 0.85%, 2.09%, 4.95%, and 9.33%, for 5%, 10%, 20% and 40% variability, respectively. The maximum gain for 40% variability is 11.35% observed for core 0 running DCT. As the number of cores is increased, the gains are also increased. For four cores, the average gain over all benchmarks is 1.89%, 3.82%, 8.46% and 15.86% for the different variabilities. The maximum gain for 40% variability is 19.85% observed for core 1 running MERGE. For eight cores, the gains are even higher, i.e., with an average gain over all benchmarks equal to 3.45%, 7.11%, 14.22% and 23.26% for the different variabilities. The maximum gain for 40% variability is 25.31% observed for C4 running MERGE.

Controller cost

Table 2.7 depicts the corresponding WCET in cycles for the *is*RA-GLO, *is*RA-FG and *is*RA-DYN approaches when executed over TMS. The time-triggered mechanism cost is 270 cycles, where 70 cycles are required to write a watchdog timer with the start time of the task and at least 200 cycles for serving an interrupt handling routine, when the timer expires. As the number of cores increases, the WCET cost for the controller of *is*RA-FG and *is*RA-DYN increases almost linearly compared to the WCET of *is*RA-GLO. Due to the additional relax phase, the overhead of the *is*RA-DYN controller is a bit higher than *is*RA-FG controller. Despite the increased overhead, *is*RA-DYN can provide further performance improvements, as it has been shown in the previous section.

Table 2.7: WCET controller overhead of *is*RA-GLO, *is*RA-FG and *is*RA-DYN approaches (cycles).

Approach	**WCET per phase**			**Total WCET**		
	Ready	**Notify**	**Relax**	$\|\mathcal{M}\|$=2	$\|\mathcal{M}\|$=4	$\|\mathcal{M}\|$=8
***is*RA-GLO**	$251^*\|\mathcal{M}\|^2$	$344^*\|\mathcal{M}\|^2$	-	2,380	9,520	38,080
***is*RA-FG**	$251^*\|\mathcal{M}\| + 436$	$213^*\|\mathcal{M}\|^2 + 185^*\|\mathcal{M}\| + 169$	-	2,323	5,745	17,701
***is*RA-DYN**	$251^*\|\mathcal{M}\| + 436$	$213^*\|\mathcal{M}\|^2 + 185^*\|\mathcal{M}\| + 169$	$183 + 201^*\|\mathcal{M}\|$	2,908	6,732	19,492

2.3 Risk-permissive run-time adaptation of task execution in mixed-critical systems

2.3.1 Context

Mixed-critical systems consist of applications with different properties and requirements, and thus, different criticality levels [263]. The criticality level depends partially on the consequences on the system when an application fails to meet its timing constraints. For instance, in avionics the Design Assurance Level (DAL) model [220] defines hard real-time applications with high criticality levels A, B or C and soft real-time applications with low criticality levels D or E. The applications with high criticality level usually have hard real-time constraints and require guarantees regarding their execution in time. To ensure these timing guarantees, safe WCET estimations must be used. However, WCET estimations are pessimistic, leading to over-allocation of the resources to high criticality applications, and in the worst case, to a system that is considered unschedulable.

In this context, in order to increase the overall system quality, e.g., by a longer execution of low criticality tasks, or even to obtain schedulable systems, the pessimism introduced during WCET estimations should be reduced.

2.3.2 State-of-the-Art

As discussed in Chapter 1, three main categories exist to reduce WCET pessimism: i) interference-free, ii) interference-controlled, and iii) risk-permissive approaches. Interference-free approaches apply isolation and resources usually cannot be claimed by low criticality tasks. Interference-controlled approaches bound the allowed interferences by analysing the memory accesses, which usually require a detailed analysis. Risk-permissive approaches take advantage of the different criticalities of tasks and follow a more optimistic approach. They allow design decisions, that may lead to timing violations, watch at run-time for risks, that can lead to timing violations, and take actions in order to mitigate them, if needed.

Table 2.8: Comparison with representative risk-permissive approaches

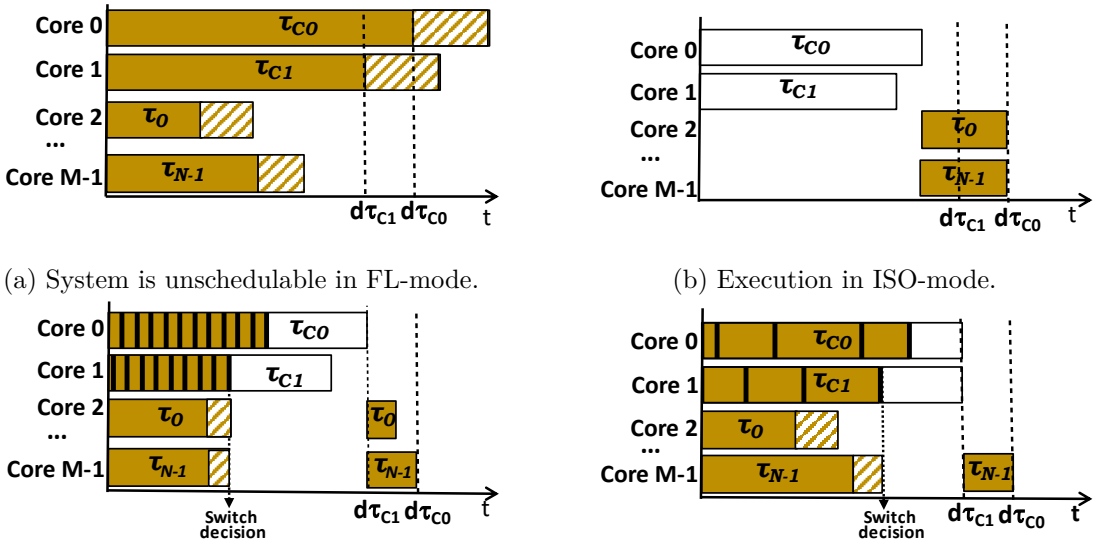
Ref.	Timing violation		Explore slack		In HI-mode			
	Interference	WCET underest.	Sta.	Dyn.	Drop low	Reduce priority	Timing budget	Extend Periods
[41, 22, 23, 142]		✓			✓			
[12, 162, 86]		✓			✓			
[229, 68, 75]		✓	✓		✓			
[111, 26]		✓	✓	✓	✓			
[185]		✓	✓	✓			✓	
[40]		✓		✓		✓	✓	✓
[21]		✓					✓	
[C8, C7, C9, W1, J14]	✓			✓	✓			

The majority of risk-permissive approaches focuses on timing violations due to underestimation of the WCET. Different confidence levels are used for the WCET estimation. The higher the criticality level, the larger and safer the WCET estimations are [41]. For instance, in dual criticality systems, a pessimistic, with high assurance, upper bound (C^H), and a less pessimistic, with lower assurance bound (C^L) is used for the WCET estimation of high criticality tasks, while only the less pessimistic WCET estimation (C^L) is used for low criticality tasks. Such a mixed-critical system has two executions modes: low criticality mode (LO-mode) and high criticality mode (HI-mode). The system starts execution in LO-mode, where both high criticality and low criticality tasks are executed. Usually, if a low criticality task exceeds its C^L, it is dropped. However, as soon as a high criticality task exceeds its C^L, the system switches from LO-mode to HI-mode to meet the timing constraints of the high criticality tasks. In HI-mode, all low criticality tasks are usually dropped, e.g.,[22, 23, 41], degrading the overall system QoS. To improve QoS, existing approaches work on two directions: i) explore other strategies, than dropping low criticality tasks in HI-mode, and ii) explore static or dynamic ways to either postpone the mode-switch or to switch back to LO-mode. Other strategies in HI-mode consist of i) setting the priority of low criticality tasks below the priority of any high criticality task, ii) reducing the execution time requirements of low criticality tasks in high criticality mode, and iii) extending the periods of low criticality tasks [40]. For instance, a high criticality WCET, lower than the low criticality WCET, is used for the low criticality tasks in order to guarantee that they progress during HI-mode [21]. Static approaches determine the largest value, to be added to the C^L of high criticality tasks, while system remains schedulable. This value extends the mode switch further than C^L. Such methods are inspired by sensitivity analysis [229] and zero-slack [68, 75]. Static approaches are applied before execution, thus exploring only the existing slack due to system under-utilisation. On the contrary, dynamic approaches exploit the slack created during execution. When the actual execution time of a task is lower than its C^L, slack is created, since the task finished earlier than expected in LO-mode. This slack can be used by the next high criticality tasks and potentially postpone the mode switch, e.g., through single budget [111], bailout protocol [26] and feedback control mechanisms [185], or at the end, after all high criticality tasks finished execution, in order to switch back to LO-mode [12, 162, 86].

However, existing approaches are based on several confidence levels for the WCET estimations, which are static estimations and, in the best case, allow to observe and use the slack, only after a task has terminated.

2.3.3 Contributions

We proposed an approach that computes dynamically new estimations of the WCET during execution, based on the task progress, to derive the available time-slack and postpone mode switch based on a safety condition. Assume that due to the WCET pessimism, the system is not schedulable when both high and low criticality tasks are executed (Full Load - FL - mode), as depicted in Figure 2.13a. Then, the safe solution is to execute first only the high criticality tasks (Isolation - ISO - mode), and only when they finish execution, the low criticality tasks are allowed to be executed, as depicted in Figure 2.13b. The proposed approach mitigates the WCET pessimism by regularly computing during execution a new value of the WCET, i.e., the remaining WCET, RC^{ISO}, which is the part of the C^{ISO} that remains for the rest of the execution from that point till the end, using information regarding the actual progress of the high criticality task and the occurred interferences. As shown in Figure 2.13c, the proposed controller, invoked at the time instances illustrated by the black lines, uses the updated RC^{ISO} in order to verify whether there is a risk for a deadline miss for the high criticality task. If not, the tasks continue their concurrent execution. Otherwise, the low criticality tasks must stop interfering with the high criticality tasks, and are suspended (switch decision taken by τ_{C1} in Figure 2.13c). When the high criticality tasks finish, low criticality tasks can resume their execution. A grammar has been proposed for modelling the high criticality tasks and for proving the safety of the proposed approach [C8]. We proposed a static version that invokes the controller at statically predefined points, illustrated by the black lines in Figure 2.13c [C8, C7, C9, W1]. The proposed approach has been leveraged in order to dynamically decide when to invoke the controller, reducing the overhead introduced due to the execution of the controller, further increasing the gains [J14]. The two approaches have been implemented on a real platform (8-core Texas Instruments TMS320C6678). The remaining sections describe the main approach, whereas all details and mathematical formulations can be found in [C8, C7, C9, W1, J14].



(a) System is unschedulable in FL-mode.

(b) Execution in ISO-mode.

(c) Execution under RWCET with statically defined points [C8, C7, C9, W1]

(d) Execution under RWCET with dynamically defined points [J14]

Figure 2.13: Motivational example.

2.3.4 System model

The platform target domain is a multi/many processor with M cores and R shared resources, whereas the application domain is a mixed-critical system consisting of a task set of size N, consisting of high criticality tasks, denoted as τ_{C_i}, and low criticality tasks, denoted as τ_i. The system has two modes of execution: i) Full Load (FL-mode) mode, where both high criticality tasks and low criticality tasks are executed on the processor, and ii) Isolation (ISO-mode) mode, where only the high criticality tasks are executed on the processor. Note that, the system modes are similar to LO-mode and HI-mode, but another notation is used to avoid confusion with the approaches that use several confidence levels for the WCET. Two WCET estimations are obtained, one WCET estimation for FL-mode considering the maximum number of interferences due to low and high criticality tasks, denoted as $C^{FL}_{\tau_{C_i}}$, and one WCET estimation for ISO-mode considering interferences due to only high criticality tasks, denoted as $C^{ISO}_{\tau_{C_i}}$. The run-time controller switches between these two modes of execution to always guarantee in-time execution of the high criticality tasks, and maximize the execution of low criticality tasks, whenever possible. A static partitioned scheduling has been applied where high critical tasks and low criticality tasks are executed on different cores. If several tasks are mapped on the same core, they are executed non-preemptively.

2.3.5 Design time analysis for high criticality tasks

Task model and instrumentation: A grammar is designed to model high criticality tasks [C8]. A high criticality task is described by a set of Control Flow Graph (CFGs), constructed by the binary code, obtained after compiling the high criticality source code. Each CFG corresponds to a function F of the high criticality task. Therefore, the high criticality task τ_{C_i} is a set of functions $\mathcal{S} = \{F_0, F_1, ..., F_l\}$, with F_0 the main function. The CFG of a function F is a directed graph $G = (V, E)$, consisting of a finite set of nodes V composed of 5 disjoint sub-sets $V = \mathcal{N} \cup \mathcal{C} \cup \mathcal{F} \cup \{IN\} \cup \{OUT\}$ and a finite set of edges $E \subseteq V \times V$ representing the control flow between nodes. $N \in \mathcal{N}$ represents a block of one or more binary instructions, $C \in \mathcal{C}$ represents the block of binary instructions of a condition statement, $F \in \mathcal{F}$ represents the binary instructions of the function caller of a function F and links the node of the current function with the CFG of the function F, IN and OUT are the input and output nodes. In the proposed grammar, a function F has exactly one input node, one output node, and a non-terminal node B as depicted in Figure 2.14. The non-terminal node B is derived as an empty node (Figure 2.14b), a single node N (Figure 2.14c), a sequential component (Figure 2.14d), i.e., the concatenation of non-terminal nodes, an if-then-else component (Figure 2.14e), i.e., the concatenation of a C conditional node with two mutually executed paths that end to the same non-terminal node, a loop component (Figure 2.14f), i.e., the concatenation of a loop condition C with two mutually executed paths, one with a non-terminal node that exits the loop and one with the non-terminal node for repetition of the loop kernel, or function call node F (Figure 2.14g).

Figure 2.15 illustrates a simple example on obtaining the CFG of a high criticality task. The C code (Figure 2.15a) is compiled and the CFG is constructed by the assembly code. L.1 to L.9 (Figure 2.15b) handle the stack and initialise the local variables and correspond to N_1 (Figure 2.15c), L.10 to L.14 describe the exit condition of the loop and correspond to C, L.15 to L.26 describe the loop kernel and the increase of i (N_2) and L.27 to L.32 manage the stack and performs the return from function (N_3).

Instrumentation points $p_{\tau_{C_i}}$ are inserted in the source code of the high criticality task τ_{C_i}, in order to invoke the controller that will compute the remaining WCET at run-time. Instrumentation

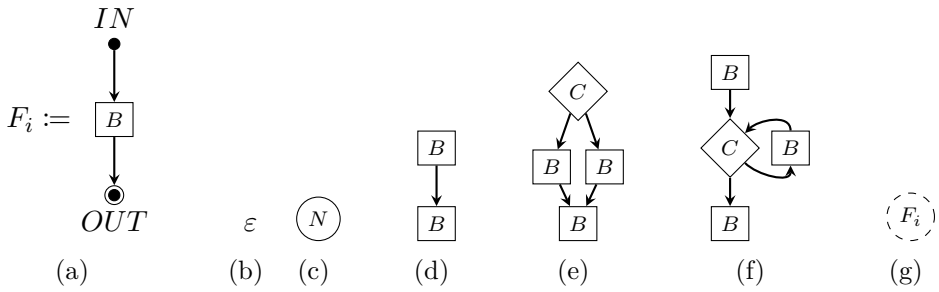


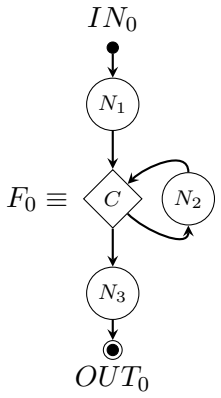
Figure 2.14: Schematic representation of grammar rules

```c
int main(void){
  int i;
  int A[10];
  for (i=0;i<10;i++){
    A[i]=i;
  }
  return 0;
}
```

(a) C code.

```
main:
        addi    sp, sp, -64
        sw      ra, 60(sp)
        sw      s0, 56(sp)
        addi    s0, sp, 64
        li      a0, 0
        sw      a0, -12(s0)
        sw      a0, -16(s0)
        j       .LBB0_1
.LBB0_1:
        lw      a1, -16(s0)
        li      a0, 9
        blt     a0, a1, .LBB0_4
        j       .LBB0_2
.LBB0_2:
        lw      a0, -16(s0)
        slli    a2, a0, 2
        addi    a1, s0, -56
        add     a1, a1, a2
        sw      a0, 0(a1)
        j       .LBB0_3
.LBB0_3:
        lw      a0, -16(s0)
        addi    a0, a0, 1
        sw      a0, -16(s0)
        j       .LBB0_1
.LBB0_4:
        li      a0, 0
        lw      ra, 60(sp)
        lw      s0, 56(sp)
        addi    sp, sp, 64
        ret
```

(b) Assembly code (RISC-V32).



(c) CFG

Figure 2.15: Illustration example where CFG is obtained from C code.

points can be inserted before the execution of the first binary instruction of each node of CFG. Representing instrumentation points by a lower-case symbol, 5 disjoint sub-sets of instrumentation points can exist, based on the node type: $\{n\}, \{c\}, \{f_i\}, in, out$. By re-compiling the instrumented source code of the high criticality task, we reconstruct the set of Extended Control Flow Graphs (ECFGs), i.e., the CFGs where each node is extended with the instructions of the inserted instrumentation point. Note that, the point *start* refers to the point before execution, i.e., point *in* of function F_0.

Structure information: After the ECFG construction, we use an ECFG parser in order to extract information regarding the ECFG structure that will allow to distinguish different visits of the same instrumentation point during execution (e.g., in loops, function calls). The structure information of a point $p_{\tau_{C_i}}$ is:

- The nested level of $p_{\tau_{C_i}}$, $level[p_{\tau_{C_i}}]$, which indicates the nested loop depth of the point and it is i) set to 0, if $p_{\tau_{C_i}}$ is the *start* point, ii) set to 1, if $p_{\tau_{C_i}}$ is a sequential point between the *IN* and

OUT of an ECFG, or iii) increased by 1, for each loop where $p_{\tau_{C_i}}$ resides in.

- The ancestor point of $p_{\tau_{C_i}}$, $head[p_{\tau_{C_i}}]$, which indicates the point where a loop entry or a function call occurred before reaching the point $p_{\tau_{C_i}}$. The $head[p_{\tau_{C_i}}]$ of a point $p_{\tau_{C_i}}$ is: i) the *start* point, if $p_{\tau_{C_i}}$ is a point with *level* 1 in the main function F_0, ii) the function caller, if $p_{\tau_{C_i}}$ is a point with *level* 1 in the called function, or iii) the condition of the loop, if $p_{\tau_{C_i}}$ is inside a loop.

- The function call behavior, $type[p_{\tau_{C_i}}]$, which is: i) *F_ENTRY*, if $p_{\tau_{C_i}}$ is a function entry (function caller), ii) *F_EXIT*, if $p_{\tau_{C_i}}$ is a function exit, i.e., the node where a function returns to, iii) *F_ENEX*, if $p_{\tau_{C_i}}$ is both a function entry and a function exit, i.e., the point $p_{\tau_{C_i}}$ where the function returns is also a function caller, or iv) $-$, if $p_{\tau_{C_i}}$ is not related to function calls.

For Figure 2.15, assuming n_1, c, n_2 and b_3 are the points inserted in the blocks of Figure 2.15c, we obtain: $level[n_1]$=1, $level[c]$=1, $level[n_3]$=1 and $level[n_2]$=2, $head[n_1]$=*start*, $head[c]$=*start*, $head[n_3]$=*start*, and $head[n_2]$=c, $type[n_1]$=$-$, $type[c]$=$-$, $type[n_2]$=$-$, and $type[n_3]$=$-$.
Timing information: The timing information between instrumentation points is provided by partial WCETs. The partial WCET two instrumentation points x_{τ_i} and point p_{τ_i} is given by $C^{ISO}_{\tau_{C_i}}[x_{\tau_{C_i}}\text{-}p_{\tau_{C_i}}] = C^{ISO}_{\tau_{C_i}}[x_{\tau_{C_i}}] - C^{ISO}_{\tau_{C_i}}[p_{\tau_{C_i}}]$, where $C^{ISO}_{\tau_{C_i}}[x_{\tau_{C_i}}]$ (reps. $C^{ISO}_{\tau_{C_i}}[p_{\tau_{C_i}}]$) denotes the WCET from point $x_{\tau_{C_i}}$ (resp. $p_{\tau_{C_i}}$) until the end of execution. Two types of partial WCET are computed:

1. For all instrumentation points, we compute $C^{ISO}_{\tau_i}[head[p_{\tau_{C_i}}]\text{-}p_{\tau_{C_i}}]$, i.e., the $C^{ISO}_{\tau_{C_i}}[x_{\tau_{C_i}}\text{-}p_{\tau_{C_i}}]$ between the head of point $p_{\tau_{C_i}}$ and the point $p_{\tau_{C_i}}$.

2. For points placed in the entry of loops, we compute the $C^{ISO}_{\tau_{C_i}}$ between any two consecutive iterations ($j-1$ and j) of the loop, i.e., $C^{ISO}_{\tau_i}[p^{j-1}_{\tau_{C_i}}\text{-}p^{j}_{\tau_{C_i}}]$. If multiple paths exist between these points (e.g., branches of if-then-else components, function calls from different entry points), the minimum difference is maintained. Note that, the minimum value is required in order to be safe, since this value will be subtracted from the overall WCET, during RWCET computation at run-time, i.e., $C^{ISO}_{\tau_{C_i}}[p^{j-1}_{\tau_{C_i}}\text{-}p^{j}_{\tau_{C_i}}] = min(C^{ISO}_{\tau_{C_i}}[p^{j-1}_{\tau_{C_i}}] - C^{ISO}_{\tau_{C_i}}[p^{j}_{\tau_{C_i}}])\ \forall\ j$.

Last, but not least, we compute the overhead, C^{FL}_{ptp}, required to reach the next instrumentation point, in the worst case, considering task execution in both forward and backward way:

$$C^{FL}_{max,F} = max(C^{FL}_{\tau_{C_i}}[head[p_{\tau_{C_i}}]\text{-}p_{\tau_{C_i}}])\ \forall\ i\ and\ \forall\ p_{\tau_{C_i}} \tag{2.1}$$

$$C^{FL}_{max,B} = max(C^{FL}_{\tau_{C_i}}[p^{j-1}_{\tau_{C_i}}\text{-}p^{j}_{\tau_{C_i}}])\ \forall\ i, \forall\ p_{\tau_{C_i}} \in \{c\}\ and\ \forall\ j \tag{2.2}$$

$$C^{FL}_{max} = max(C^{FL}_{max,F}, C_{max,B}) \tag{2.3}$$

To illustrate the required timing information using the example of Figure 2.15, the following partial WCET are computed: $C^{ISO}_{\tau_{C_i}}[start\text{-}c]$, $C^{ISO}_{\tau_{C_i}}[start\text{-}n_3]$, $C^{ISO}_{\tau_{C_i}}[c\text{-}n_2]$, $C^{ISO}_{\tau_{C_i}}[c^{j-1}\text{-}c^{j}]$ *with* $j = 0 \ldots 9$, and the maximum of these values.

2.3.6 Run-time control mechanism

Overview: At run-time, each high criticality task executes its own run-time control mechanism, which monitors the ongoing execution time, dynamically computes the remaining WCET of the task in isolated execution and checks its safety condition to locally decide if the low criticality tasks should be suspended. The high criticality tasks are not responsible for the suspension of the low

criticality tasks. They send a request to a master, which has a global view and is in charge of collecting the high criticality tasks requests, suspending and restarting the low criticality tasks. The master suspends the low criticality tasks when at least one critical task sends a request for isolated execution, because its safety condition is not satisfied. The master updates the number of active requests and it restarts the low criticality tasks when all requests have been executed. Figure 2.16 illustrates the behavior through an example with two critical tasks running in parallel. The safety condition of τ_{C_0} is violated and thus it sends a request for isolated execution to the master, depicted by the arrow iso. The master upon receiving this request sets the number of active requests to 1 and suspends the low criticality tasks (arrow stop). Then, the requester (τ_{C_0}) informs the master that its execution is finished (arrow end). As the critical task τ_{C_1} has not yet requested isolated execution, no risk exists for its deadline. The master resumes the low criticality tasks (arrow restart). Later, the critical task τ_{C_0} requests isolated execution after the request of τ_{C_1}. The master will restart the low criticality tasks only when both critical tasks have finished. Note that, the master is not assigned on the same core where a high criticality task is executed, as this option will increase the WCET of the critical task due to the received requests.

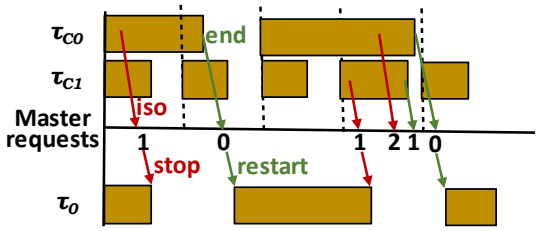


Figure 2.16: Run-time behavior among the master, high and low criticality tasks.

Statically defined points

RWCET computation: Due to the instrumentation points inserted to the source code of the high criticality task τ_i, the controller is invoked during execution and re-computes in a safe way the remaining WCET (RWCET), noted as $RC^{ISO}_{\tau_{C_i}}$ at each point $p_{\tau_{C_i}}$, based on the task progress. Algorithm 1 summarises the computation of $RC^{ISO}_{\tau_{C_i}}$ at a point $p_{\tau_{C_i}}$. The algorithm takes as input the instrumentation point $p_{\tau_{C_i}}$ along with its Structure and Timing Information ($STI_{\tau_{C_i}}$), which includes the *type*, *level*, *head* and partial WCETs of the point, pre-computed during the design-time analysis of the high criticality task. To be able to compute the RC, without unrolling the code of the high criticality task, the computation is performed per level, with the help of the array $RL_{\tau_{C_i}}$. A local level $ll_{\tau_{C_i}}$ is used to depict the current nested level of point $p_{\tau_{C_i}}$, taking into account function calls and loops. The local level is computed by adding the $offset_{\tau_{C_i}}$ and the *level* of the point $p_{\tau_{C_i}}$ (L. 5). Note that, the $level[p_{\tau_{C_i}}]$ depicts the level of nested loops inside the ECFG of a function, by definition. The $offset_{\tau_{C_i}}$ provides the level that must be added, because of any occurred function call. Therefore, when a function entry point is observed (*C5* is true, L. 14), i.e., a function call occurs, we increase the offset with the level of the entry point (L. 15). When an exit point is observed (*C1* is true, L. 2), i.e., a function returns, we decrease the offset by the level of the entry point (L. 3). Then, the observation level $o\_level_{\tau_{C_i}}$ is used to decide if we are traversing ECFG in a forward (*C2* or *C4* is true) or backward direction (*C3* is true). When the ECFG is traversed in a forward direction, the remaining WCET in local level $ll_{\tau_{C_i}}$, $RL_{\tau_{C_i}}[ll_{\tau_{C_i}}]$, is

computed by subtracting the partial WCET of the point $_{\tau_{C_i}}$ from the remaining WCET computed on the previous local level (L. 7 and L. 11). By definition, the point in the previous local level is the head point of $p_{\tau_{C_i}}$. When the ECFG is traversed backwards, we are in a loop. Thus, we have reached the point that corresponds the condition statement of the loop and we subtract the partial WCET computed between any two iterations, $j-1$ and j (L. 9). In this way, the remaining WCET of the head point at local level $ll_{\tau_{C_i}} - 1$ is updated accordingly, before entering the loop, where points have a local level equal to $ll_{\tau_{C_i}}$. Note that, before execution, the initialisation is as follows: $RL_{\tau_{C_i}}[0] = C^{ISO}_{\tau_{C_i}}$ (the overall WCET of τ_{C_i}), the remaining elements of the array $RL_{\tau_{C_i}}$ to zero, $offset_{\tau_{C_i}} = 0$, $o\_level_{\tau_{C_i}} = 0$, and $last\_point_{\tau_{C_i}}[0] = start$.

Function *Compute_static_RWCET*($p_{\tau_{C_i}}$, $STI_{\tau_{C_i}}$)

```
    if (type[p_τCi] ==F_EXIT||F_ENEX) then                                  /* C1 */
1   |   offset_τCi = offset_τCi − level[p_τCi];
2   |   o_level_τCi = o_level_τCi − 1;
3   end
4   ll_τCi = offset_τCi + level[p_τCi];
5   if (o_level_τCi < ll_τCi ) then                                         /* C2 */
6   |   RL_τCi[ll_τCi] = RL_τCi[ll_τCi] − C^ISO_τCi[head[p_τCi]-p_τCi];
7   end
8   else if (last_point_τCi[ll_τCi] == p_τCi ) then                         /* C3 */
9   |   RL_τCi[ll_τCi] = RL_τCi[ll_τCi − 1] − C^ISO_τCi[p^{j-1}_τCi, p^j_τCi];
10  end
11  else                                                                    /* C4 */
12  |   RL_τCi[ll_τCi] = RL_τCi[ll_τCi] − C^ISO_τCi[head[p_τCi]-p_τCi];
13  end
14  last_point_τCi[ll_τCi] = p_τCi ;
15  o_level_τCi = ll_τCi ;
16  if (type[p_τCi] ==F_ENTRY||F_ENEX) then                                 /* C5 */
17  |   offset_τCi = offset_τCi + level[p_τCi]
18  end
19  return RL_τCi[ll_τCi];
```

Algorithm 1: Static control mechanism at point $p_{\tau_{C_i}}$.

For instance, let's illustrate how the RWCET is computed for the example of Figure 2.15. At the first invocation of the controller at point c, $ll_{\tau_{C_i}} = 1$. Since $o\_level_{\tau_{C_i}} = 0$, the graph is traversed in forward direction and the RWCET is given by $RL_{\tau_{C_i}}[1] = RL_{\tau_i}[0] - C^{ISO}_{\tau_{C_i}}[start\text{-}c]$. The rest of the variables are updated, i.e., $last\_point_{\tau_{C_i}}[1] = c$ and $o\_level_{\tau_{C_i}} = 1$. For the first invocation at point n_2, $ll_{\tau_{C_i}} = 2$. The graph is still traversed in forward direction and the RWCET is given by $RL_{\tau_{C_i}}[2] = RL_{\tau_{C_i}}[1] - C^{ISO}_{\tau_{C_i}}[c\text{-}n_2]$, $last\_point_{\tau_{C_i}}[2] = n_2$ and $o\_level_{\tau_{C_i}} = 2$. When c is invoked in the second iteration, $ll_{\tau_{C_i}} = 1$. Since $o\_level_{\tau_{C_i}} < ll_{\tau_{C_i}}$ and the last point in this level was c, the graph is now traversed in backward direction. The RWCET is updated by $RL_{\tau_{C_i}}[1] = RL_{\tau_{C_i}}[1] - C^{ISO}_{\tau_{C_i}}[c^{j-1}\text{-}c^j]$, $last\_point_{\tau_{C_i}}[1] = c$ and $o\_level_{\tau_{C_i}} = 1$. With this update, the RWCET will be correctly computed for the points inside the loop. The RWCET in the remaining points is computed similarly.

Safety condition: Per core that runs a critical task, at each point, the control checks whether the

low criticality tasks should be suspended through the safety condition $RC^{ISO}_{\tau_{C_i}} + C^{FL}_{max} + t_{cntr} \leq d_{\tau_{C_i}} - t$, where $RC^{ISO}_{\tau_{C_i}}$ is the remaining WCET of τ_{C_i} in isolation mode from this point up to the end of execution, C^{FL}_{max} is the WCET in the full load mode until the next observation point, t_{cntr} is the total WCET of the proposed run-time control mechanism (the overhead to monitor the actual execution time, the WCET of the run-time controller, the overhead to send the request to the master and the time to suspend the tasks), and t is the actual execution time.

Dynamically defined points

In contrast to the previous static approach, the proposed control is executed at each core at different points, which are a priori unknown and decided during the execution of the tasks. Algorithm 2 depicts the functionality of the dynamic control mechanisms. First of all, the controller verifies if the execution is in full load mode (condition $C1$), and checks if the current point is active. When the execution changes from one ECFG to another ECFG, the corresponding point is always active, in order to keep track of the traversing of ECFGs. This is achieved by updating the variable *offset*, similar to the static version (L. 1 and L. 10). Then, the number of visited points (variable *counter*) is increased by one (L. 2). The controller checks whether this point is active, i.e., we have reached the number of inactive points given by variable *points* or it describes a traversal between ECFGs (condition $C3$). If the point is active, the controller: i) computes the RC^{ISO} (L. 4), ii) monitors the real execution time t (L. 5), iii) computes the values of points that can be skipped based on the existing slack (L. 6), and iv) when no more points can be skipped, it sends a request to the master for suspending the execution of the low criticality tasks (L. 7). Then, the counter is initialized to 0 (L. 8). Note that, the *counter* and the *offset* are initialised to 0 before execution.

RWCET computation: The algorithm for the RWCET computation takes as inputs the $STI_{\tau_{C_i}}$, the instrumented points, the current values of the loop iterators where the point is placed into (*iterators*) and the *offset*. Similar to the static version, the actual local level of the point p_{τ_i}, ll, is used to compute the RC^{ISO} at that level, $R[ll]$. The local level ll derives from the addition of the *offset* and the *level* of the point p_{τ_i}. If points have been skipped (condition $C3$), the RWCET of the skipped levels has to be computed. To achieve that, the controller finds the head points of the levels ll up to the level equal to the *offset* (L. 13). Then, the $R[i]$ for these head points is updated using the information of the loop iterators $iterator[y]$, $d[y]$ and $w[y]$ of each head point y. To compute $R[i]$ for a head point y with local level i, we subtract from the remaining time of the head point of the previous local level $i-1$ ($R[i-1]$) the time passed up to now. This time is derived by multiplying the $w[y]$ of the head point with local level i with the value of the loop iterator $iterator[y]$ and the $d[y]$ of the head point with local level i (L. 14). Then, the $R[ll]$ of the active point p_{τ_i} can be computed in a similar way (L. 16). Before execution, the *last_head* of *level*[1] is initialised with the initial point *start*, and $R[0]$ with the total WCET in isolation.

For instance, in Figure 2.15, the first active point is c before the first loop iteration c^0. It has a level equal to 1 and belongs to the main function, therefore $offset = 0$ and $ll = 1+0 = 1$. The RC^{ISO} for point c^0 is given by $R[1]=R[0] - (0 * C^{ISO}[c^{j-1} - c^j]) - C^{ISO}[start - c]=R[0] - C^{ISO}[start - c]$. Assuming that the second active point is in the sixth execution of the condition c_5, the RC^{ISO} of c^5 is computed by $R[1]=R[0] - (5 * C^{ISO}[c^{j-1} - c^j]) - C^{ISO}[start - c]$.

Safety condition: We extend the static safety condition to support the dynamic approach, by multiplying the variable C^{FL}_{max} with the number of skipped points *points*, i.e., $t + RC^{ISO}_{\tau_{C_i}} + (points * C^{FL}_{max}) + t_{\text{cntrl}} \leq D_{\tau_{C_i}}$. Based on this equation, we can compute at run-time the number of points that can be safely skipped until the run-time control has to be re-executed. When $points \leq 0$, the

```
Function Run-time_control(p_{τ_{C_i}}, STI_{τ_{C_i}}, counter, iterator)
    if (C_RT) then                                                              /* C1 */
1       if (type[p_{τ_{C_i}}] ==F_EXIT||F_ENEX) then offset = offset - level[p_{τ_{C_i}}] ;   /* C2 */
2       counter ++;
3       if (counter==points)||(type[p_{τ_{C_i}}] ==F_ENTRY||F_ENEX) then        /* C3 */
4           RC^ISO=Compute_dynamic_RWCET(p_{τ_{C_i}}, STI_{τ_{C_i}}, iterator, offset);
5           t = Monitoring_time();
6           points = Next_active_point(RC^ISO, t);
7           if (points ≤ 0) then Request_suspension() ;
8           counter = 0;
9       end
10      if (type[p_{τ_{C_i}}] ==F_ENTRY || F_ENEX) then offset = offset + level[p_{τ_{C_i}}] ;   /* C4 */
11  end
Function Compute_dynamic_RWCET(p_{τ_{C_i}}, STI_{τ_{C_i}}, iterator, offset)
    ll = offset + level[p_{τ_{C_i}}];
    if (points > 1) then                                                        /* C3 */
12      last_head[ll − 1] = h[p_{τ_{C_i}}];
13      for (i=ll-1;i> offset;i–) do last_head[i − 1] = h[last_head[i]] ;
14      for (i= offset + 1;i<ll;i++) do R[i] = R[i − 1] - (iterator[last_head[i]]) * w[last_head[i]] -
          d[last_head[i]] ;
15  end
16  R[ll] = R[ll − 1] - (iterator[p_{τ_{C_i}}]) * w[p_{τ_{C_i}}]) - d[p_{τ_{C_i}}];
```

Algorithm 2: Dynamic control mechanism at point $p_{\tau_{C_i}}$.

low criticality tasks must be suspended in the current active point, as not enough time slack exist to safely decide suspension at the next active point.

2.3.7 Evaluation

This section presents the main results for evaluation, while the complete evaluation can be found in [C7, J14]. We compare the performance gain of the proposed static and dynamic RWCET approaches compared to the isolated execution based on the observed execution time of the low criticality tasks tasks allocated on a core, i.e., $\frac{Makespan_{iso} - Makespan_{sta}}{Makespan_{iso}}$ and $\frac{Makespan_{iso} - Makespan_{dyn}}{Makespan_{iso}}$. We provide the number of active instrumentation points and overhead of the static and dynamic RWCET approaches. Here, we present the experimental results for one low criticality task running in parallel with the high criticality tasks, which is the hardest case as it provides the smallest slack between the C^{ISO} and C^{FL}. We explore the deadline of the high criticality tasks $D_{\tau_{C_i}}$, i.e., from tight deadlines close to the WCET of the high criticality tasks in isolation up to more relaxed deadlines, and the granularity of run-time control: i) coarse-grained, with points at the head points of nested level 1 (HP1), ii) medium-grained, with points at the head points of nested levels 1 and 2 (HP2), and iii) fine-grained, with points at the head points of all three nested levels (HP3).

Experimental set-up

Platform & implementation: We use the TMS multicore platform for the experiments, described in Table 2.5. Both static and dynamic approaches have been implemented as a bare-metal library, with low-level functions for the time monitoring of the on-going execution and for the sus-

pension/resuming of the low criticality tasks. A set of timing functions have been developed to read the current clock by accessing the control registers TCSL and TCSH of the local core clock. The suspension and the resume of the low criticality tasks is implemented using the event and interrupt mechanisms of the TMS. A set of event functions have been designed to configure the events and the interrupts of the TMS, to allow the use of the events by providing software setting, clearing and monitoring mechanisms for the events, and to keep suspended or resume the low criticality tasks.
Benchmarks: To experimentally evaluate the approach, we have conducted experiments using gemm as the high criticality task, executed on two cores. The gemm benchmark has been selected due to the regularity and the symmetry of each structure, which favours the static approach and under-privileges the dynamic approach. In this way, the experimental results provide a lower bound on the gains of the dynamic approach. A set of loop and data dominated low criticality tasks executed on the remaining cores, which consist of infinitive loops that perform read and write accesses to the memory.
WCET acquisition: Since no existing static WCET analysis tool supports the TMS platform, a measurement-based approach has been applied using the local timer of the cores that run high criticality tasks. To acquire the RC^{ISO} and the partial RWCETs we run only the high criticality tasks on the platform. To increase the reliability of the measurements, we have performed 50 times our experiments and maintained the maximum observed value for RC^{ISO} and the minimum observed value for the partial RWCETs. In addition, we increase the maximum observed value and decrease the minimum observed value by 10%. The same technique has been applied to compute C^{FL}_{max}, with the high criticality tasks executed in parallel with low criticality tasks.
Data placement: The memory configuration for all the cores is the following: i) the L1P, L1D, and L2 are configured as SRAMs for better predictability, and the stack, data and code sections (`.stack`, `.data`, `.text` ...) are allocated in the L2 SRAM. The inputs and outputs of the benchmarks are allocated in the DDR. In this configuration, interference occurs in the shared resources among the benchmarks executed over the platform.

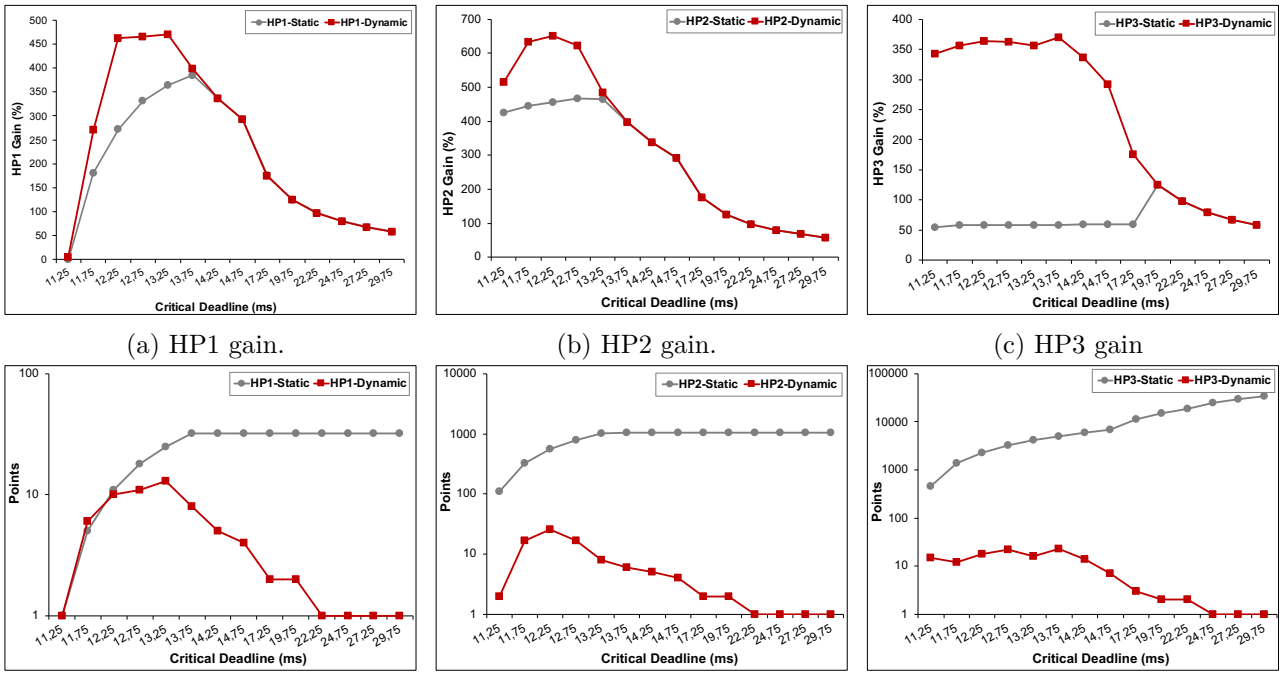
Performance Gains

Figures 2.17a, 2.17b and 2.17c present the gains in the execution time of the low criticality tasks achieved with the static and dynamic RWCET approaches, compared to the isolated execution with the HP1, HP2 and HP3 configurations and different deadlines. Overall, both approaches achieve significant gains. The static approach achieves a gain of, on average, 197,69% 277,76% and 68,22% for HP1, HP2 and HP3, respectively, while the dynamic approach 236,11%, 324,20%, and 241,83% for HP1, HP2 and HP3, compared to isolated execution, respectively.

Comparing the dynamic and static approach, the dynamic controller is called significantly less times, as depicted in Figures 2.17d, 2.17e and 2.17f. As a result, the overall overhead of the dynamic controller is reduced, leading to switching decisions that occur later than the static approach. The more time we execute in parallel high and low criticality tasks, the higher is the gain. Furthermore, high criticality tasks finish earlier in the dynamic approach, due to the reduced overhead because of less active points, and thus, more time is left for the execution of the low criticality tasks.

Controller cost

Table 2.9 depicts the maximum time overhead of the proposed approaches and the time-triggered execution. The dynamic RWCET control has higher overhead due to the re-computation of the $RC^{ISO}_{\tau_{C_i}}$ of the head points for the skipped points (full control).



(a) HP1 gain. (b) HP2 gain. (c) HP3 gain

(d) HP1 active points. (e) HP2 active points. (f) HP3 active points.

Figure 2.17: Comparison of RWCET static and dynamic approaches w.r.t. the gain of the execution time of the low criticality tasks compared to the isolated execution and the number of active points.

Table 2.9: Risk-permissive run-time adaptation controller time overhead (in cycles)

	Read timer	**Light control**	**Full control**	**Suspend**
Static RWCET	70	-	501	200
Dynamic RWCET	70	150	1551	200

2.4 Conclusions

Execution of real-time systems on multicore architectures requires guarantees for meeting deadlines. At the same time, energy efficient has become also important. To guarantee timely and energy efficient execution, efficient deployment solutions are needed. We propose a set of decomposition-based approaches. Initially, we design an algorithm to provide the optimal solution for independent IC tasks executed on a SMP [C10], which has been extended for platforms with DVFS capabilities [J15]. We have leveraged our decomposition-based solution for dependent IC tasks and heterogeneous multicore platforms and proposed an accelerated, but still optimal, version of our decomposition-based method [J16]. Last, optimal and heuristic approaches considering task migration are proposed in order to take advantage of AMPs, such as big.LITTLE platform [C16].

To reduce the WCET pessimism, *is*WCET estimations have been used, which are only valid for a specific schedule solutions. To maintain the *is*WCET solution during execution, time-triggered execution is usually used, which, however, does not allow any performance improvement when the tasks finish earlier than their *is*WCET. To support a safe adaptation of interference-sensitive schedule solutions, we proposed a run-time approach that enables parallel execution of the control phases on each core with a fine-grained protection [C18]. Our second contribution [C21] comes from

the observation that by enforcing the partial order of tasks, we limit the performance improvement that can be achieved through run-time adaptation. To further improve performance gains, we leverage our approach with a safe relaxation of the partial order of tasks.

Existing approaches are based on WCET estimations obtained during design-time, and thus, they are not able to take advantage of the actual execution progress of the tasks. To deal with this limitation, we proposed an approach that computes dynamically new safe estimations of the WCET during execution, based on the task progress. The updated WCET estimations are used to derive the available time-slack and postpone mode switch [C8, C7, C9, W1]. The proposed approach has been leveraged in order to dynamically decide when to invoke the controller, reducing the overhead introduced due to the execution of the controller, further increasing the gains [J14].

Chapter 3

Fault-aware techniques for hardware design

This chapter summarises our contributions on system reliability under hardware faults. More precisely, Section 3.1 presents the proposed hardware mechanisms to perform instruction level fault tolerance through dynamic re-scheduling in VLIW processors, under short transient, long transient and permanent faults, conducted during the PhD period of Rafail Psiakis [195]. Section 3.2 describes the proposed hardware mechanisms to perform data level fault tolerance by data shuffling in order to reduce the impact of permanent faults over NoC on multicore and manycore platforms, performed during the PhD period of Romain Mercier [155]. Section 3.3 presents our cross-layer reliability analysis for complex hardware systems against transient faults due to radiation, designed during FLODAM project. Section 3.4 summarises the aforementioned contributions.

3.1 Run-time instruction re-scheduling for VLIW processors

3.1.1 Context

Instruction Level Fault Tolerance (ILFT) improves the processor reliability and it can be implemented through Hardware (HW) or Software (SW). ILFT can be achieved through HW redundancy, where additional FUs are inserted to the original processor, in order to execute in parallel the same instructions and compare the obtained results [128]. Although small performance overhead is normally observed due to the comparison of the results, the area overhead is significant. Through SW redundancy, the program is modified by inserting replicated instructions, which will be executed on the original processor. Although the area is not increased, the impact on the execution time can be significant [207]. A better area and performance trade-off for ILFT can be achieved over processors with several FUs, such as Very Long Instruction Word (VLIW) processors. VLIW processors have several issues, which can process instructions in parallel. However, the Instruction Level Parallelism (ILP) of applications is typically limited and variant in time, while the VLIW issues consist of different types of FUs. As a result, not all VLIW FUs will be used at the same time during the application execution.

In this context, idle FUs can be used to execute replicated instructions or re-execute faulty instructions, improving the processor reliability and the performance overhead of fault-tolerant approaches.

Table 3.1: Comparison with representative SoA fault tolerant VLIW approaches.

Ref.	Replication						Re-execution		VLIW		
	F	P	SW	HW	Det.	Cor.	All	Healthy	Hom.	Het.	Coupled
[36]	✓		✓		✓					✓	
[158]		✓	✓		✓					✓	
[161]		✓	✓		✓	✓				✓	
[112, 129, 135]		✓	✓	✓	✓					✓	
[233]	✓			✓	✓		✓				✓
[232, 235]		✓		✓	✓		✓				✓
[234]		✓		✓	✓		✓		✓		
[59, 60]	✓			✓	✓	✓			✓		
[238]	✓		✓	✓	✓			✓	✓		
[C12, J26]	✓			✓	✓	✓				✓	
[C11]	✓			✓	✓	✓		✓		✓	
[C17, C15]								✓		✓	

3.1.2 State-of-the-Art

Table 3.1 categorizes representative ILFT approaches on VLIW processors. The instructions can be Fully (F) or Partially (P) replicated, whereas the replication can be preformed statically, through software (SW), dynamically, through hardware (HW), or combining both in a hybrid way. Duplicating instructions detects an error (Det.). Correction (Cor.) is performed either by triplicating instructions or by instruction re-execution, excluding or not the faulty FUs.

Static approaches are usually implemented by the compiler, which replicates the instructions and inserts instructions for comparison, e.g., full duplication and comparison is done in the compiled code [36]. Although static approaches can theoretically obtain a, as dense as possible, schedule, the code and storage size are increased, having a negative impact on system reliability. To partially reduce the number of additional instructions, approaches duplicate only a part of the instructions and implement the comparisons in hardware. For instance, the compiler excludes control flow instructions from duplication, and distributes error detection overhead across VLIW clusters [158] and uses the instruction fault masking capability to select which instructions to duplicate [161]. Partial instruction duplication to maximize the number of duplicated instructions, within a performance overhead bound, is performed by the compiler, while the comparison is performed by the hardware [129, 112]. In [135], the compiler encodes information in the instructions and a hardware mechanism decodes the information to run-time duplicate the instructions. In [238], instruction duplication is done by the compiler and the comparison of results is done by the hardware. If an error is detected, the hardware adds a time slot and re-executes the instruction to another FU.

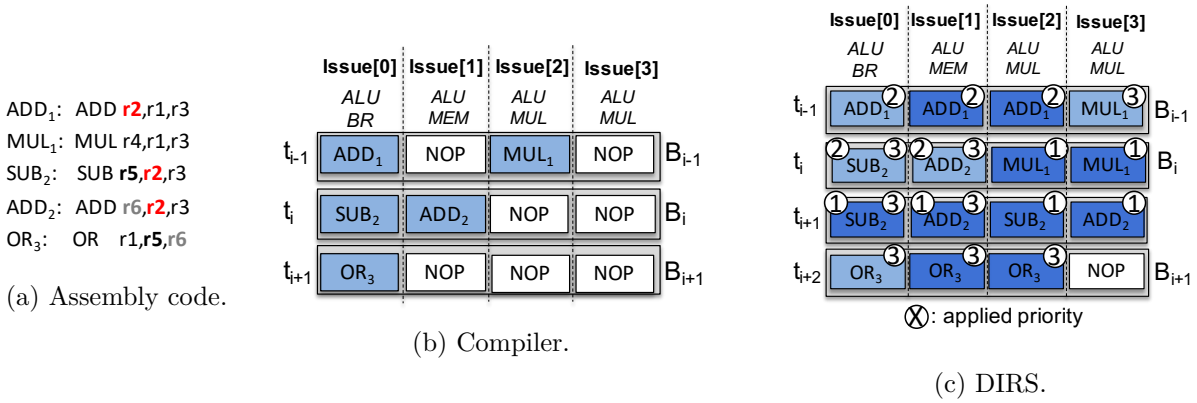
Dynamic approaches eliminate the need for high storage requirements and additional instructions. They are usually implemented through hardware that replicates and schedules the instructions during execution. Some hardware mechanisms avoid the need of dynamic scheduling by using VLIWs that have coupled issues, one for executing the original instructions and one for potentially executing the duplicated instructions. In this way, the duplicated instructions follow the schedule given by the compiler. Full instruction duplication is applied and when an instruction bundle has more instructions than the half of its issue-width, the bundle is divided into two and a time slot is added [233]. Partial instruction duplication is performed by not duplicating instructions for which there is no idle coupled issue [232, 235]. However, such approaches require coupled VLIW issues, being less flexible.

The remaining hardware approaches apply dynamic scheduling, assuming homogeneous VLIW, where all issues include all types of FUs, and thus, can execute any instruction. For instance, when enough resources do not exist to execute twice the scheduled instructions, the instructions are partitioned in groups that are executed sequentially, and the use of spare FUs is explored to reduce performance degradation [59, 60]. However, when the VLIW consists of issues with different type of FUs, the existing schedulers are not applicable, as they ignore the type of FUs. In such cases, a heterogeneous VLIW can be transformed to a homogeneous one, by inserting the missing FUs at each issue, e.g., [234]. With increasing number of VLIW issues and considering complex FUs, e.g., floating point FUs, these solutions lead to VLIW processors with significant area overhead.

3.1.3 Contributions

To address these limitations, a hardware mechanism is proposed to perform fault correction through Dynamic Instruction Replication and Scheduling (DIRS) by exploring at run-time the idle resources, inside and across consecutive instruction bundles, for heterogeneous VLIW processors [C12]. We will illustrate the proposed approach through an example. Let's assume a 4-issue VLIW processor, as the one depicted in Figure 3.3, with the following configuration: one Arithmetic Logic Unit (ALU) and one Branch unit (BR) in the first issue, one ALU and one Memory FU (MEM) in the second issue and one ALU and one Multiplication unit (MUL) in the third and fourth issues, on top of decode (DC) and Write-Back (WB) FUs. The assembly instructions are depicted in Figure 3.1a. Figure 3.1b shows the corresponding schedule given by the compiler, with three instruction bundles, B_{i-1}, B_i and B_{i+1}. Figure 3.1c shows the execution obtained by the proposed DIRS, when instruction triplication is applied and instructions are scheduled based on the type of FUs in the current and next bundle. The light (dark) blue boxes represent original (replicated) instructions. The DIRS approach is extended towards a cluster-based design to tackle the issues of scalability, while maintaining a reasonable area and power overhead [J26].

The proposed approach efficiently deals with short transient faults, i.e., faults with a duration less than one clock cycle. However, when faults become persistent, applying instruction triplication without taking into account the FU status, i.e., healthy or faulty, may lead to problems. Such an example is illustrated in Figure 3.2b, where all replicas of MUL_1 operation are scheduled in the third issue. In this case, we cannot deal with the persistent error which occurs in the MUL unit of the third

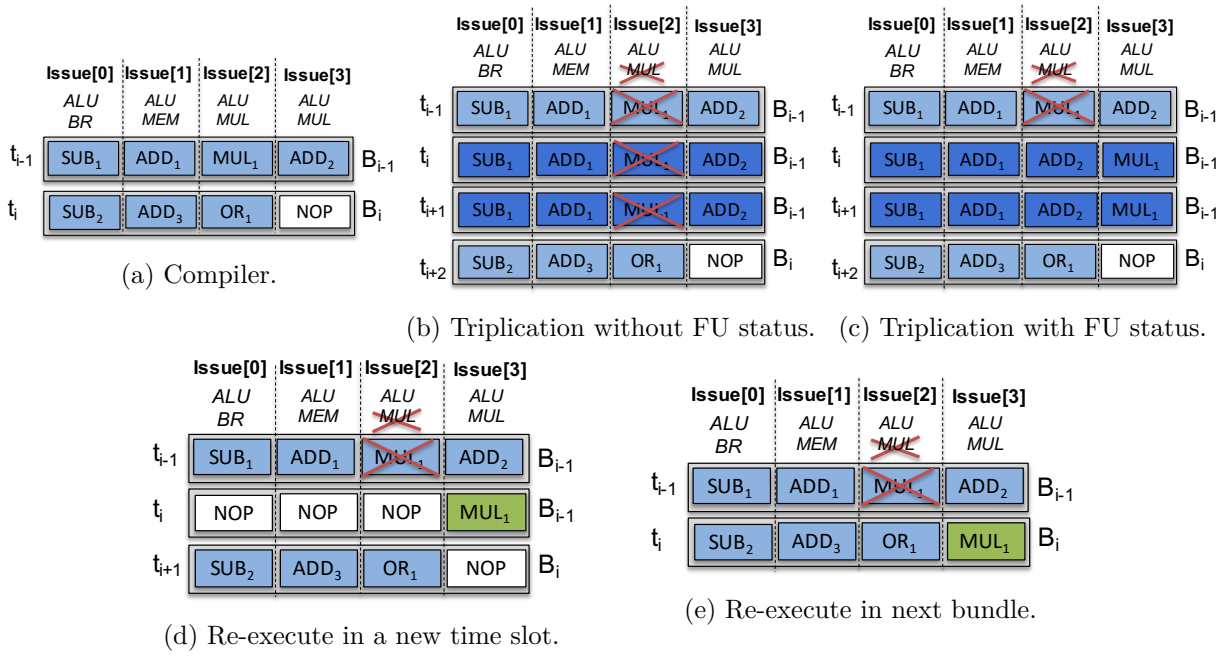


(a) Assembly code.

(b) Compiler.

(c) DIRS.

Figure 3.1: a) Assembly instructions and register dependencies. Corresponding execution by the b) compiler and b) DIRS approach [C12, J26] for an 4-issue VLIW.



(a) Compiler.

(b) Triplication without FU status.

(c) Triplication with FU status.

(d) Re-execute in a new time slot.

(e) Re-execute in next bundle.

Figure 3.2: Execution based on a) compiler, b) DIRS [C12, J26], c) DIRS-CG [C11], d) instruction re-execution at a new slot [238] and e) DIS [C17, C15].

issue. Therefore, we extend the proposed approach in order to take into account the state of the FUs in a coarse-grained way (DIRS-CG), considering single and multiple permanent faults [C11]. As soon as a FU is detected as faulty, it is excluded permanently from the scheduling, and original and replicated instructions are re-binded to the healthy FUs, as illustrated in Figure 3.2c. Note that, the ALU of the third issue is still used, and only the MUL FU is excluded. However, instruction replication inserts significant performance overhead. To decrease the performance overhead and to support not only permanent, but also single and multiple Long-Duration Transient (LDT) faults, the hardware mechanism is leveraged with a transistor level fault detector able to detect active faults. With this information the proposed mechanism performs Dynamic Instruction Scheduling in a Coarse-Grained way (DIS-CG) by temporally excluding the faulty FUs [C15]. When the fault faints, the excluded FUs can be reused. The proposed approach is further enhanced in order to exploit the FUs in a fine-grained way (DIS-FG), i.e., by dividing internally the FU into components whose status is monitored [C17]. Existing approaches only re-execute the faulty instruction to a new time slot as depicted in Figure 3.2d, whereas DIS exploits the idle slots in the next bundle.

3.1.4 System model

Our system is a heterogeneous VLIW processor. Figure 3.3 illustrates an example of a 4-issue VLIW processor having a 3-stage pipeline with Fetch (F), Decode (DC) and Execute/Memory-WriteBack (EX/M-WB), and the corresponding FUs.

3.1.5 Fault model

As the probability of error occurrence is generally proportional to the area of the circuit, we focus mainly on faults occurring in the combinational logic of the execution stage, since these are the

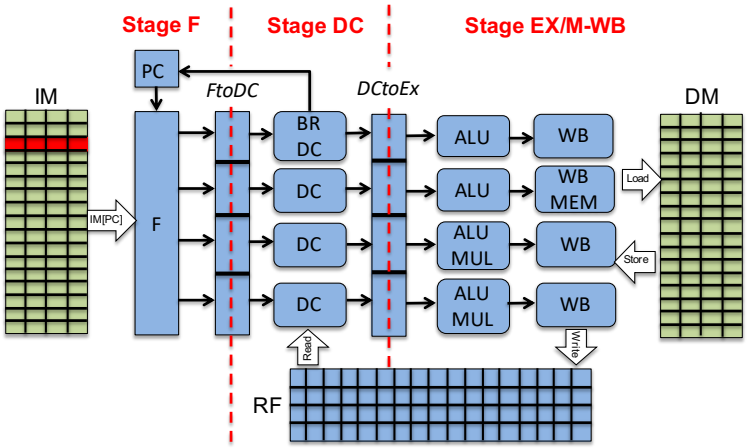


Figure 3.3: Original VLIW processor.

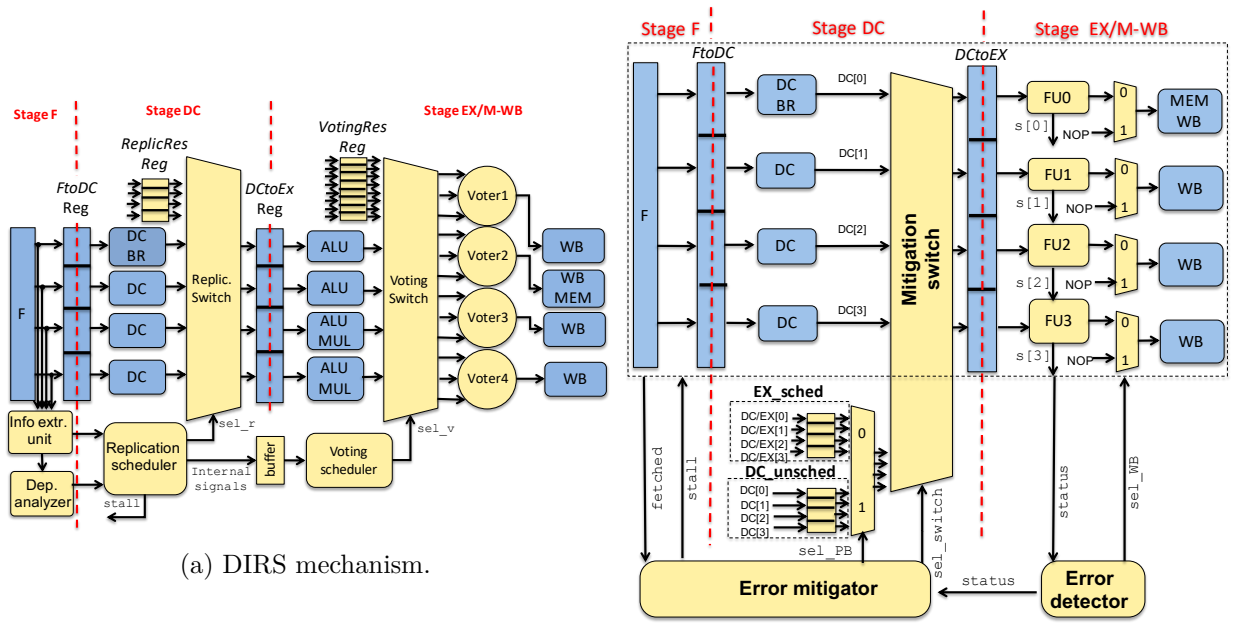
components with the higher area in the VLIW datapath. We assume that the register file, pipeline registers and the memories are protected by Error Correction Codes (ECC). We focus on single bit transient faults occurring due to radiation in [C12, J26], where instruction triplication is applied. This approach is extended for single and multiple permanent errors due to radiation and aging effects by excluding permanently the faulty FUs under instruction duplication and triplication [C11]. The approach in [C15, C17] uses a transistor level fault detection mechanism to detect active faults and performs instruction re-scheduling by excluding temporary the FU in a coarse-grained way [C17] and in a fine-grained way [C15].

3.1.6 Dynamic instruction replication and scheduling mechanism

The proposed DIRS approach replicates instructions and dynamically schedules original and replicated instruction on the FUs, within a scheduling window of two instruction bundles. Figure 3.4a depicts the proposed architecture, where the yellow boxes indicate the DIRS hardware components. The processing components are the replication switch, the voting switch, and the voters. The control components are the information extraction unit, the dependency analyzer, the replication scheduler, and the voting scheduler. The storage components are the ReplicRes register and the VotingRes register. The components, that do not necessarily require to be placed inside the VLIW datapath, are designed to run in parallel and to have a smaller critical path than the VLIW pipeline stages, in order to not affect the clock frequency. The remaining components, that must be obligatory added in the VLIW datapath to support the functionality of the proposed approach, are designed with reduced critical path and are placed in different pipeline stages, so as to reduce the impact on the overall clock frequency. The next paragraph describe the functionality of DIRS hardware components.

Information extraction unit: It performs an early decoding in the F stage and provides the information regarding the bundle, the issue number and the instruction type, to the dependency analyzer and the replication scheduler.

Dependency analyzer: It identifies dependent instructions between two concurrent bundles, i.e., an instruction in bundle B_{i-1} that uses, as destination register, a destination or source register of an instruction in bundle B_i. Parallel execution of the dependent instructions is forbidden. Independent instructions of B_{i-1} can be postponed and scheduled at any idle FUs of the next



(a) DIRS mechanism.

(b) DIS mechanism

Figure 3.4: VLIW datapath enhanced with a) DIRS [C12, J26] and b) DIS [C15, C17].

bundle. Using the example presented in Figure 3.1a, MUL_1 of bundle B_{i-1} is independent, since none instruction of bundle B_i (SUB_1, ADD_2) uses the destination register of MUL_1 as destination or source register. However, SUB_2 and ADD_2 of bundle B_i both depend on ADD_1, since they read register $r2$, updated by ADD_1. In a similar way, the instruction OR_3 of B_{i+1} reads registers $r5$ and $r6$, which are used as destination registers by SUB_1 and ADD_2 of B_i.

Replication switch & scheduler: The replication switch propagates previously decoded instructions (stored at the ReplicRes register) and the currently decoded instructions to the pipeline DCtoEX register, following the dynamic schedule provided by the replication scheduler. The replication scheduler dynamically reschedules original and replicated instructions, avoiding the insertion of additional time slots, by postponing the execution of independent instructions to the next bundle. The scheduler operates according to three priorities, applied in the following order: ① instructions of a previous bundle have a higher priority than instructions of the current bundle, ② the dependent instructions have a higher priority than the independent instructions, and ③ instances of different instructions of the same bundle have higher priority than the replicated instances of the same instruction. For instance, the example presented in Figure 3.1c depicts which of the three priorities the hardware scheduler uses to obtain the schedule. At the time slot t_{i-1}, the original, the first replica and the second replica of ADD_1 are scheduled, due to priority ②. Then, the original MUL_1 is scheduled, due to priority ③. Since the remaining instructions from bundle B_{i-1} are independent, they are allowed to be scheduled alongside with the instructions of the upcoming bundle B_i. Due to priority ①, the first and second replica of the MUL_1 of bundle B_{i-1} are scheduled at the time slot t_i. Then, each of the original SUB_2 and ADD_2 are scheduled, due to priorities ② and ③. However, the remaining unscheduled instructions of B_i are dependent, and thus, they cannot be executed with the upcoming bundle B_{i+1}. Hence, a new time slot has to be added. At this new time slot t_{i+1}, the remaining dependent and redundant instructions of B_i are scheduled based on

priority ① and ③. At t_{i+2}, no instruction remains from the previous bundle, thus original and replicated OR_3 are scheduled, according to priority ③.

Voting switch, voting scheduler and voters: The voting switch propagates the results of replicated and original instructions. If all replicated instructions have been executed, the results are send to the voters, otherwise to the VotingRes register, to be stored for a later commint. The voting scheduler is responsible for synchronizing and grouping the original and the replicated instructions. By upfront grouping the results in adjacent positions of the VotingRes register, the voting switch connections are significantly simplified, compared to common switch designs. The scheduler is implemented through a comparison-based sorting algorithm. The voters compare the results and mask any occurred error.

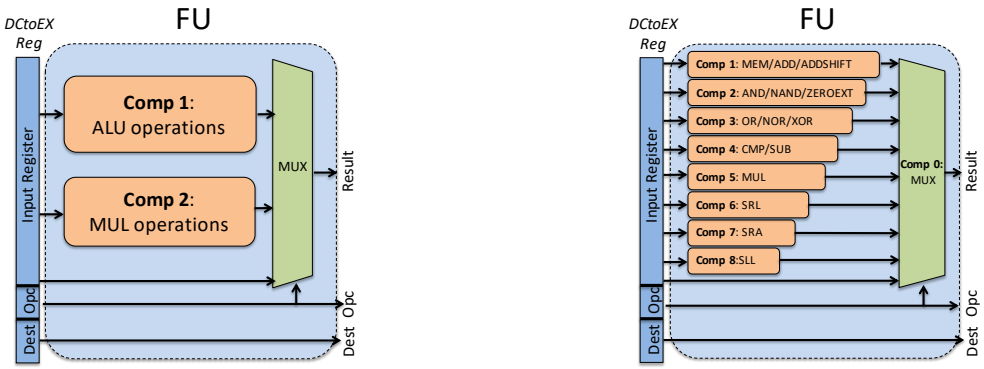
3.1.7 Cluster-based instruction replication and scheduling mechanism

By analysing the area overhead of the components of the DIRS mechanism, we observe that the most area costly components are the switches, which can have negative impact on the clock frequency. It should be stressed that this is true for any approach that performs dynamic re-scheduling of the instructions, since this type of techniques requires to dynamically dispatch the instructions to different issues. Although we have carefully designed and positioned DIRS switches, with the increase of the number of issues, the complexity of the switches is also increased. To reduce this overhead, we divide a VLIW into several smaller clusters, where the DIRS is internally applied. Note that, the compiler usually schedules the instructions as dense as possible in order to occupy less area in the memory. Thus, the compiler schedules the instructions by prioritizing the first clusters, leading to unbalanced distribution of the instructions to the clusters. To remove this negative impact, a virtual VLIW configuration is generated by randomly shuffling the FUs position. The virtual configuration is provided to the compiler and a static de-shuffling takes place according to the real VLIW cluster configuration at fetch stage.

3.1.8 Coarse-grained and fine-grained dynamic instruction scheduling mechanism

We leverage the aforementioned hardware mechanism to exclude permanently [C11] or temporally the faulty FUs [C15, C17]. This section presents the proposed architecture, where fault detection is performed by a transistor-level detector, and the faulty FUs can be temporally excluded. Figure 3.4b describes the additional hardware components of the proposed architecture: the error detector to obtain the status of FUs depending on active faults, the error mitigator to perform dynamic instruction scheduling and two shadow registers, i.e., the DC_unsched for the decoded unscheduled instructions and the EX_sched for the executing instructions.

Error detector: It keeps the faulty status of the FU components and identifies new fault occurrences. FUs are analyzed in gate-level to identify the individual circuits. We group the individual circuits based on the instruction opcode into FU components in a coarse-grain way (DIS-CG), as depicted in Figure 3.5a [C15], and in a fine-grained way (DIS-FG), as depicted in Figure 3.5b [C17]. Each component and the final multiplexer (comp 0), which selects the result of the executed operation according to the opcode, is assumed to be enhanced with Built-In-Current-Sensors (BICS) [42]. BICS are able to monitor the induced transient currents to detect a fault, and set (reset) bits in the signal status to inform that a FU component is currently faulty (healthy). A multiplexer exists in each issue to discard the results that are miscalculated in case of an error. The status is passed to the error mitigator to perform instruction re-scheduling, accordingly.



(a) Coarse-grained FU components.

(b) Fine-grained FU components.

Figure 3.5: a) Coarse-grained and b) fine-grained decomposition of FUs.

Error mitigator, shadow registers and mitigation switch: The instructions in the FtoDC register are decoded at cycle k − 1. Based on the status signal, the error mitigator schedules the instructions to be executed at the next cycle k. The error mitigator uses the information extraction unit and the dependency analyser, as in DIRS. The error mitigator performs the scheduling similar to the DIRS approach, but it does not perform instruction replication and takes into account the status of the FUs. Thus, the instructions to be scheduled can potentially come from three sources: 1) the currently decoded instructions, 2) the remaining, not yet scheduled, instructions of the previous cycle, and 3) the potentially faulty executed instructions of the current cycle. The decoded instructions not scheduled for execution are stored in the DC_unsched shadow register. The instructions executed at the current cycle are backed-up in the EX_sched shadow register, in case a fault occurs during their execution. A switch is required in order to implement the assignment of the scheduled instructions into issues. To decrease the switch complexity, we divide it into two parts: i) a multiplexer to select which shadow register will be used as an input to the mitigation switch, and ii) a mitigation switch, which passes the instructions from the shadow registers and the decoded instructions to the main pipeline DCtoEX register.

3.1.9 Evaluation

We compared the VLIW processor enhanced with the DIRS approach performing scheduling in the current and next bundle (DIRS-CNB), the DIRS approach performing scheduling only in the current bundle (DIRS-CB), the unprotected VLIW and the VLIW where the FUs are triplicated (3FU). Furthermore, we present the performance overhead of the proposed DIS-CG and DIS-FG approaches, compared to the unprotected version. Here, we present the results for the 4-issue configured with 2 MUL, 8 ALU, 1 MEM and 1 BR FUs, whereas all results can be found in [C12, C11, C17, C15, J26].

Experimental set-up

Platform & implementation: The VEX VLIW processor is used as a case study. All approaches have been developed in C++ and synthesized using the Catapult High Level Synthesis (HLS) tool to obtain the RTL design. The gate-level netlist was generated by the Design Compiler of Synopsys using 28 nm ASIC technology.

Benchmarks: We selected Mediabench as a workload to evaluate the behavior of the proposed approach as it is the one of the popular multimedia benchmark suite. The behavior of the proposed approach depends on the number of idle issues left by the application and the compiler. We present the results for ten applications with different characteristics, i.e., applications with low ILP and several multiple dependencies (adp_dec, adp_denc), high ILP and different dependencies (bcnt, dct, fft32x22s, matrix_mmul) and low ILP and less multiple dependencies (huff_ac_dec, motion,fir, crc). The benchmarks have been compiled with VEX compiler.
Fault injection: We randomly injected multiple faults during the benchmarks' execution and each benchmark is tested 0 up to 4 multiple faults, which is the maximum number of concurrent faults under which the application can still be executed on the 4-issue VLIW. The performance results are obtained by taking the mean value of 20 simulations running the same benchmark, but the faults are injected at random cycles for each simulation.

Dynamic Instruction Replication and Scheduling mechanism

Performance: Figure 3.6 depicts the performance overhead in execution cycles compared to the unprotected execution and 3FU execution. These values provide the impact on execution time, when all approaches are using the same frequency. The smaller the value, the better is the approach. The overhead of DIRS-CB is above 100% for almost all benchmarks (except crc benchmark), with an average of 152.95%. The minimum overhead of DIRS-CNB is 21.34% ($huff\_ac\_dec$ benchmark) and the maximum 139.16% (matrix_mul benchmark), with an average of 77.52%. Last, but not least, the speed-up of DIRS-CNB compared to DIRS-CB is from 13.47% (matrix_mul benchmark) up to 43.68% (huff_ac_dec benchmark), with an average of 29.93%.
Hardware overhead: Table 3.2 depicts the area of the different proposed mechanisms, implemented with a target frequency of 200MHz and the area overhead compared to the unprotected original version. The 3FUs approach has the maximum area overhead, i.e., 44.60%. The DIRS-CB has the lower overhead, with an area increase of 15.56%. The DIRS-CNB, compared to the unprotected processor, has an area increase of 23.54%. Table 3.2 also provides the critical path and the overhead compared to the unprotected original processor. The critical path in all approaches is

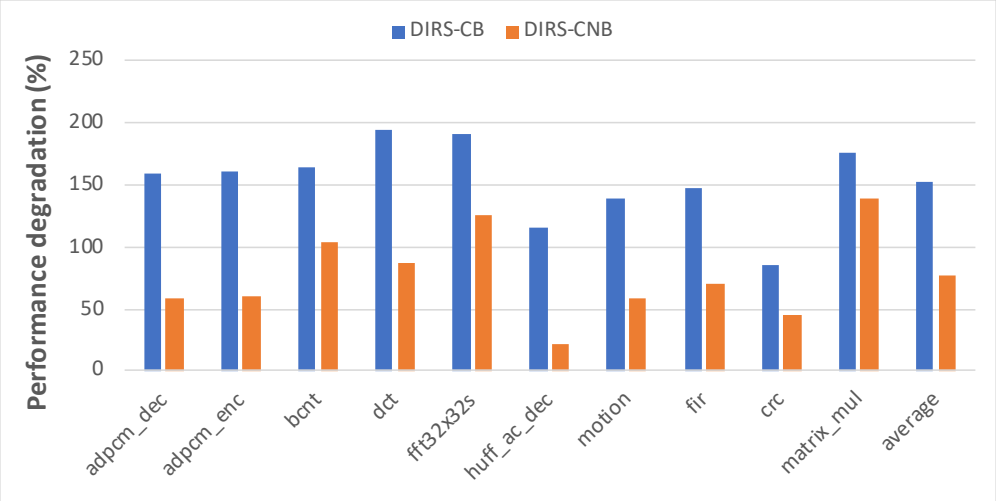


Figure 3.6: Performance degradation under DIRS-CB and DIRS-CNB.

Table 3.2: Area and critical path delay overhead.

Approach	Cells (μm^2)	Overhead (%)	Delay (nsec)	Overhead (%)
Original	50,844	-	2.62	-
3FUs	73,523	44.60	2.89	9.92
DIRS-CB	58,819	15.68	3.22	22.09
DIRS-CNB	62,812	23.54	3.22	22.09
DIS-CG	60,143	18.29	2.65	1.14
DIS-FG	62,314	22.56	2.65	1.14

given by the EX/M-WB stage. The delay impact of the 3FU is 0.27 ns due to the voters inserted for comparison. The delay impact of DIRS-CB and DIRS-CNB approaches is increased to 0.33 ns due to the voting switch required to correctly group the results for comparison. The replication switch placed in the DC stage does not increase the critical path.

Coarse-grained and fine-grained dynamic instruction scheduling mechanism

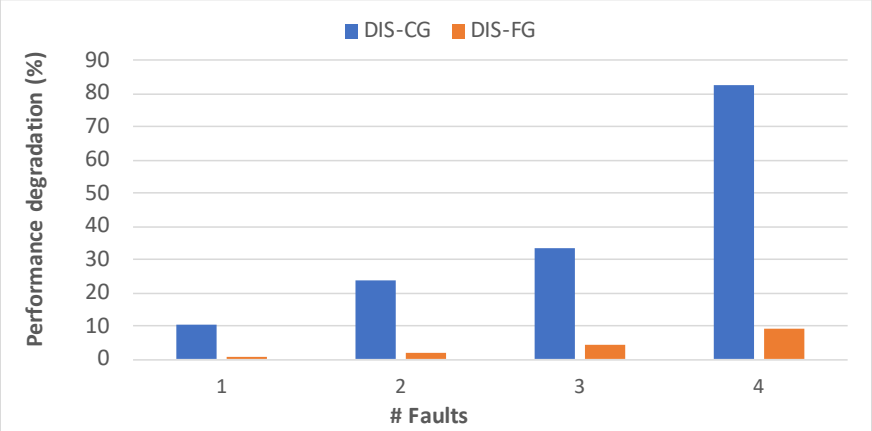


Figure 3.7: Performance degradation, compared to fault-free execution, taking into account the status of FUs in a coarse-grained [C15] and fine-grained [C11] way.

Performance: When no faults occur, both DIS-CG and DIS-FG approaches have the same performance, i.e., the original execution cycles. On the contrary, DIRS approaches increase the number of execution cycles even if no error occurred, due to the execution of replicated instructions, as shown in Figure 3.6. From Figure 3.7, we observe that DIS-FG inserts significantly lower overhead than the coarse-grained approach. We have also observed that in several benchmarks our performance is very close to the original one, i.e., without faults, even for several multiple faults. In contrast to the coarse-grained approach, the gain of the fine-grained mechanism is achieved because whenever a fault is detected, the proposed approach is capable of still utilizing the healthy FU components in the current and the next instruction bundle.

Hardware overhead: As shown in Table 3.2, DIS-CG and DIS-FG have 18.29% and 22.56% area overhead, respectively. Furthermore, the voting switch and the voters have been replaced by a multiplexer in DIS, leading to a critical path to 2.65ns reducing the overhead at 1.2%, respectively.

3.2 Run-time data shuffling for NoC

3.2.1 Context

Multicore architectures consist of several cores, requiring a high amount of data transfers, which cannot be handled by conventional communication means. To address this limitation, NoC appeared as a scalable solution for communications. Due to transistor shrinking and core number increasing in SoC, fault tolerance has become essential. Not only cores and memory, but also NoC became more sensitive to permanent faults. During system operation, aging defects [132], like electromigration, Bias Temperature Instability (BTI), Hot Carrier Injection (HCI), Time-Dependent Dielectric Breakdown (TDDB) and radiations [1] become additional sources of permanent faults. Faults occurring to NoC of those systems have a significant impact, due to the high amount of data crossing the NoC for communication.

In this context, fault tolerant techniques are required to remove the impact of permanent faults on NoC.

3.2.2 State-of-the-art

The majority of existing approaches focus on removing completely the impact of faults through fault mitigation/correction, while a few approaches focus on reducing the fault impact when the application accepts approximations. However, existing fault correction approaches cannot efficiently address several permanent faults on NoC, due to their high hardware costs.

Table 3.3: Comparison with representative SoA fault tolerant NoC approaches.

Ref.	Mitigation		Correction		Reduction
	New path	New resources	Circuit replication	Information redundancy	
[88, 63]	✓				
[63]	✓	✓			
[125, 83]		✓			
[82, 166]			✓		
[56, 280, 147, 228]				✓	
[173]					✓
[J23, C22, C24]					✓

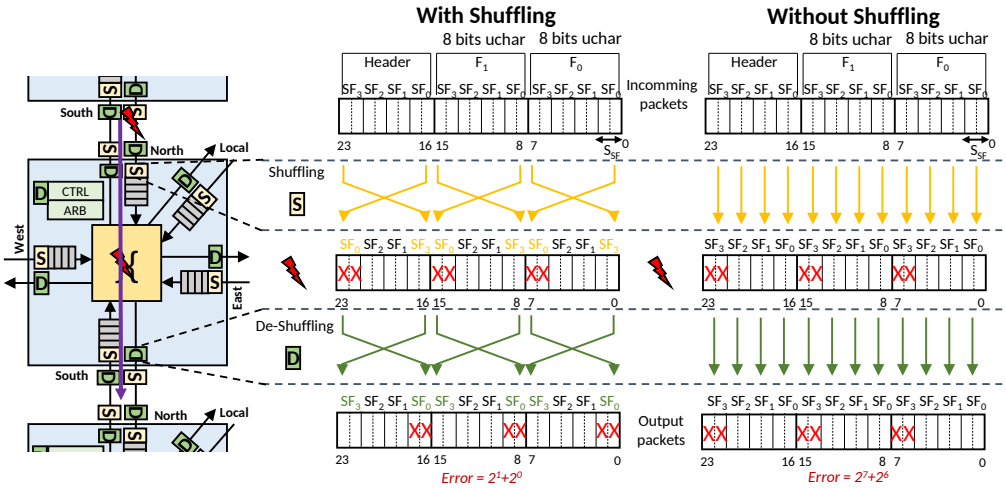
Table 3.3 summarises representative SoA approaches to obtain reliable NoCs. Existing techniques perform fault mitigation through i) routing algorithms, and ii) hardware reconfiguration using spare resources or default backup path, and fault correction through iii) circuit replication and iv) information redundancy. Routing algorithms are used to avoid faulty paths or faulty regions in NoC, keeping only the healthy resources [88]. Typical techniques are adaptive routing algorithms [63] including rules to avoid congestion and deadlock during packet transmissions. Reconfiguration replaces a faulty element of the NoC with spare resources at different levels [125]. Other reconfiguration approaches use default-backup paths to avoid data corruptions and packet re-transmissions [83] with low area and power consumption. Last, NoC can be reconfigured in degraded mode, using only the remaining healthy resources [63]. Circuit replication, called NMR, replicates N times, fully or partially, the architecture and votes the replicated outputs. The most popular approach is TMR [82], where a module is replicated three times. Despite several approaches which focus on reducing the hardware costs [166], the area and power consumption overhead remains

significantly high, e.g., more than three times for TMR. Information redundancy inserts additional bits inside messages using ECC. The most commonly used coding scheme for NoC is the extended Hamming code, which can detect two faulty bits and correct only one. Despite the increase of the bus size of the complete NoC, Hamming code is efficient for correcting single faults [147]. The number of correctable faulty bits can be increased by encoding the message on two dimensions [56] and interleaving several ECC [280]. Although the aforementioned approaches are efficient for single permanent fault, they are less adequate for multiple permanent faults and large NoC, since they induce high hardware costs while their mitigation capabilities are limited. Spare resources can be used only once and faults in the same module are masked, leading to approaches with large area and power overhead that tolerate few faults. Using ECC to correct more than one bit dramatically increases the hardware costs [228], limiting the use of ECC approaches against multiple faults.

Furthermore, few techniques have as interest to reduce the impact of faults for NoC, leading to approximated results, targeting applications that can tolerate errors until a certain level, such as image processing and machine learning [7]. For instance, an approach statically changes the assignment of lines in datapath busses, by placing the MSB on the borders of the bus, to attenuate the electromagnetic influences between neighbored lines [173]. As the assignment of lines is static, it cannot be modified at run-time to deal with new occurring faults during system execution.

3.2.3 Contributions

We propose a Bit-Shuffling (BiSu) approach that reduces the impact of multiple faults by shuffling at run-time the bits inside a flit transmitted through the NoC, with the goal of placing the faults on the Low Significant Bit (LSB) [C22, C24]. As depicted by the purple arrow of Figure 3.8-a, flits are crossing a faulty router from north to south. Flits of size S_F bits are divided into N_{SF} blocks of S_{SF} bits, named SubFlit (SF), e.g., $S_F = 8$, $S_{SF} = 2$, and $N_{SF} = 4$ (SF_0 to SF_3) in this example. Let's assume two permanent faults occur in the input buffer, affecting the bits 7 and 6 of all incoming flits. When no shuffling takes place (right part Fig 3.8-b) the errors are within the range $\{0, \pm 64, \pm 128, \pm 192\}$, depending on the initial value of the affected bits. When bit-shuffling is applied in the input ports of the router, before crossing the faulty path, the subflits



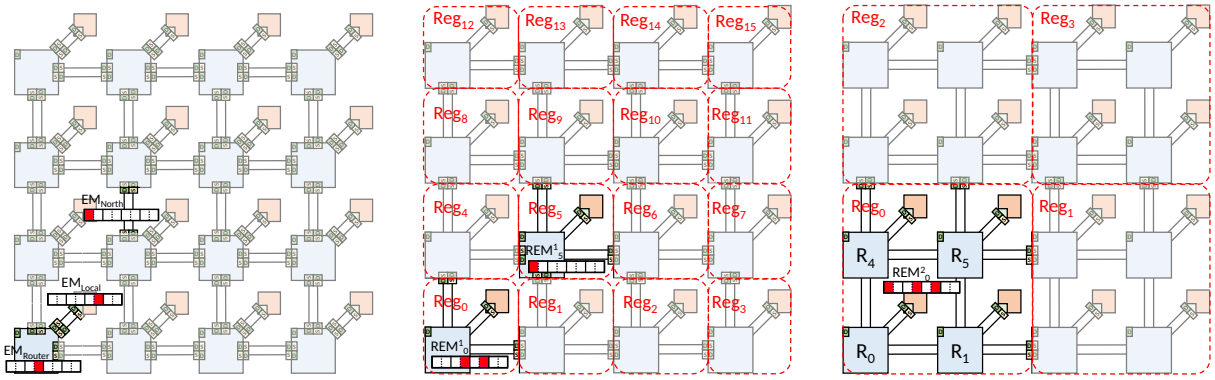
a) Original NoC extended with proposed approach.

b) Illustration of the bit-shuffling technique. (S_{pck} = 16, S_F = 8, S_{SF} = 2)

Figure 3.8: NoC extended with the BiSu method.

are re-organized by swapping LSB and Most Significant Bit (MSB) of data to allocate the MSB at the non-faulty hardware paths, i.e., SF_0 and SF_3 are swapped inside each flit. Hence, the impact of the faults is reduced to the range $\{0, \pm1, \pm2, \pm3\}$, depending on the values of the LSB. Before the flit leaves the router, the subflits are brought to their initial position, and the flit is sent to the output port.

BiSu protects every router and interconnection, as displayed in Figure 3.9a, providing a fine-grained protection, which, however, has increased hardware cost. To reduce the hardware cost, while providing data protection, we propose a Region-based Bit-Shuffling technique (R-BiSu) [J23], which divides NoC into regions and applies BiSu at the region borders. Figure 3.9b (Figure 3.9c) depicts a R-BiSu configuration of size 1 (2), i.e., the NoC is divided in regions of 1×1 (2×2). Note that, the basic BiSu approach corresponds to a region of size 0. Furthermore, we design a hardware block that computes the register shuffling values, instead of using the dedicated core IP of the routers.



(a) Size 0 (b) Size 1 (c) Size 2

Figure 3.9: Illustration of the R-BiSu technique for different region sizes.

3.2.4 System model

We consider a NoC $R \times R$, with the routers, interconnections and Network Interface (NI). Each router is connected to an Intellectual Property (IP), e.g., cores, memories, and hardware accelerators. The routers are connected together through the interconnections. When an IP sends a message in the NoC, the associated NI formats the message. Messages are split into smaller fixed-length Flow controL uniTS (FLITS), where each flit has the same size as the interconnections of the NoC. The routers transmit the flit through the interconnection to a neighboring router. When the message reaches the destination, the router forwards the message towards the NI of the destination, which decodes the message to send it to the associated destination IP.

3.2.5 Fault model

We consider multiple permanent faults [170], expressed as stuck-at, short or bridge faults [70]. As the buffers and the crossbar are the biggest components of a router [76], they have higher probability of accumulating faults due to radiation effects, manufacturing defects or other intrinsic failures. Thus, we consider faults located in i) the interconnections between routers, or ii) the buffers and the crossbar within each router. We assume that an Error Mask (EM) with the position of faults,

which has the same size as the data-bus of the NoC, is provided by methods that can diagnose faults in interconnections and routers [272], such as BIST techniques [160, 33]. Each bit in EM gives the state of the datapath bit-line.

3.2.6 Basic Bit-Shuffling method

Shuffling blocks: To achieve BiSu, the NoC routers are extended with Shuffler (S) and De-shuffler (D) blocks. The S block re-organizes the subflits with the objective of minimizing the impact of the faults. The D block brings back the initial order of the sublits. To deal with the targeted faults, BiSu is applied i) between two routers, to mitigate errors on the interconnection bus, and ii) between the input and output ports, to mitigate errors inside the router. One extra D block is required for the routing controller (CTRL), which reads the routing information of the shuffled header and forwards the current packet towards the expected output, as depicted in Figure 3.8-a. The hardware architecture of the S and D blocks is composed of N_{SF} multiplexers of N_{SF} inputs of S_{SF} bits to one output of S_{SF} bits, and registers, whose values give the multiplexer selections, and thus, the bit-shuffling and de-shuffling. The register values are computed using a bubble sort algorithm [15] that takes as input the EM. It divides EM in equivalent submasks (based on the S_{SF} value), and orders them, along with the corresponding part of the register, in an decreasing order to minimize the impact of faults. The algorithm is executed on the IP core of the router.

Critical packets: Faults cannot be tolerated in the header flits, since they contain control information. When the header contains several unused bits, the BiSu technique uses them to mitigate faults. When the header contains few unused bits, the BiSu technique distributes the header information into two flits. In this way, the number of unused bits is artificially increased (by duplicating the header flit) with a small impact on the NoC latency, i.e., adding a single flit in a packet. The same technique is used for sensible data, e.g., instructions. Notice that, today's NoC are typically based on large buses. Hence, the header duplication is a solution that is applied only when BiSu technique cannot provide full protection using the unused header bits.

Different data and flit sizes: The BiSu technique takes into account differences between both data and flit sizes, when organizing the flits inside the NI. To achieve this, a merger block and a de-merger block are added to the NI, and more precisely, to the packetization and de-packetization blocks always included in classic NoC. These blocks sort the data in the flits at the subflit scale, to reduce as much as possible the fault impact on the data.

3.2.7 Region-based Bit-Shuffling method

The R-BiSu method relaxes the mitigation efficiency of the BiSu technique, since it splits the NoC into regions and the BiSu method is applied in the region frontiers. To compute the S and D registers, we need to consider all faults within a region, which can be accumulated when a packet crosses the region. This information is given by the Region Error Mask (REM), computed based on an hierarchical method, where the REM of a region of size X is obtained by an OR operation among the REM of the related regions of size $X - 1$. Figure 3.9 shows an example of 4×4 NoC with 8-bit flit size and 2-bit subflit size. The NoC is affected by three faults on i) the bit number 4 of the router R_0, ii) the bit number 2 of the router R_0 local interconnection, and iii) the bit number 7 of the router R_5 north interconnection. The associated error masks indicating faults (highlighted by red color) for the different region sizes are displayed in Figure 3.9. A hardware block is designed to compute the values of the S and D registers, reducing the pressure on the core IP of the routers.

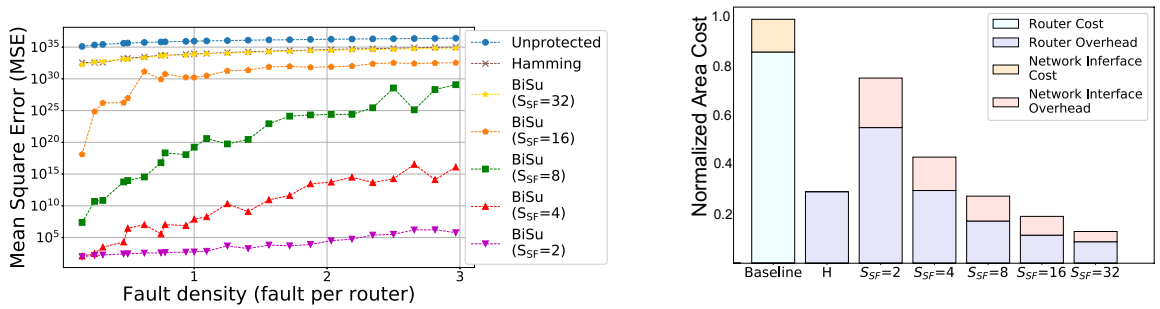
3.2.8 Evaluation

The BiSu and R-BiSu approaches are implemented considering regular square regions and a header distribution on two flits, which correspond to the worst case. This section presents a subset of our evaluation results at the NoC level, while the complete multi-level efficiency evaluation of the proposed techniques through several experiments made on payloads and headers can be found in [C22, J23, C24]. We initially compare the reliability efficiency and the area of basic BiSu, an extended Hamming code and the unprotected NoC, and then, we present the improvements of R-BiSu compared to the basic BiSu. For the reliability efficiency, we use the MSE metric over $10,000$ fault injection sets.

Experimental Setup

Platform & implementation: We consider a 8×8 NoC using the using the 5-ports CONNECT router [186], with a round-robin arbitration and a XY routing algorithm. For the hardware results, the hardware blocks are synthesized on 28 nm FDSOI technology through HLS tools of Mentor Graphics, targeting a clock frequency of 1 GHz. All simulations are performed on the Fedora 28 linux distribution with 8-cores Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz.
Benchmark: Packets of 16 flits are injected according to the TORNADO injection model, where each IP sends a packet at any other IP, considering 64 bit flit size.
Fault injection: The faults are randomly injected in the NoC datapath and modeled using the stuck-at fault model [70]. We consider that the injected faults have always an impact on the data by applying a bit-flip on the affected bits. In this way, the masking effect due to data values is avoided and the fault impact is always visible.

Basic Bit-Shuffling method



(a) MSE.

(b) Area overhead.

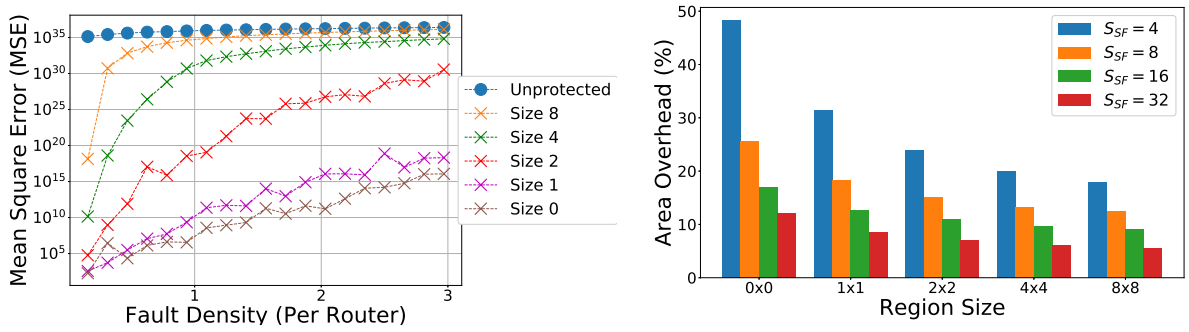
Figure 3.10: Reliability efficiency and area overhead of BiSu technique and the Hamming code.

Reliability efficiency: Figure 3.10a displays the MSE according to the fault density in terms of average number of faults per router, i.e., number of faults divided by the number of NoC routers. Thus, a density of 1.00 fault per router does not mean that each router is impacted by one fault, but that the fault average per router in the NoC is equal to 1.00. BiSu significantly reduces the MSE compared to the unprotected case, even under a high fault density. The BiSu efficiency increases when the subflit size is reduced. The results of extended Hamming code are approximately equal to the BiSu technique with a subflit size equal to half the size of the flit, i.e., the largest possible

size. The BiSu method is more efficient than the extended Hamming code with smaller subflit sizes and higher fault densities.

Hardware overhead: Figure 3.10b shows the global overheads for the BiSu technique and the extended Hamming code for different subflit sizes (S_{SF}). Comparing with the Hamming overheads, BiSu technique can have higher, equal or lower hardware overheads according to the considered subflit size. For example, considering 64-bit flits with 4-bit subflits, the area overhead of the BiSu is 43.5% against 29.3% for Hamming. However, when the subflit size is increased to 8 bits, the overhead is reduced to 27.5%. Note that, the NoC performance are not impacted by the combinatorial D and S blocks, contrary to the Hamming implementation. For example, considering 64-bit flits with 4-bit subflits, the D and S critical path is 0.55 ns against 0.32, 0.85 and 0.76 required for Hamming encoder, checker and decoder, respectively.

R-BiSu method



(a) MSE ($S_{SF} = 4$) (b) Area overhead

Figure 3.11: Reliability efficiency and area overhead comparison.

Reliability efficiency: Figure 3.11a depicts the MSE fault density, i.e., the number of faults per router, for different region sizes considering 64-bit flits divided into 4-bit subflits. The MSE increases with the region size, until it reaches the same results with the unprotected NoC, when the density of faults is high. However, we observe that the size of the region can be increased from 0 to 1 with only a small impact on the MSE. For example, the MSE is increased from 2.17×10^8 to 3.99×10^9, when the region size changes from 0 to 1 for a fault density equal to one.

Hardware overhead: Figure 3.11 presents the hardware costs of the R-BiSu method in terms of area and power overhead for different region sizes and subflit sizes. We observe that the overhead is decreased with the subflit size and the region size increase. When the region size increases from 0 to 1, the area overhead is reduced from 48.3% to 31.4%. The dedicated hardware block to compute the value of the shuffling and de-shuffling registers has a small impact on the global hardware cost of the method. For example, the area and power overheads are increased by 5.3% and 5.9%, respectively, for a region of size 1, and by 1.3% and 1.4%, for a region of size 2. The latency to compute the register values depends on the number of subflits present inside a flit. For example, if the flit is composed of 16 subflits, then the latency to update the registers is equal to 370 ns. When the flit is composed of 8 and 4 subflits, the latency is 120 ns and 44 ns, respectively.

Pareto front

The hardware costs reduce with the region size increasing, but the reliability of the method is decreased. To exploit this trade-off, Figure 3.12 plots the Pareto front for different subflit sizes and region sizes considering 64-bit flits and a fault density equal to one fault per router. Overall, cases exist where the basic BiSu does not belong to the Pareto front. The region size can be increased from size 0 to size 2 with a small impact on the efficiency, while the area overhead is reduced from 48% to 33%. Furthermore, an increase of the subflit size, reducing hardware overheads, has a higher impact on the efficiency. Last, we conclude that increasing the region size is a better way to reduce the hardware overheads than increasing the subflit size.

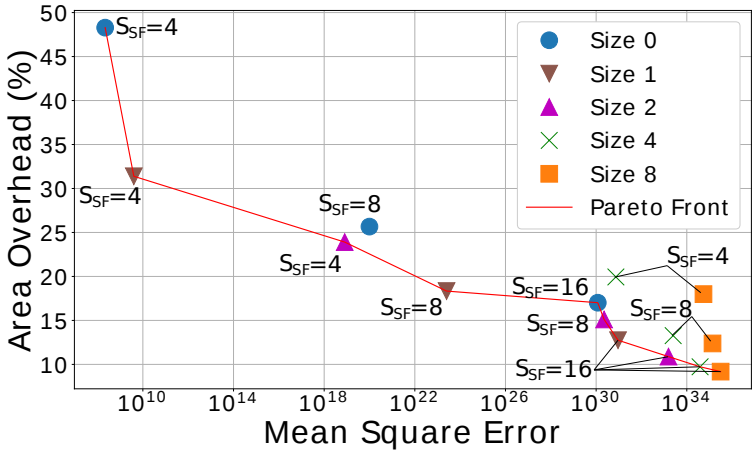


Figure 3.12: Area and MSE Pareto front.

3.3 Cross-layer reliability analysis for complex hardware designs

3.3.1 Context

Due to the increased level of chip integration, the reduced transistor sizes and the lower supply voltages of modern technologies [118, 80], the hardware designs are becoming more and more sensitive to environmental sources [206], such as radiation. Radiation-induced faults can be caused by particles from the atmosphere and their impact on the reliability must be carefully assessed [27]. Ionizing particles can create charges in semiconductor and their density depends on the ion species and their energy, characterized by the Linear Energy Transfer (LET) [117]. The transferred energy from particles can (i) corrupt the state of sequential logic, by flipping bits stored in a memory cell or a flip-flop, and (ii) impact the combinational logic, by creating current-voltage transients, known as Single-Event-Transient (SET). The SET is propagated in the forward cone of the impacted combinational cell and it can be eventually latched by sequential logic [239]. As a result, one or several bits are modified leading to Single-Event-Upset (SEU) or Multiple-Event-Upset (MEU). Such errors can jeopardize the system execution, since their appearance in hardware can lead to failures in the application.

In this context, evaluating the reliability of a system under radiation-induced faults is an essential part of the system design process.

3.3.2 State-of-the-Art

Reliability analysis can be performed by injecting faults into the system and observing its behavior. Fault injection can be done by hitting the real system with a radiation beam and by simulating the impact of injected faults. The first method requires dedicated and costly material, whereas the system can be destroyed during experiments. The second method does not have these limitations and allows a more detailed system analysis. Two main trends exist in reliability analysis through simulation, i.e., the first category characterises the induced faults by analysing the radiation impact at the lower hardware design layers, i.e., Technology and Circuit (T&C) layers, while the second category is applied at higher hardware and application layers to characterize the system execution under transient faults (or soft errors). Table 3.4 summarizes representative vulnerability approaches, and compare them with regard to the layers and fault models they consider.

Table 3.4: Comparison with representative reliability analysis approaches.

Ref.	Layer					Fault model		
	T&C	Gate	RTL	Microarch.	Application	SEU	MEU	SET
[117, 208]	✓					✓	✓	✓
[168]					✓	✓	✓	
[265, 123]				✓	✓	✓		
[269, 93]				✓	✓	✓	✓	
[47]		✓			✓	✓	✓	✓
[236]		✓	✓		✓	✓	✓	✓
[C26, C20]	✓	✓		✓	✓	✓	✓	✓

Radiation analysis at technology and circuit layers can take into account the circuit layout, the fabrication technology, the radiation and operational environments. For instance, soft errors due to neutron strikes are characterized through a SPICE simulator [208] and Monte-Carlo simulations are used to compute neutron, proton, heavy ion and α emitter contamination [117]. Although these approaches accurately characterize the impact of radiation on the physical circuit, they remain at low hardware layers and do not analyse the propagation of such faults to the system execution.

For reliability analysis at higher layers, a trade-off exists between the accuracy and the time for analysing complex hardware designs. Fault injection at the application level is fast, but less accurate. It is agnostic of the hardware state, fault injection occurs only between instructions and in application variables [168]. To improve accuracy, the underlying hardware should be taken into account. Fault injection approaches at the hardware level typically use random fault models following uniform distributions [93]. The majority of existing approaches focus on single-bit faults in sequential logic, e.g., the fault model is based on a random single bit-flip in the microarchitectural state [265] and a bit-flip on the value of one storage element [123]. Some approaches consider multiple-bit flips in sequential logic, e.g., multiple architectural vulnerability analysis is computed for faults affecting a number of contiguous bits in SRAMs [269, 93]. Last, approaches consider faults occurring both in the sequential and the combinational logic. For instance, random single and multiple faults, occurring to instructions that use arithmetic units, are analysed in [47]. An instruction-set simulator executes the application and invokes a gate-level simulator to inject and propagate the fault. A hybrid fault injection framework combines Register Transfer Level (RTL) and gate-level simulation, injecting an SET of one clock cycle following a uniform probability [236]. Existing approaches characterise the impact of transient faults on the system execution, typically considering random faults and uniform distributions.

Although these two categories are complementary, few works consider their combination in order to accurately analyse the impact of the radiation from the environment on the hardware design and application workload under study. A recent cross-layer approach considers technology, circuit, hardware and application layers based on Bayesian models for single-bit faults in memory components [262]. However, multiple-bit faults and combinational logic faults cannot be neglected and should be included in the reliability analysis.

3.3.3 Contributions

We address this limitation by providing a novel cross-layer reliability analysis from the semiconductor layer up to the application layer [C26, C20] in order to quantify the risks of faults under a given context, taking into account the characteristics of the environmental radiation, the physical hardware design and the application. The overview of the cross-layer reliability analysis flow is depicted in Figure 3.13. The first step to obtain an accurate analysis is to use fault models that reflect the reality of the system's environment, e.g., actual occurring faults on a given hardware design during a flight from Paris to Los Angeles due to single energy particles. To achieve that, we have used models for the radiation-induced faults that take into account both the specific environmental conditions and the characteristics of the specific hardware. Such models are obtained by a technology and circuit layer simulation tool that characterises the impact of ionizing particles, under different scenarios, on the hardware design under study. The tool's output is a set of databases that model the distribution of the transient faults at the circuit layer, depending on the cell type, size, inputs and radiation scenario. Such technology- and circuit-layer analysis is required once per fabrication technology and radiation scenario. We apply a gate-level analysis, based on statistical fault injection through a single-cycle simulator, in order to characterise with significant statistical confidence the propagation of radiation-induced faults, modelled at the circuit level. This simulation is able to analyse logical masking and latching window masking, since the hardware design frequency, the area, the delay, the type of cells and the netlist of the hardware design are taken into account. The output is a set of databases describing single-bit and multi-bit error patterns that avoided masking and finally latched in the hardware design registers. Such gate-level analysis is applied once per hardware design. Last, we analyse the impact of single-bit and multi-bit error patterns, occurring at microarchitecture layer, on the system execution, taking into account the application workload. This is achieved through fault injection using a fast Cycle-Accurate-Bit-Accurate (CABA) simulator executing the application. The microarchitecture-level analysis is applied per application workload.

3.3.4 Fault models through Technology and Circuit Analysis

The cross-layer reliability analysis uses as input the results obtained with MUSCA-SEP3 [117] and ATMORAD [44] tools from ONERA, which analyse the physical impact of radiation to the hardware circuit, therefore considering the environment and the physical circuit of the hardware design. As an output, a set of databases of fault models per technology cell is created, along with their probability to occur based on the cell type, size and inputs.

The environmental models are based on generating environment databases according to the considered missions (e.g., aircraft trajectories, ground trajectories, orbits) and space weather conditions (e.g., solar activity, solar flare). The physical models correspond to the device description, including the circuit layout, fabrication technology, semiconductor active zones, passivation, metallization layers and package. The technology and circuit level analysis is based on Monte-Carlo

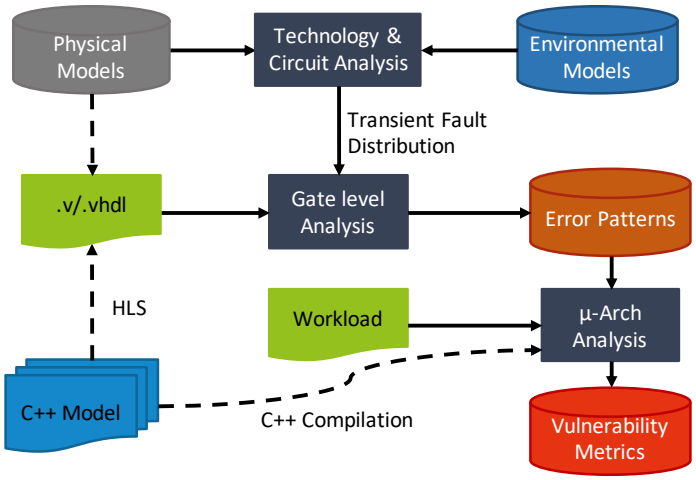


Figure 3.13: FLODAM cross-layer reliability analysis flow.

simulation using a sequential model of the various physical mechanisms occurring when a particle hits a circuit, and potentially leading to a radiation-induced fault. The model describes (i) the particle transport in materials (modifications of the particle primary characteristics via the interaction with the structure, shielding, package and over layers), (ii) the electron-hole pairs and charge generation in semiconductor, (iii) the transport and collection of the charges when electron/holes reach the electrodes, (iv) the transient pulses induced on the different electrodes (drains, sources and taps), and (v) the final effects at the circuit layer. When the circuit is impacted by a particle i of energy E_i, the first step is to identify the cells and transistors potentially impacted. Depending on i, E_i, the size of the transistor and the positioning of the particle with respect to the Drain-Grid-Source topology, the most relevant induced current is identified. After this analysis, the tools select a current from an $I(t)$ database, depending on the characteristics of the transistor to be impacted (size, type), the particle and its characteristics, to be injected at the circuit layer. Following the aforementioned approach, a set of databases is generated with fault models per cell from the technology library used for the considered design, and for the different particles encountered in atmospheric and space environments.

3.3.5 Error patterns through Gate-Level Analysis

As exhaustive fault injection is not possible for complex hardware designs, the aim of the gate-level analysis is to create statistically representative models at this layer for radiation-induced faults occurring at both combinational and sequential cells of the hardware design. To obtain such fault models with statistical confidence and within reasonable time, the gate-level analysis is performed through statistical fault injection per pipeline stage. This is achieved through single-cycle gate-level simulation using the fault models obtained by the technology and circuit analysis. The number of faults N to be injected is defined from the required confidence level in the statistical analysis, i.e., $N = \frac{t^2 \times p \times (1-p)}{e^2}$, where t is the critical value related to the statistical confidence interval, e the error margin, and p the percentage of the fault population individuals assumed to lead to errors [260].

In order to be able to inject faults, the netlist of the hardware design is extended with an injection block inserted at the output of each cell. With these injection blocks, we can insert SEUs, MEUs and SETs anywhere, at any time and with any duration in the netlist. Note that, in order

not to affect the timing characteristics of the hardware design, the propagation delays of all the injection blocks are set to zero. The inputs of the hardware design are randomly generated, e.g., for the execution stage of a processor the inputs are instructions randomly generated from the Instruction Set Architecture (ISA) using random operands. For each such input, a fault-free cycle is executed to obtain the golden reference, i.e., the fault-free output. Based on the technology and circuit layer analysis, the higher the area of a technology cell, the higher its probability to be affected by radiation. Therefore, the selection of the cell, where the fault is injected, it driven by the area of the cell. If the selected cell is of sequential type, the fault is injected directly to the pipeline register. If the selected cell is of combinational type, an SET is inserted into the netlist. The time offset, when the SET is inserted, is randomly chosen within a clock cycle, since radiation may affect the hardware at whatever time instance. The SET duration is given from the databases obtained during the technology and circuit layer analysis. Then, the SET is injected and the output is latched by register at the output of the design stage. If the result in the register is different than the golden reference, the injected fault has led to a single-bit or multiple-bit error. The number and the position of faulty bits are logged, in order to create a set of databases with error patterns.

3.3.6 Vulnerability metrics through Microarchitecture-Level Analysis

The microarchitecture-level analysis is based on fault injection that is able to evaluate the masking due to the microarchitecture and the application workload. The fault injection is driven by the error patterns, modelled during the gate-level analysis and describe the single-bit and multiple-bit faults to be injected at the hardware design registers. Prior to any fault injection, we execute the application under study, without faults, in order to obtain a set of golden references: (i) the application output, (ii) the system state (memory and registers), and (iii) the number of cycles required for the execution of the application. Then, the fault injection tool executes the application and injects faults to the hardware design registers, while the application runs. The cycle to inject the faults is chosen randomly between the first cycle and the total number of cycles needed for the fault-free execution, since radiation may impact the system any time. The location, where the faults are injected, is driven by the size of the combinational and sequential logic of the overall hardware design. The larger the area, the higher its probability to be selected. The characteristics of the fault (i.e., how many and which register bits are affected), to be injected in the register of the selected hardware pipeline stage, are provided by the gate-level error patterns. The more times a specific error has appeared, the higher is its probability to be injected, during the microarchitecture analysis. After the fault injection and upon application termination, the results are compared to the golden references, categorising the impact of faults as:

- *Hang (H):* The execution time has exceeded a threshold, and thus, it is assumed that it has entered an infinite loop.
- *Crash (C):* The execution of the application has terminated unexpectedly and an exception has been thrown (e.g., out of bound memory access, misaligned PC, hardware trap, etc.)
- *Application Output Mismatch (AOM):* The application output is different than the golden reference.
- *Internal State Mismatch (ISM):* The system state (memory and registers) are different than the golden reference.
- *Functionally Masked (FM):* The application has finished execution, with no AOM and no ISM.

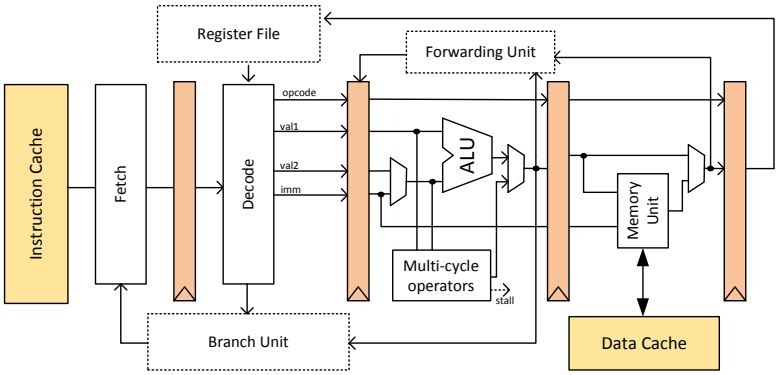


Figure 3.14: RISC-V core with 5-stage pipeline [213].

3.3.7 Evaluation

An open-source 32-bit RISC-V processor [213] is used as a case study, consisting of a standard 5-stage pipeline, including a forwarding mechanism, a hardware multiplier in its execution stage, a register file with 32 registers in the write-back stage, and an instruction and data level one cache memory. The considered processor is fully specified using C++ and has been synthesized to the gate-level using Mentor Graphics Catapult High-Level Synthesis and Synopsys Design Compiler with a target frequency of 500 MHz. The target fabrication technology is 28 nm FDSOI from ST-Microelectronics using a supply voltage of 1.0 V. The sequential logic of the processor corresponds to 45.85% of the total area and the combinational logic to 54.15%. Table 3.5 depicts the relative area of the pipeline stages.

Table 3.5: Relative area of RISC-V pipeline stages.

Pipeline stage	**Fetch**	**Decode**	**Execute**	**Memory**	**WriteBack**
Area	6.01%	11.02%	35.47%	5.10%	42.41%

Technology and Circuit Layer Results

Thanks to our collaboration with ONERA, Toulouse, under FLODAM project, we obtain fault models for 28nm FDSOI technology cells considering different radiation scenarios, considering several flights departing from Paris to New York, Los Angeles, Johannesburg and Sao Paulo, and vice versa. For each flight plan, the natural radiation environment can be calculated, which describes the differential energy spectrum of neutron, proton and muon for each point of the flight plan trajectory. Analysis takes place considering different particle types, particle energies and technology cells. The transistors potentially impacted when a particle i of energy E_i hits a technology cell and the most relevant induced currents are identified. A current database is created depending on the transistor size and type, the particle and its characteristics. Then, current injection at the circuit level takes place leading to a set databases with fault models characterised with distributions of SET widths in the cell. For instance, Figure 3.15 summarizes the distribution of the width of the SET pulses created when neutrons hit the 28nm FDSOI cells, normalized to the cell area and input sizes. In this radiation scenario, we considered an LET equal to $58 MeV.cm^2.\mu m^{-1}$, a temperature $25°C$ and an incidence angle of 90°. Overall, 212,000 injections took place on 91 logic gates of the

fabrication technology. We observe that the majority of SET widths are below 500 ps (89.457%), while a minority is larger than 1 ns (0.651%) and than 1.5 ns (0.13%) for our case study.

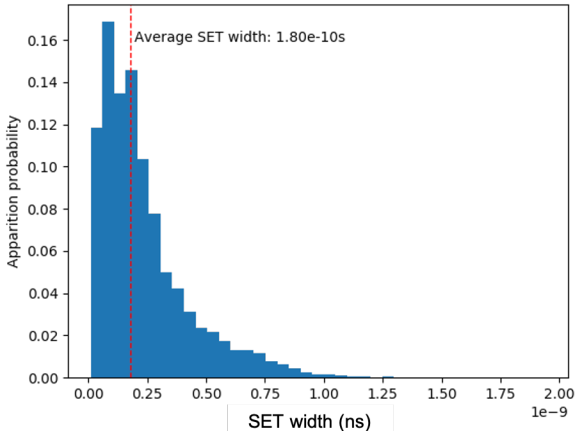


Figure 3.15: SET distribution normalized to cell area and input sizes

Gate-Level Results

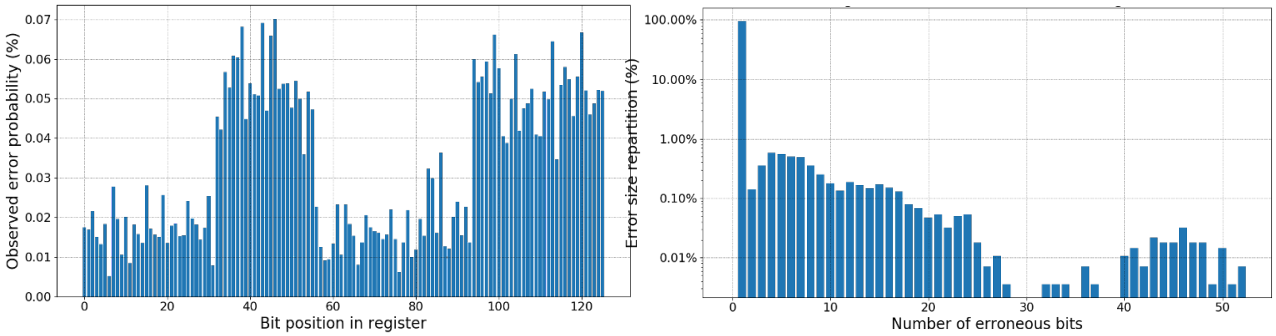
The gate-level analysis provides the error patterns, created due to injected faults at the circuit layer, that managed to avoid logical and timing window masking and latched at the output register. Considering a 99.8% confidence interval with 5% error margin for each input set values, we injected 10^3 for each of the 10^3 different inputs per pipeline stage (10^6 total fault injections per stage). The time required to perform gate-level analysis for the five RISC-V pipeline stages is $1,039$ s, as depicted in Table 3.6 with Questa Advanced Simulator 10.7b running on a second-generation Intel Xeon CPU. Figure 3.16 depicts the gate-level analysis results for the RISC-V execution stage, considering an SET pulse width equal to 400 ps and an operating frequency of 500 MHz. Figure 3.16a shows the probability of each bit of the output register of the execution stage to be erroneous. Some bits in the register have higher error probabilities, such as the bits corresponding to the ALU output (bit 32 up to bit 63) and data resulting from logical operations on values from Control and Status Registers (CSRs) (bit 94 up to bit 125). Figure 3.16b shows how many bits were faulty due to a single SET. The higher number of observed erroneous bits for the execution stage is 52 bits, which is 41.3% of the pipeline register size. Experiments for different pulse widths have shown similar results. Regarding SET propagation, for pulse widths 100 ps and below, the SETs are not often propagated within the latching window of the register, and with a pulse width below 50 ps, no latching of SETs is observed.

Table 3.6: Time of gate-level analysis.

Fetch	**Decode**	**Execute**	**Memory**	**WriteBack**	**Total**
37 sec	392 sec	496 sec	82 sec	32 sec	1,039 sec

Microarchitecture-Level Results

Although the error patterns managed to survive at the microarchitecture-level, they may not necessarily lead to a visible error during the system execution. Using a 99.8% confidence interval



(a) Error probability for each register bit

(b) Size of error patterns.

Figure 3.16: Gate-level results of the RISC-V execution stage.

and a 1% error margin, we injected 23,874 faults, using the error patterns obtained from the gate-level analysis. The microarchitecture layer output is the distribution of vulnerability metrics. Figure 3.17 compares the vulnerability results for four benchmarks (matrix multiplication, qsort, strsearch, blowfish), obtained by the proposed flow and by typical approaches which inject SEUs in the microarchitecture with a uniform fault occurring probability for all registers. Overall, with the proposed flow (FLODAM), we observe that less faults are masked, compared to standard microarchitecture-level analysis.

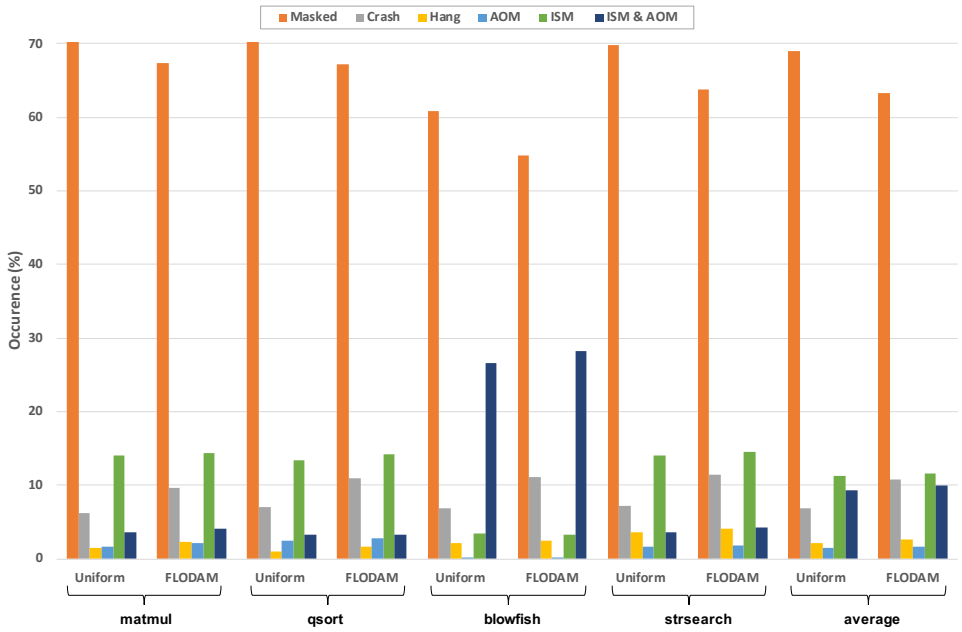


Figure 3.17: Vulnerability metrics.

The CABA simulator performance is 18.2 million instructions per second on average, using an 8^{th} generation Intel i7 CPU running at 2.40 GHz. The time required to perform the vulnerability analysis at the microarchitecture-level based on the CABA simulator is shown in Table 3.7. Note that, performing similar analysis using full gate-level methods would require a prohibited time.

Table 3.7: Time of microarchitecture-level analysis.

Bench.	**matmux**	**qsort**	**blowfish**	**strsearch**	**average**
Time	388 min 40 sec	327 min 20 sec	466 min	427 min 20 sec	406 min 32 sec

3.4 Conclusions

Due to the increasing intrinsic failure rate in the electronic field, electronic systems became more sensitive to faults affecting their correct functionality. Transient faults are challenging because of their nature, and processors have to the extended with fault tolerant capabilities. We extend heterogeneous VLIW processors with hardware mechanisms to support reliable benchmark execution. The proposed approach explores the idle slots in the current and next bundles and prioritizes dependent instructions. More precisely, the proposed mechanism perform instruction triplication considering short-term transient faults [C12, J26]. We leveraged this mechanism for permanent faults, where instruction triplication and re-scheduling is applied taking into account the status of the FUs [C11]. To reduce the performance degradation due to the instruction level fault tolerance, dynamic instruction re-scheduling is applied based on the status of the faulty FUs in a coarse-grained way [C15] and a fine-grained way [C17].

Permanent faults are critical since there is no recovery, contrary to transient faults. Existing approaches to deal with multiple permanent faults over NoC often induce high hardware costs in terms of area and power consumption, while their reliability efficiency is drastically impacted. We proposed BiSu technique [C24, C22] for fault mitigation in NoC datapath, which re-organizes the flits at the subflit scale to move the fault impact on the LSBs. A redundancy approach is presented to handle critical data, such as headers, by distributing critical information on two flits allowing to artificially increase the number of unused bits to enable the BiSu method. Hardware blocks are inserted in the NIs to manage the difference between the flit size and the data size. The obtained results show that BiSu can manage high fault densities contrary to state-of-the-art methods, such as the extended Hamming code, with small critical path and latency overheads. To reduce hardware overheads, a region-based version (R-BiSu) [J23] is proposed, which protects regions, instead of routers. An hierarchical method computes the error mask of a region. R-BiSu trade-offs reliability efficiency and hardware costs through different subflit sizes and region sizes. Passing from BiSu to R-BiSu with a region size of 2 reduces the hardware costs with small impact on the reliability efficiency. A hardware block computes the shuffling values with negligible hardware overheads.

To analyse the reliability of complex hardware designs, we proposed a cross-layer reliability analysis [C26, C20], from the semiconductor layer up to the application layer. We focused on transient faults caused by single radiation particles. Particles with different characteristics have different impacts when hitting sequential and combinational cells of the design, providing different fault models. From the analysis results, particles impacting combinational logic can lead to SET of different width pulses. Depending on the characteristics of the particles and the hardware design, SET can be latched in the registers and alter a significant number of bits, leading to a significant number of MEUs. Such MEUs can be significantly large in size and they not disturb only adjacent bits. Performing analysis considering the impact of radiation to combinational logic leads to less masked faults at the application layer, compared to typical methods using randomly injected faults based on uniform distribution. Combining statistical analysis with single-cycle gate-level fault injection and microarchitecture-level fault injection, the reliability analysis is significantly reduced compared to full gate-level fault injection.

Chapter 4

WCET- and fault-aware task deployment for multicore architectures

This chapter summarises our contributions on desing-time task deployment on multicore architectures with both hard timing guarantees and reliability requirements. More precisely, section 4.1 presents our contributions regarding optimal and heuristic design-time approaches, characterized with pessimistic WCET regarding interferences, over shared memory multicore architectures, under hard real-time and reliability constraints for different DVFS schemes. These works have been performed during the PhD period of Minyu Cui. Section 4.2 presents a task deployment approach, under real-time and energy constraints, considering a multicore architecture with NoC as communication mean, taking into consideration the energy consumption for communication and the routing paths. This work is performed during the 3-year internship of Qi Zhou. Section 4.3 summarises the aforementioned contributions.

4.1 Design-time mapping under different DVFS schemes

4.1.1 Context

Safety-critical applications have not only real-time constraints, but also reliability constraints [175]. However, designing for energy efficiency, real-time and reliable task execution are conflicting objectives. Enhancing real-time execution and reliability often requires more energy consumption. To meet real-time constraints, execution with higher frequencies maybe required, increasing energy consumption. To increase the reliability, high frequencies and task replication is usually applied [275]. However, task replication has a significant impact both on energy and time; more tasks are executed, thus, more energy and time is required. Furthermore, DVFS has a negative impact on task execution and reliability. Lower frequencies lead to longer execution times and increased transient fault rates [95].

In this context, fault tolerant approaches, such as task duplication, and frequency assignment should be taken into account during real-time and reliable task deployment [201].

4.1.2 State-of-the-Art

Table 4.1 summarises representative task mapping approaches with DVFS with goal of minimizing energy consumption, under Real-Time (RT), and Reliability (R) constraints. Tasks can be Independent (I) or Dependent (D) and the multicore platform Homogeneous (HO) or Heterogeneous (HE). Based on the problem formulation and solving method, solutions are obtained through Heuristic (H) or Optimal (O) approaches. Fault tolerance is provided by task Recovery (Rec) or task Replicas (Rep).

To tackle the negative impacts of DVFS on reliability, existing techniques preserve the original system reliability for all tasks, i.e., the reliability that can be obtained with the maximum platform frequency. Then, slack is reserved to execute a recovery task with the maximum frequency. In the individual-recovery scheme, a recovery task is scheduled for every selected task, while in the shared-recovery scheme, multiple tasks running on the same processor share a single recovery task, for independent [197] and dependent tasks [96] on homogeneous platforms. Then, DVFS is used to scale down tasks. If an error is detected, the recovery task is evoked. Shared-recovery scheme has also been considered for dependent tasks on heterogeneous systems [283]. However, such approaches usually require high frequencies to satisfy the original reliability, increasing energy consumption. It is also possible that reliability constraints cannot be satisfied, even with the maximum frequency, leading to empty solution space.

Other approaches compute the required number of replicas to always meet reliability constraints. Due to high complexity, heuristics are typically proposed, e.g., a first-fit decreasing heuristic [103] and a layered worst-fit decreasing heuristic [101] for independent tasks on homogeneous platforms, and an iterative heuristic for independent tasks [90] and a list scheduling heuristic for dependent tasks [275] on heterogeneous platforms. Hybrid approaches use a given number of replicas and apply task recovery, when an error occurs [225]. However, the increased number of replicas leads to large energy consumption, combined with a negative impact on execution time. When the real-time constraints are strict, solutions may not exist. To reduce this negative impact, the number of replicas is restricted. In [226], a heuristic decides among single execution, task duplication and task triplication. However, no guarantees are provided on real-time and reliability constraints. In [95], a heuristic determines which tasks to be duplicated, removing the need of a recovery task, without

Table 4.1: Representative DVFS task deployment approaches targeting energy minimization.

Ref.	Task		Platform		Fault tolerance		Constraints		Solution	
	I	D	HO	HE	Rec	Rep	RT	R	H	O
[197]	✓		✓		✓		✓	✓	✓	
[96]		✓	✓		✓		✓	✓	✓	
[283]		✓		✓	✓		✓	✓	✓	
[103, 101]	✓		✓			✓	✓	✓	✓	
[90]	✓			✓		✓	✓	✓	✓	
[257]	✓			✓		✓	✓	✓	✓	✓
[225, 226]		✓	✓			✓	✓	✓	✓	
[95]		✓	✓			✓	✓		✓	
[275]		✓		✓		✓		✓	✓	
[C19, J24]	✓		✓			✓	✓	✓		✓
[C23]		✓	✓			✓	✓	✓		✓
[J25]		✓	✓			✓	✓	✓	✓	

considering reliability constraints. If an error strikes a task's execution, this task is re-executed at maximum frequency. An Integer Linear Programming (ILP) approach [257] maps independent tasks on a heterogeneous platform to satisfy a given percentage of duplicated tasks, under energy budget constraint. However, the reliability of tasks is not considered.

4.1.3 Contributions

We extend the State-of-the-Art by proposing a task mapping approach that decides between reliable single task execution and task duplication, under real-time and reliability constraints, considering DVFS schemes with different flexibility regarding frequency assignment. Our approach applies partial task duplication, which duplicates a task when its reliability constraint cannot be satisfied even with high frequencies or when the energy consumption of original and duplicated tasks is less than executing only the original task with a high frequency. We leverage our approach for systems that support DVFS at task, processor and system level. First, we design methodologies for task mapping on multicore platforms that provide optimal solutions for independent [C19, J24] and dependent [C23] tasks under different DVFS schemes. Then, to cope with high computation time required to obtain optimal solutions, we propose a set of heuristics that provide near-optimal solutions with reduced computation time, leading to scalable approaches [J25]. In the next paragraphs, we summarize the task and platform models considered in our contributions and provide a general description of the proposed problem formulations and solutions. For the exact mathematical formulations for each problem, please refer to [C19, J24, C23, J25].

4.1.4 System model

We consider an interference-pessimistic set of N independent tasks, i.e., $\{\tau_1, \ldots, \tau_N\}$ in [J24, C19] and N application frame-based dependent tasks in [C23, J25]. Dependent tasks are represented by a Directed Acyclic Graph (DAG) $G(V, E)$, where V denotes the set of N tasks and E represents the partial order, corresponding to the precedence constraints among tasks. A task is ready for execution when all its predecessors have been completed. Each task τ_i is described by a tuple $\{W_i, R_i^{th}\}$, where W_i is the Worst Case Execution Cycles (WCEC) and R_i^{th} is its reliability threshold. Each task has its own reliability constraint, since functions of an application exhibit distinct significance and/or vulnerabilities, due to variations in the spatial and temporal vulnerabilities of different instructions [226]. Without loss of generality, the release times of all tasks are considered at the start of the scheduling period, which provides also the global deadline D.

A multicore platform is considered with M homogeneous processors, i.e., $\{\theta_1, \ldots, \theta_M\}$. The following DVFS schemes are exploited: i) Task-Level DVFS (TL-DVFS), where the frequency assignment is performed independently per task, such as in [114, 225, 103, 221] and ARM DynamIQ big.LITTLE platform [199], ii) Processor-Level DVFS (PL-DVFS), where the frequency assignment is performed independently per processor, such as in Intel-Xeon E5620, and iii) System-Level DVFS (SL-DVFS), where the same frequency is assigned at the same time to all processors, such as in [138, 140]. For each core, there are L different Voltage/Frequency (V/F) pairs $\{(v_1, f_1), \ldots, (v_L, f_L)\}$. When task τ_i is assigned with frequency f_k, its execution time is calculated as $et_i = \frac{W_i}{f_k}$. For each processor θ_m, the power consumption is modeled as the sum of static power P_k^s and dynamic power P_k^d, i.e., $P_k = P_k^s + P_k^d$. The dynamic power consumption with V/F level (v_k, f_k) is $P_k^d = C_{eff} f_l v_k^2$, which is a common used model when frequency and voltage scaling have a linear relation.

We focus on soft errors that follow a Poisson Distribution with fault rate $\lambda(f)$ at frequency f [287], where fault rate is the expected number of faults per time unit [81]. For systems supporting

DVFS, the fault rate at frequency f_k follows an exponential distribution $\lambda(f_k) = \lambda_0 \times 10^{d\frac{f_{max}-f_k}{f_{max}-f_{min}}}$, where λ_0 is the average fault rate at maximum frequency, d is a constant, used to measure the sensitivity of fault rate to voltage/frequency scaling. f_{max} and f_{min} are the maximum and minimal frequency in the L voltage/frequency levels, respectively. The reliability of a task execution is the probability of executing the task without any fault, and it varies exponentially as a function of its execution time as $R(f_k) = e^{-\lambda(f_k)\times et}$ [81]. If the reliability of original task τ_i is larger than its reliability constraint, the execution is considered as reliable, i.e., $R_i = R_i^o$. Otherwise, the task τ_i is duplicated and executed on a different processor. Then, the reliability of τ_i is $R_i = 1 - [1 - R_i^o][1 - R_i^d]$, where R_i^d is the reliability of duplicated task.

4.1.5 General problem formulation

Given a set of tasks, our goal is to map them on M processors, such that the overall energy consumption is minimized, under task real-time and reliability constraints. To achieve that, we determine 1) which processor should the tasks be executed on (task-to-processor allocation), 2) which tasks should be duplicated, 3) what frequency should be used for the original and duplicated tasks (frequency-to-task assignment). When we study set of dependent tasks, we define 4) when should the task start (task scheduling). Overall, a processor is able to execute one task at a time instance (task non-overlapping constraint), the tasks should finish before their deadline (real-time constraint) and meet their reliability threshold (reliability constraint). Table 4.2 summarizes the constraints and the variables used in each contribution, along with the type of the proposed problem formulation and the proposed solutions.

Table 4.2: Summary of problem formulations.

Constraints	**Binary**	**Continuous**	[C19, J24]	[C23]	[J25]
Task-to-processor allocation	✓		✓	✓	✓
Frequency-to-task assignment	✓		✓	✓	✓
Task-duplication decision	✓	✓	✓	✓	✓
Task dependencies	✓	✓			✓
Task non-overlapping	✓	✓	✓	✓	✓
Real-time	✓	✓	✓	✓	✓
Reliability	✓	✓	✓	✓	✓
Type			MINLP	MINLP	MINLP
Solution			O	O	H

4.1.6 Optimal solution

Similar to Chapter 2.1, we use variable replacement to eliminate the non-linear items related to the task non-overlapping constraints to safely transform the MINLP to an equivalent MILP. Then, the MILP is solved using optimization solver tools, such as Gurobi.

4.1.7 Heuristic methods

The proposed heuristic consists of two phases:
Phase A obtains, per task, the set of possible configurations that meet the reliability for each task, a reliability, execution time, energy consumption table is created for each task based on all possible configurations. A pruning step removes the task configurations that do not satisfy the

reliability constraint. The result is the feasible configurations space of the task. The feasible configurations considering only the original task serve as baseline configurations. The next step prunes any feasible configuration with duplicated tasks, if both energy consumption and execution time are larger than any baseline configuration. The result is the possible configurations, which is ranked based on decreasing energy consumption.

Phase B uses the task configurations obtained from phase A and performs the application mapping, subject to the real-time constraint. Phase B consists of three steps:

In Step 1, priorities are given to tasks for task allocation based on the largest average execution time. The average execution time of a task is computed by the average execution time among all possible configurations. The task priority list is ordered in decreasing execution time.

In Step 2, the initial application mapping is generated to check if the problem is feasible and time slack is available. For each task, the initial application mapping uses the first configuration in the task priority list as the selected configuration. For each task, we choose the processor with least total execution time to obtain the task mapping. The set of all task mappings provides the initial application mapping and the total execution time is obtained. If the total execution time of at least one processor exceeds the global deadline D, the studied problem is infeasible, and the algorithm stops. If the total execution time of all processors is equal to the deadline, the initial application mapping is the final mapping and the algorithm stops.

Otherwise, time slack exists in the initial task mapping. Step 3 relaxes the mapping leading to energy savings. Overall, different task configurations and different tasks can be relaxed. The following process decides which task and with which configuration to be selected for relaxation. The initial mapping is set as the current mapping. Then, the following process is applied iteratively, until there is no available time slack for relaxation or all tasks have reached their configuration with the least energy consumption. Before the relaxation, we compute the Energy Saving (ES) and execution Time Increase (TI) for each task and each remaining configuration, compared to the first configuration used in current task and application mapping. First, we search among all tasks and all their possible configurations to select a new configuration for a task that achieves the highest value $\frac{ES}{TI}$. After selecting a task with a new configuration, all task mappings are updated accordingly. Furthermore, the new application mapping and the total execution time for each processor are obtained. Last, all configurations that have a higher energy consumption than the selected configuration for the relaxed task are removed from the task configuration space.

4.1.8 Evaluation

This section presents representative results for the evaluation of the proposed approaches, whereas the complete evaluation can be found in [C19, J24, C23, J25]. We present the feasibility, energy consumption, and computation time for dependent tasks (N=10, M=4) obtained by the proposed optimal approach (O_RAFTM), compared with two SoA approaches: i) the optimal Reliability-Aware Mapping (RAM) (O_RAM) approach, similar to [273] and "ESRG" algorithm in [275], and ii) the Task Duplication Mapping (TDM) (O_TDM), approach, always performing task duplication, similar to [103, 275], when the number of replicas is two, or to [257], with 100% task duplication. Furthermore, we compare the solutions of the proposed heuristic (H_RAFTM) compared to the optimal ones. The approaches are implemented in Matlab and solved with Gurobi. The experiments took place on several servers, provided by IRISA-INRIA infrastructure.

Experimental set-up

Platform: The processor model used for the multicore platform in the experiments considers 64 nm technology and $L = 6$ voltage/frequency levels [200], depicted in Table 4.3.
Benchmarks: To obtain realistic inputs for our experiments regarding the WCEC of the tasks, we count the execution cycles and memory accesses of common benchmarks from MiBench suite [97], using Comet simulator, which is based on a a high-level C++ model with 32-bit RISC-V ISA and standard 5-stage pipeline [213]. The sources of timing variability are eliminated to obtain safe and context-independent measurement [78] without interferences ($WCEC_{iso}$). Then, the $WCEC_{inf}$, considering worst case interferences from the other processors, is computed. As our contribution is not WCET estimation, a trivial pessimistic approach is applied: all processors may conflict during a memory access. From the obtained results, we randomly generate task graphs, where the WCEC of a task is within the range $[1 \times 10^8, 4 \times 10^8]$.

Table 4.3: Summary of experimental set-up for real-time and reliable DVFS task deployment

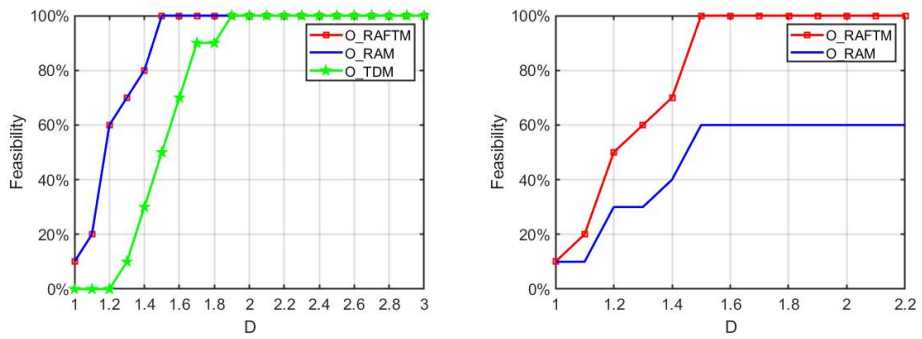
l	f_1	f_2	f_3	f_4	f_5	f_6	W	$[1 \times 10^8, 4 \times 10^8]$
$\boldsymbol{f_l}$ (GHz)	0.801	0.8291	0.8553	0.8797	0.9027	1.0	R^{th}	[0.999, 0.9995], [0.999, 0.9999]
$\boldsymbol{v_l}$ (V)	0.85	0.90	0.95	1.00	1.05	1.1	λ_0	5×10^{-5}
C_{eff}	7.3249	8.6126	10.238	12.315	14.998	18.497	d	3

Constraints: The real-time constraint is tuned from strict deadlines to relaxed ones, based on $D_i = \hat{t}_s^i + k \times d_i$, with a step of 0.1. $\hat{t}_s^i$ is the relative start time of task of task τ_i, i.e., $\hat{t}_s^i = max_{\forall j \in Pre\{i\}}\{D_j\}$, with $Pre\{i\}$ the predecessors of task τ_i i and the relative deadline given by $d_i \in [\frac{C_i}{f_{max}}, \frac{C_i}{f_{min}}]$. The reliability threshold R_i^{th} is selected within the range $[0.9990, 0.9995]$, considering a typical magnitude 10^{-3} for reliability target [103].

Optimal approach

Regarding feasibility, O_RAFTM can find solutions in significantly more experiments then O_TDM, because O_RAFTM is not obliged to duplicate every task. More precisely, when feasibility has not reached 100% for both approaches, O_RAFTM finds a solution, on average, in 33% more experiments than O_TDM (Figure 4.1a). O_TDM cannot find solutions in strict deadlines, only after $D = 1.3$ and O_RAFTM achieves 100% feasibility in earlier deadlines than O_TDM, i.e., $D = 1.5$. Note that, with increasing number of processors, the capability of O_TDM to find solutions improves, as more processors are available to schedule original and duplicated tasks. O_RAFTM has the same feasibility with O_RAM, due to the values of reliability thresholds, which can be achieved by executing only the original task with a high processor frequency. To further explore the behavior of O_RAFTM and O_RAM, we extend the range of the task reliability threshold to [0.999, 0.9999], in order to have few tasks with higher reliability requirements, that cannot be met even with the highest platform frequency. From the obtained results (Figure 4.1b), O_RAFTM is able to still meet these high reliability requirements, by duplicating tasks with a high frequency.

The minimum, average and maximum gains regarding energy consumption for all DVFS schemes are depicted in Figure 4.2. Note that, the minimum gain is 0 between O_TDM and O_RAFTM, which occurs when the deadlines are relaxed and O_RAFTM performs duplication for all tasks, as O_TDM. Compared to O_RAM, we observe a minimum gain of 0 for the less flexible frequency assignments of PL-DVFS and SL-DVFS. In strict deadlines, O_RAFTM and O_RAM have a similar behavior: applying a high frequency to meet the timing constraint, thus no duplication is possible.

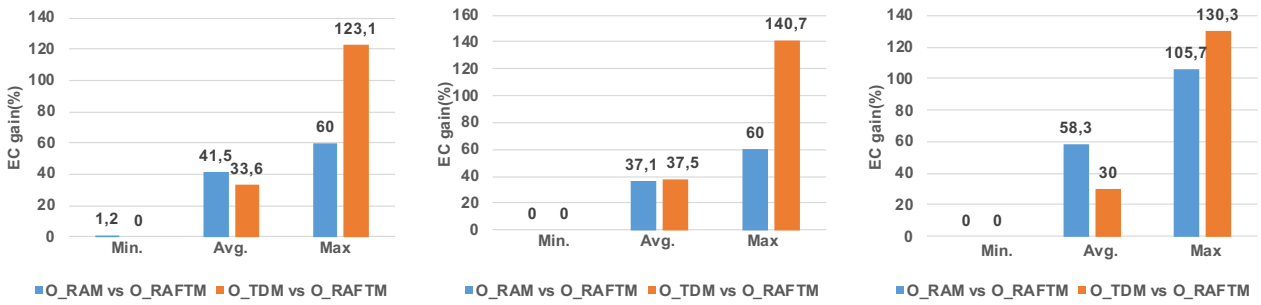


(a) Reliability threshold always met. (b) Reliability threshold not always met.

Figure 4.1: Feasibility for all DVFS schemes for optimal solutions.

We observe a slight decrease in average savings regarding O_RAM in TL-DVFS and PL-DVFS schemes, due to the fact that O_RAFTM behaves as O_RAM in very strict deadlines. Note that, when more processors are available, our proposed approach has more freedom to use the available processors when performing the task mapping, minimizing the total energy consumption. Among different DVFS schemes, we observed that SL-DVFS has a higher impact on the observed gains of O_RAFTM vs O_RAM, compared to the impact it has on the observed gains of O_RAFTM vs O_TDM. When the supported DVFS scheme is flexible, O_RAM performs a more fine-grained assignment, achieving a lower energy consumption. However, when SL-DVFS is supported, O_RAM is obliged to select a high frequency in order to meet the highest reliability threshold of the tasks and executes all tasks with this high frequency, increasing energy consumption. On the other hand, the proposed O_RAFTM can exploit task duplication, applying a lower frequency, and thus, reduces the energy consumption, when time slack is available for relaxed deadline cases.

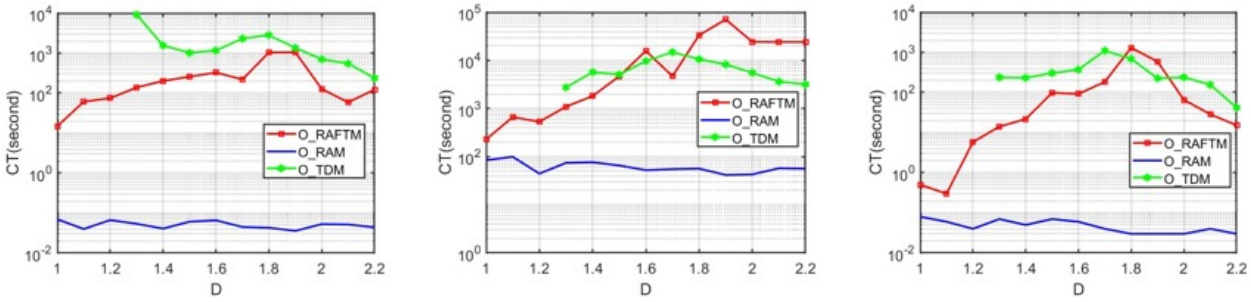
The computation time for the optimal approaches is depicted in Figure 4.3, when the deadline is very strict or very relaxed, O_RAFTM needs less time to obtain the solutions. However, with intermediate deadlines, more time is required as O_RAFTM needs to explore the available time slack to decide which, and how many, tasks to be duplicated, without violating constraints, while consuming the least energy. O_TDM is the most running time expensive approach, because all tasks need to be duplicated, increasing the number of tasks to be scheduled, and thus, the time to find the solutions. O_RAM is the least running time expensive approach. However, as we have seen



(a) TL-DVFS (b) PL-DVFS (c) SL-DVFS

Figure 4.2: Energy consumption gain for all DVFS schemes for optimal solutions.

previously, this approach provides less energy savings compared to O_RAFTM. Note that, with more tasks, more time is required to find a solution, as expected.



(a) TL-DVFS (b) PL-DVFS (c) SL-DVFS

Figure 4.3: Computation time for all DVFS schemes for optimal approaches.

Heuristic method

Regarding feasibility, as shown in Figure 4.4, H_RAFTM and O_RAFTM achieve the same feasibility. Note that, when the number of cores is reduced, e.g. $M = 2$, in few cases when the deadline is strict the optimal approach has 3.3% higher feasibility.

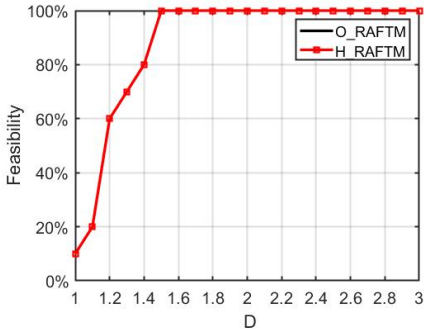
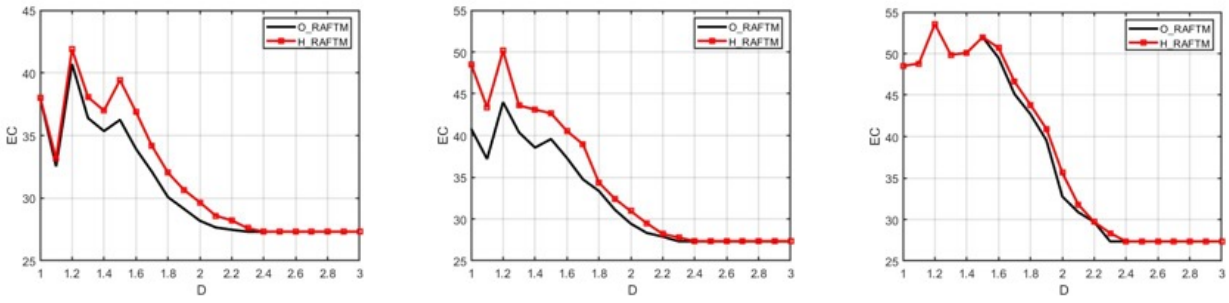


Figure 4.4: Feasibility for all DVFS schemes for optimal and heuristic approaches.

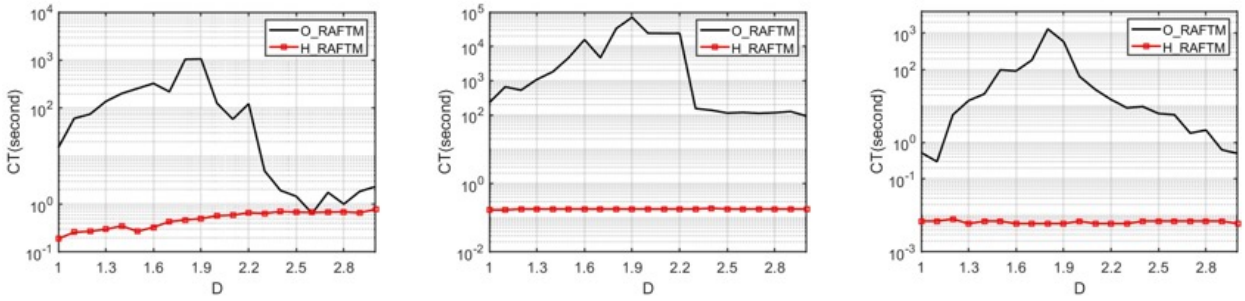
The energy consumption for all DVFS schemes is depicted in Figure 4.5. H_RAFTM generally consumes slightly more energy than O_RAFTM, e.g., on average 2.9% for TL-DVFS, 5.6% for PL-DVFS and 1.3% for SL-DVFS. When deadline is relaxed, H_RAFTM and O_RAFTM obtain solutions with the same energy consumption. With the processor number increasing, the energy consumption of both proposed heuristic and optimal solution flattens at earlier deadlines, since there are more processors available to perform the task mapping, and thus, more opportunities to start the tasks earlier.

The computation time is depicted in Figure 4.6. It can be observed that although few tasks and processors are used, the time to obtain the optimal solution is very long, on average $\times 10^4$ more than the proposed H_RAFTM. Meanwhile, the computation time is largely reduced when the heuristic is used to solve the problem.



(a) TL-DVFS (b) PL-DVFS (c) SL-DVFS

Figure 4.5: Energy consumption gain for all DVFS schemes for optimal and heuristic approaches.



(a) TL-DVFS (b) PL-DVFS (c) SL-DVFS

Figure 4.6: Computation time for all DVFS schemes for optimal and heuristic approaches.

4.2 Design-time mapping considering NoC routing

4.2.1 Context

As shown in the previous section, task deployment plays an important role in the energy consumption, real-time execution and system reliability, especially for cores with DVFS capabilities. Complex architectures with many cores typically use a NoC for inter-processor communication. In this case, the communication cost in terms of time and energy is not negligible compared to the cost of computation [107]. When dependent tasks are allocated on different processors, data must be transmitted. As multiple routing paths for the data transmission can exist in NoC [31], the communication cost depends not only on task mapping, but also on the routing path selection.

In this context, inter-processor communication and routing path selection should be also taken into account the task deployment process in order to balance the overall energy consumption under real-time and reliability constraints.

4.2.2 State-of-the-Art

Table 4.4 categories representative approaches from the literature, performing task allocation (All.), scheduling (Sch.), and duplication (Dup.) on multicores with DVFS, considering multi-path routing (MP), task reliability (Rel.) and communication cost (Com.). The solutions are given by optimal (O) and heuristic (H) algorithms. Several approaches exist to map independent and dependent tasks on the multicore platforms under multiple constraints, such as energy, real-time, and reliability [140,

J16, 103, 275]. These works consider processors typically connected with a high-speed data bus. Thus, the communication cost between any two processors is usually ignored, as it's much smaller compared to task execution cost. However, for the platforms based on NoC, the communication cost becomes important. To reduce the communication cost on NoC-based platforms, the common methods include mapping tasks to processors [237, 107] and adjusting the operating voltage of the routers [100, 2]. However, reliability is not taken into account, especially when DVFS is available. Existing approaches to enhance NoC reliability, include spare processors [31] and task duplication [175, 48]. However, no DVFS is considered in these works.

Table 4.4: Classification of representative task deployment approaches

Ref.	Task			Multicore platform				Solution	
	All.	Sch.	Dup.	DVFS	Rel.	Com.	MP	O	H
[140]	✓	✓		✓					✓
[J16]	✓	✓		✓				✓	✓
[275]	✓	✓	✓	✓	✓				✓
[103]	✓	✓	✓	✓	✓				✓
[107]	✓	✓				✓			✓
[237]	✓	✓				✓			✓
[100]	✓	✓		✓		✓			✓
[2]	✓	✓		✓		✓			✓
[31]	✓	✓			✓	✓	✓		✓
[175]	✓	✓	✓		✓	✓		✓	✓
[48]	✓	✓	✓		✓	✓			✓
[C25]	✓	✓	✓	✓	✓	✓	✓	✓	✓

4.2.3 Contributions

To bridge this gap, a task deployment approach [C25] is proposed to balance the overall system energy consumption, including both core computation and NoC communication, under reliability and real-time constraints. More precisely, the task deployment approach simultaneous optimizes the task allocation and scheduling, frequency assignment, task duplication, and path routing. The task deployment problem is formulated using MINLP. To find the optimal solution, the original problem is equivalently transformed to MILP by adding auxiliary variables and constraints, and solved by state-of-the-art solvers. Furthermore, a decomposition-based heuristic, with low computational complexity, is proposed to deal with scalability issues. It divides the original problem into three subproblems, having a simpler structure with less constraints and variables. Finally, extensive experimental results are performed to demonstrate the advantages of the proposed approaches.

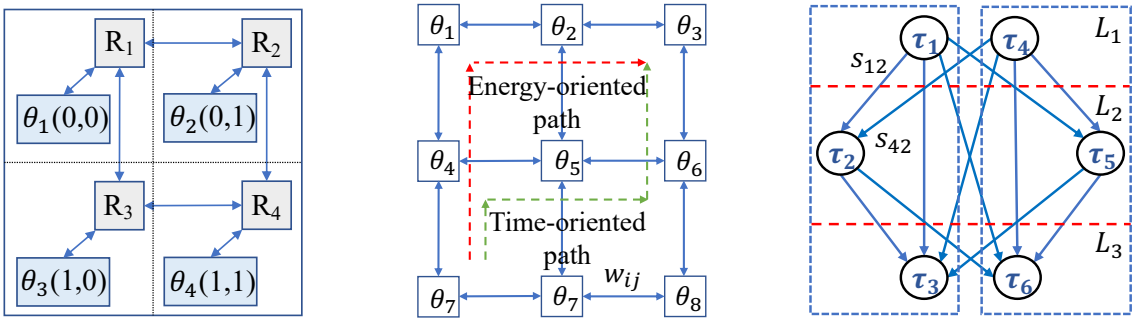
4.2.4 System Model

The task set consists of N periodic tasks $\{\tau_1, \ldots, \tau_N\}$, released at time 0, sharing a common scheduling horizon H. Each task τ_i is described by a tuple $\{C_i, D_i\, R_i^{th}\}$, where C_i is the Worst Case Execution Cycle (WCEC), D_i is the relative deadline and R_i^{th} its reliability threshold.

The target platform consists of M processors $\{\theta_1, \ldots \theta_M\}$ and M routers $\{R_1, \ldots R_M\}$, connected through a 2D-mesh network, as a 2×2 example shown in Figure 4.7a, due to its regularity, high bandwidth and short interconnections [100].

The processors are homogeneous and they support task-level DVFS. A processor has L different Voltage/Frequency (V/F) levels $\{(v_1, f_1), \ldots, (v_L, f_L)\}$. A typical power model [2] is considered: the processor power with (v_k, f_k) is $P_k = P_k^s + P_k^d$. The static power is $P_k^s = L_g(v_k K_1 e^{K_2 v_k} e^{K_3 v_b} + |v_b| I_b)$. The dynamic power is $P_k^d = C_e v_k^2 f_k$. C_e is the average switched capacitance. L_g is the number of logic gates. K_1, K_2 and K_3 are parameters depending on the processor type. v_b and I_b are the body-bias voltage and body junction leakage current. The computation energy of task τ_i is $e_i^{exe} = P_k et_i$, where et_i its execution time.

The NoC topology is described by a directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where the vertexes represent each processor $\theta_k \in \mathcal{V}$, while the edge $l_{ij} \in \mathcal{E}$ represents a direct communication link between the routers of processors θ_i and θ_j. The goal of the task deployment is energy balance under real-time and reliability constraints, our approach explores both energy-oriented and time-oriented paths as available options to transmit data, as shown in Figure 4.7b. Note that, the path with minimal energy can be different from the path with minimal latency [107]. Based on the NoC communication model and the Manhattan distance between the source and destination processors, a positive weight w_{ij} is associated with the edge l_{ij}. For the energy (time)-oriented path, the weight w_{ij} represents the energy (time) required for transmitting and receiving a unit of data between processors θ_i and θ_j. Hence, the aim of energy (time)-oriented routing is to find the shortest path, e.g., according to Dijkstra's algorithm. From the graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, we obtain an energy matrix $\boldsymbol{e} = [e_{\beta\gamma k\rho}]_{M\times M\times M\times P}$ and a time matrix $\boldsymbol{t} = [t_{\beta\gamma\rho}]_{M\times M\times P}$, where $e_{\beta\gamma k\rho}$ represents the energy consumed at processor θ_k, if a unit of data is routed from θ_β to θ_γ through the ρ^{th} path, while $t_{\beta\gamma\rho}$ denotes the time required to transmit a unit of data from θ_β to θ_γ through the ρ^{th} path. When two dependent tasks are mapped on the same processor, the communication cost is zero [237].



(a) Multicore platform (b) Multi-path routing. (c) Task duplication.

Figure 4.7: An example of NoC-based multicore system.

As in previous section, this work focuses on transient faults and adopts the Poisson fault probability model [275]. When the reliability of task τ_i is lower than its threshold R_i^{th}, task τ_i is duplicated, considering that it is unlikely to have faults occurring concurrently in both copies [95]. Note that the task duplication affects the task model, and thus, the computation and communication cost. For instance, tasks τ_1, τ_2 and τ_3 are the original tasks, and tasks τ_4, τ_5 and τ_6 are their copies in Figure 4.7c. By task duplication, the task dependencies change, e.g., due to the dependency of τ_1 and τ_2, τ_4 and τ_2 become dependent and τ_4 generates data to τ_2.

4.2.5 Problem Formulation

Given a set of tasks, our goal is to map them on M processors taking into account inter-processor communication, such that the overall energy consumption, both for computation and communication, is balanced, under task real-time and reliability constraints. To achieve that, we determine 1) which processor should the tasks be executed on (task-to-processor allocation), 2) on which path the data re transferred (routing path selection), 3) what frequency should be used for the tasks (frequency-to-task assignment), 4) which task should be duplicated (task duplication), and 5) when should the task start based on dependencies (task dependencies). Overall, a processor is able to execute one task at a time instance (task non-overlapping constraint), the tasks should finish before their deadline (real-time constraint) and meet their reliability threshold (reliability constraint). Table 4.5 summarizes the constraints and variables used, along with the type of the proposed problem formulation and solutions.

Table 4.5: Summary of problem formulation in [C25].

Constraints	**Binary**	**Continuous**
Task-to-processor allocation	✓	
Routing path selection	✓	
Frequency-to-task assignment	✓	
Task duplication	✓	✓
Task dependencies	✓	✓
Task non-overlapping	✓	✓
Real-time	✓	✓
Reliability	✓	✓
Type	MINLP	
Solution	O + H	

4.2.6 Optimal approach

Similar to the previous Section 4.1, we use variable replacement to eliminate the non-linear items related to task duplication, task allocation and frequency assignment to safely transform the MINLP to an equivalent MILP. Then, the MILP is solved using optimization solver tools, such as Gurobi.

4.2.7 Heuristic method

The proposed heuristic method applies three phases, as depicted in Figure 4.8 Since the processors are homogeneous and have the same V/F levels, the decision regarding the frequency assignment can be performed independently. If a frequency is assigned to each task, and then, the tasks are allocated to the processors or different paths are selected for data transmission, the frequency assignment remains valid. In addition, as task duplication is determined by frequency assignment, it should be still jointly optimized. Therefore, we first solve the Frequency Assignment and Task Duplication (FATD) problem, where the maximum energy consumption of each task execution is minimized. This minimization helps in balancing the energy consumption of the processors during task allocation occurring in the next step. Since task allocation and path selection are currently unknown, the communication cost (time and energy) is currently not considered. The problem is an INLP problem, as the variables regarding frequency assignment and task duplication are coupled non-linearly. We propose a heuristic based on the Greedy Algorithm [187], where the frequencies

$\{f_1, \ldots, f_L\}$ are assigned iteratively for each task τ_i in increasing task index order, with the aim to minimize the increase of energy consumption among the tasks that have already been assigned a frequency. If the real-time constraint cannot be satisfied with a given frequency, the frequency is excluded. When the frequency of the original task τ_i is known, the decision for duplicating the task can be computed. A similar method is used to assign a frequency to the duplicated tasks taking into account the reliability constraint.

With the frequency assignment and task duplication decisions, the computation time and energy of task τ_i are computed. The next step is to determine the task allocation, task sequence and task start time. To balance the energy consumption of the processors under the task sequence and the task non-overlapping constraints, the Task Allocation and Scheduling (TAS) problem is to minimize the maximum summation of energy consumption for computation and communication. Note that the communication energy and communication time are influenced by the routing path selection, which is unknown at the current step. Thus, we set these values to the average communication time of task τ_i and average communication energy of processor θ_k, respectively. Once the path selection is determined, these values are updated accordingly. The problem is an MINLP and solved based on the following approach. The in- and out-degrees of all tasks are calculated and the tasks are divided into layers. Tasks in the same layer are sorted in a descending order based on their execution cycles. If the tasks in same layer have same execution cycles, they are ordered randomly. Task allocation is performed following this ordering taking into account task sequence and task non-overlapping constraints.

The final phase determines the path selection. Note that the path selection does not influence the computation energy and time, only the communication energy and time. The Routing Path Selection (RPS) problem also minimizes the maximum summation of computation and communication energy consumption. To solve this ILP problem, we propose an algorithm that determines iteratively the routing path for each pair of processors $(\theta_\beta, \theta_\gamma)$. The aim is to find a path for θ_β and θ_γ, that causes the minimum increase of communication and computation energy among these processors. During this process, the real-time constraint should be satisfied.

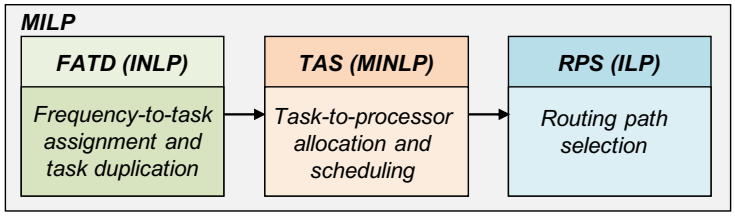


Figure 4.8: The structure of heuristic approach.

4.2.8 Evaluation

This section presents the main results to evaluate the behavior of the proposed approach, whereas the complete evaluation can be found in [C25]. We present the energy consumption for the proposed optimal approach compared to an optimal approach using single path routing, where the path selection is fixed. Furthermore, we compare the obtained optimal solutions when the objective of our approach is to balance the energy consumption (BE) and minimize the energy consumption (EE). Last, we compare the feasibility, energy consumption, and computation time of the proposed heuristic compared to the optimal approach. The task deployment problems and algorithms are implemented in Matlab and the optimal solutions are provided by Gurobi.

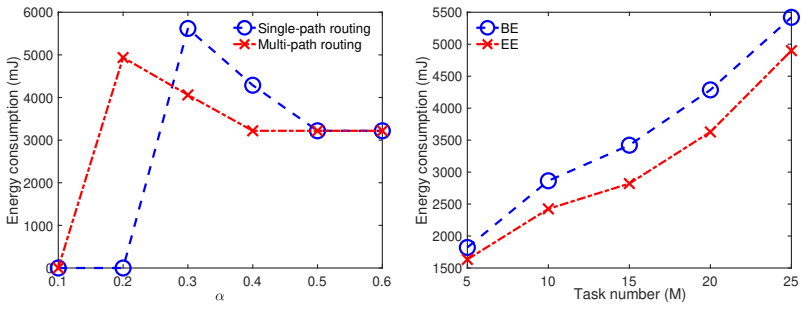
Experimental set-up

Platform: The evaluation is performed considering a 4×4 2D-mesh NoC. The modeling of the energy consumed by processors and routers is based on [100]. The following parameters are considered in our experiments: the number of processors (N), the number of the tasks (M), the number of V/F levels (L).
Benchmarks: The task set is created by randomly generating task graphs with a total number of task N equal to 5 up to 25 tasks, with a step of 5. The WCEC of a task are assumed to be within the range $[4 \times 10^7, 6 \times 10^8]$ [286], provided from the execution of MiBench and MediaBench benchmark suites [203].
Constraints: The reliability threshold is set to $R_i^{th} = 0.9995$. The real-time constraint is given by the sheduling horizon $H = \alpha \sum_{i \in \mathcal{C}} (t_{i,ave}^{comp} + t_{i,ave}^{comm})$, where $\mathcal{C}$ is the set of tasks belonging to the critical path. $t_{i,ave}^{comp} = (\max_{\forall l}\{\frac{C_i}{f_l}\} + \min_{\forall l}\{\frac{C_i}{f_l}\})/2$ and $t_{i,ave}^{comm} = M_1(\max_{\forall \beta,\gamma,\rho}\{t_{\beta\gamma\rho}\} + \min_{\forall \beta,\gamma,\rho}\{t_{\beta\gamma\rho}\})/2$ are the average computation time and communication time of task τ_i, respectively. The task relative deadline i sgiven by $D_i = t_{i,ave}^{comp}$.

Optimal approach

Figure 4.9a compares the energy consumption and the feasibility of the proposed optimal approach compared to the optimal approach using single-path routing. We set $N = 16$, $M = 20$, and $L = 6$. With small α, e.g., $\alpha = 0.1$ or $\alpha = 0.2$, the problem is infeasible, since the constraints are hard to satisfy. The problem feasibility increases with α; the larger the value of α, the smaller the energy consumption. This is because the feasibility region of the problem enlarges with α, and the task deployment is a minimization problem. Under the same value of α, multi-path routing has a higher problem feasibility and achieves lower energy consumption than single-path routing, because the path selection $c_{\beta\gamma\rho}$ is considered in the optimization.

Figure 4.9b compares the energy consumption of the proposed task deployment scheme, with the goal of Balancing the Energy consumption (BE), and the task deployment scheme with the goal of minimizing the Energy consumption (EE), i.e., $\min \sum_{k \in \mathcal{N}} E_k^{all}$, where $E_k^{all} = E_k^{comm} + E_k^{comp}$ is the total energy of processor θ_k. The total energy consumption of EE is lower than BE (average 13.62%). However, the value of ϕ for BE is smaller than EE. This is because ME allocates the tasks to the same processors to reduce communication energy. Therefore, some processors will consume more energy than others. However, BE avoids this trend in order to achieve energy balance.

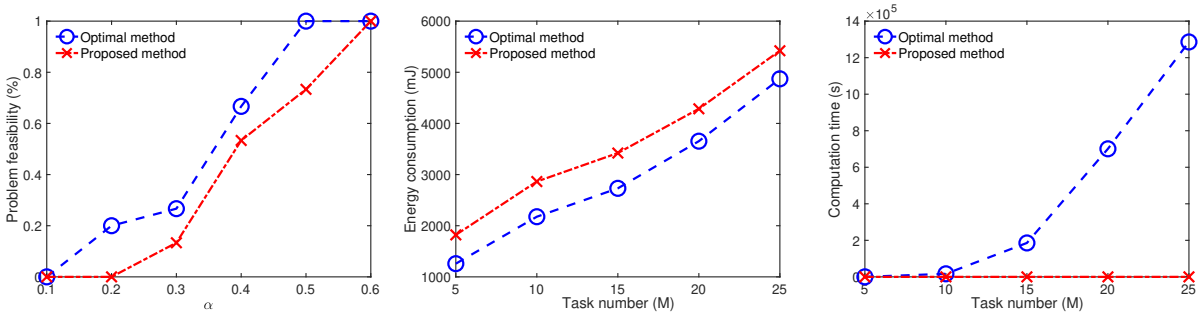


(a) Routing path comparison. (b) Objective comparison.

Figure 4.9: Energy consumption of optimal approach compared to a) a single-path approach, and b) having the objective of minimizing energy consumption.

Heuristic method

Figure 4.10a shows the feasibility in solving the task deployment problem for the optimal and the heuristic methods. The experiments have been repeated $n_a = 30$ times with different task graphs. The used metric is the problem feasible ratio $\delta = n_f/n_a$, where n_f is the number of experiments with feasible solutions. Figure 4.10a shows that δ increases with α. The constraints are relaxed with α, thus the feasible region of problem is enlarged. The optimal feasibility is higher than the heuristic, since it optimizes the variables concurrently, whereas the heuristic optimizes the variables step by step. Figure 4.10b compares the energy consumption of the solutions achieved by the proposed heuristic and the optimal solution. We observe that the solution of the heuristic has a higher, but still acceptable, energy consumption (average 26.05%) than the optimal solution, since it only provides a feasible solution. Figure 4.10c shows the computation time required for the optimal and heuristic approaches to find a solution. The algorithm computation time increases with task number M, as more variables and constraints are involved. On the contrary, the proposed heuristic has a negligible computation time, since it divides the problem into three subproblems solved in sequence.



(a) Problem feasibility. (b) Energy consumption (c) Computation time.

Figure 4.10: Comparison of optimal and heuristic approaches: a) feasibility, b) energy consumption, and c) computation time.

4.3 Conclusions

Efficient task deployment approaches are required in order to achieve low energy consumption, reliable and real-time execution for multicore systems enhanced with Dynamic Voltage and Frequency Scaling (DVFS). Overall, the majority of the existing approaches focus on a single DVFS scheme, which is usually the task level.

We proposed a Reliability-Aware Fault-Tolerant Task Mapping (RAFTM) approach based on partial duplication technique, which has been enhanced and evaluated for three DVFS schemes, i.e., task-level, processor-level and system-level, for independent tasks [C19, J25] and dependent tasks [J24, C23]. The original MINLP problems for the DVFS schemes have been equivalently translated to MILP, using a variable replacement method, and solved by Curobi optimization solver. We also proposed heuristic algorithms for each DVFS scheme in order to deal with the scalability issue of optimal approaches. Experimental results show that the proposed approaches achieve better energy saving and ability to obtain feasible solutions.

To take into account the inter-processor communication, a task deployment process has been

proposed for NoC-based multicore platforms with DVFS [C25], that balances the overall energy consumption, under reliability and real-time constraints. The deployment process is formulated as an MINLP problem and equivalently transformed to an MILP problem. Our formulation jointly optimizes frequency assignment, task allocation, task scheduling, task duplication and path selection. Moreover, a novel heuristic method is proposed to enhance scalability, achieving good solutions with low computation time.

Chapter 5

Perspectives

The future directions of this research focus on providing the means to design, in a near-optimal and efficient way, real-time and reliable embedded systems for safety-critical domains, with unreliable components, under multiple reliability threats. In this context, we can make the following observations regarding the limitations of existing works and motivate our future research perspectives:

- A large part of existing WA approaches focus on providing timing guarantees, considering fault-free architectures. Existing WAFA approaches mainly focus on typical hardware faults, i.e., soft errors and permanent faults, that have an impact of the functional behavior of the applications, i.e., failure to execute the application or to erroneous outputs. However, hardware faults can degrade the hardware components, due to permanent faults, or modify the execution flow, due to temporary faults, potentially increasing the execution time, and thus, impact the WCET estimations. Few approaches address the impact of hardware faults to the timing behaviour of applications, but they target only memories. Furthermore, existing reliability approaches focus on functional behavior and do not consider interferences.

  Nonetheless, with technology size reduction, faults in combinational logic and sequential logic of cores cannot be considered negligible anymore [150]. Therefore, the impact of such faults to the functional and timing behavior of applications should be considered. WAFA approaches are required to analyse the timing impact of hardware faults occurring not only in memories, but also in the processing elements and the interconnect of the platform. To achieve that, we will leverage our reliability analysis framework to incorporate not only functional behavior, but also timing behaviour of these faults. Furthermore, we extend our framework to include the impact of interference on shared resources during reliability analysis and reliable design, since it affects the timing vulnerability. Last, but not least, we will enhance WCET estimations with fault awareness and design low cost fault tolerance techniques to protect the system from these faults. Further details are provided in Section 5.1.

- The majority of existing WA and WAFA approaches remain at the task-level and core level considering mainly homogeneous multicore systems and COTS platforms, neglecting custom hardware designs and accelerators.

WA and WAFA approaches have to be leveraged for heterogeneous systems and systems with specialized hardware accelerators, which is the next promising architecture to deal the increasing demands for high computation capabilities in timely manner. Therefore, we will design hardware accelerators for WCET-aware and fault-aware systems to extend homogeneous multicore systems towards domain specific heterogeneous multicore architectures. To achieve that, we will adapt functional and timing reliability and WCET analysis frameworks and design real-time and fault tolerance techniques for hardware accelerators. As a first step, we will focus on modern hardware accelerators, such as accelerators dedicated for AI, as described in Section 5.2.

- A large majority of existing FA and WAFA approaches focuses on typical hardware faults, such as soft errors and permanent faults. Furthermore, the majority of FA and WAFA approaches usually focus to a single type of faults, whereas the system is susceptible to multiple types of reliability threats, potentially correlated with each other [11]. Last, the majority of approaches are usually focusing on a single abstraction layer.

Although soft errors have been considered as the most important ones, until recently [206], with the further ongoing reduction of transistors size (2-3 nm technology [91]), the small bounded guard-band margins are vanishing and system aging is becoming more and more sensitive to the workload [98, 130]. Different cores are subjected to different amount of stress as a result of varying workloads, leading to aging imbalance among cores [206]. Such aging effects increase electronic waste (e-waste) with significant impact on the environment. We will focus on dedicated cross-layer FA and WAFA approaches to efficiently deal with workload-dependent aging faults for safety-critical systems. Last, fault models, FA and WAFA approaches should be leveraged in order to consider multiple sources for reliability threats. Further details are provided in Section 5.3.

- Computing architectures of today are mainly based on CMOS technology. However, the continuous decrease of technology size has pushed CMOS devices to their limits, suffering from high static power consumption, reduced reliability, high cost and scaling issues. To overcome the limitations of the computer architectures of today, future computing systems will exploit emerging technologies, e.g., technology-level solutions for replacing CMOS memory elements, such as Oxide-based (e.g., RRAMs) [271] and Ferroelectric (e.g., FeRAM) [182] materials, and novel computation paradigms, such as Processing In Memory (PIM). However, these emerging technologies have not yet been considered for safety-critical domains, which have strict timing guarantees and reliability requirements.

Future computing systems will consist of heterogeneous architectures combining traditional computing elements, such as processors, with application specific hardware accelerators, exploiting both the emerging technologies and novel computing paradigms [110]. In order to be used in safety-critical domains with real-time and reliable guarantees, novel WA and WAFA approaches are required to analyse and design architectures with the new emerging technologies and novel computing paradigms used on shared resources. Further discussion is provided in Section 5.4.

The next sections describe the research challenges that we will tackle in the future and provide primitive ideas on how we will address them. Section 5.1 and Section 5.2 focus on short- and medium-term perspectives, while Section 5.3 and Section 5.4 on long-term perspectives.

5.1 Impact of hardware faults on timing behavior

Context

As mentioned in Chapter 1, safety-critical applications executed on embedded systems require guarantees for hard real-time and correct application execution. Nonetheless, the majority of existing WCET estimation approaches is fault-unaware, since during WCET estimation, the hardware of the target platform is assumed to be fault-free [45]. As reliability issues become imminent due to technology scaling, such fault-unaware WCET approaches become unsafe [105]. Real-time, approaches that provide timing guarantees, apply fault-tolerant techniques to detect, correct or mitigate hardware faults, and extend the fault-free WCET to include the time overhead of the applied fault-tolerant techniques [149]. However, the focus is on hardware faults impact the functional behaviour of applications [46].

In this context, not only the impact of faults on the functional behavior of applications should be taken into account, but also on the timing behavior.

State-of-the-art

Few approaches address the impact of faults on the timing behaviour of applications, but they target memory components, usually considering permanent faults into cache memories. Approaches focus on estimating the timing impact of faults to WCET, by accounting for the hardware degradation due to the presence of faults. For instance, static analysis probabilistically quantifies the WCET impact of permanent faults at instruction caches. The probability of an SRAM cell to be faulty is used to probabilistically evaluate how many additional cache misses may occur and upper bound their impact on WCET [105]. A measurement-based approach for permanent faults occurring to caches provides the WCET impact, when cache lines are disabled due to faults [248]. Approaches extend the aforementioned works to incorporate the timing impact of inserted fault tolerant techniques to detect, correct or mitigate faults. For instance, the worst-case additional misses, due to defected cache lines, are computed, and a parity bit is assumed for error detection of permanent and transient faults [52]. Static probabilistic timing analysis is performed assuming fault detection mechanisms that periodically checks caches for faults and disable faulty cache blocks, under permanent faults [50] and also soft errors [51]. The maximum delay, introduced by error detection and correction codes, is computed in [247], considering permanent and transient faults. Other approaches focus on mitigating the hardware degradation, due to occurring faults, using redundant hardware. As a result, the timing impact of faults on WCET is mitigated and the timing characteristics of hardware are maintained, leading to WCET estimations close to fault-free WCET estimations, despite the presence of faults. For instance, timing analysis is provided considering a reliable victim cache to replace faulty entries [6], an extra reliable cache way per set and a shared reliable buffer [106].

Nonetheless, with the technology size reduction, faults in combinational logic and smaller sequential logic of the system cannot be considered negligible anymore [150]. Although reliability approaches, from the embedded system design domain, are currently addressing these faults, they concentrate on functional behaviour and average performance, without considering WCET aspects. The majority of existing vulnerability approaches, as the ones presented in Chapter 1, focus on functional behaviour, i.e., checking for functional interruptions and binary correctness of the system under study [168, 265, 123, 269, 93, 47, 236, 262]. Few recent studies explore the impact of soft errors on the timing behaviour. They use software fault injection, focusing on average performance, whereas their application domain is limited to iterative methods, e.g., the performance impact is

given by the number of iterations required for iterative solvers to converge [169, 168] and their execution time [137].

To provide hard real-time and reliable execution, accurate timing vulnerability analysis is required, where hardware fault injection is performed in sequential and combinational logic, considering single-bit and multiple-bit faults in the cores and the interconnection. Then, fault impacts must be analysed and mitigated accordingly.

Future contributions

Our goal is to provide the means to analyse both the functional and timing behaviour of applications and to design reliable and real-time multicores. To achieve that, we foresee to provide a functional and timing reliability analysis framework, through fault impact analysis, considering fault propagation with respect to logic aspects, time aspects, application structure and core structure, under realistic fault models. As a next step, we will leverage typical WCET estimation approaches to consider the impact of faults on the timing behaviour of the applications. Last, we will design mechanisms for time-guaranteed and reliable execution, avoiding over-approximation whenever is possible. Initially, we will consider a single processor as our short-term goal. Then, the proposed approaches will be extended to take into account interferences, in order to be applicable for multicore architectures. This part will be performed during the ANR JCJC FASY project during 2022-2025. Our long-term goal is to obtain adapt these approaches for interconnection network and finally the overall multicore architecture, where we can analyse the occurring interferences in the shared resources.

Functional and timing reliability analysis framework

We will estimate the reliability of an application executed on a core, when hardware faults occur, considering not only the functional behaviour, but also the timing behaviour of the application.

Initially, we will extend our current framework (Section 3.3), which addresses only functional behaviour, to also consider the timing behaviour of the application. To achieve that, we will perform instruction reliability analysis and application timing behaviour analysis, considering both spatial and temporal vulnerability of the instructions. This means that the probability of the occurrence of a fault, during the execution of an instruction, depends on both the area and the duration of the specific processor components used by the instruction. Therefore, at the gate-level, we will perform instruction vulnerability analysis for the processor Instruction Set Architecture (ISA). Per instruction, we will obtain a set of error patterns, describing the output bits that changed due to a fault, along with two indicators regarding the instruction's capacity in masking and propagating faults. At the micro-architectural level, we will perform reliability analysis, taking into account the application structure and instruction vulnerabilities, including the number of execution cycles and the execution traces. To reduce the time for this analysis, analytical methods will be explored.

Last, our framework will be leveraged with the notion of interference. Interference, due to shared resources in multicore architectures, insert timing delays in the execution of memory instructions. As a result, a memory instruction will be stalled and reside longer in the pipeline of the core. This delay has to be taken into account in vulnerability analysis; the longer an instruction remains in the pipeline of a core, the higher is the probability to be affected by faults, i.e., its temporal vulnerability. Interference affect not only the temporal vulnerability of the memory instructions, but also the temporal vulnerability of dependent instructions, instructions already residing inside the pipeline, and the variables temporarily stored in the register file. The vulnerability analysis

framework will be enhanced with identification of the instructions, whose vulnerability is affected directly and indirectly by interference, along with estimation of the instruction vulnerability periods for the pipeline components and the register file.

Fault-aware WCET estimation

We will provide methodologies to estimate the WCET, considering the impact of soft errors occurring in processors.

Initially, we will chose an appropriate WCET estimation method that fits our goal. Since faults introduce non-determinism, since fault number and location are unknown a priori [6], we expect that Probabilistic Timing Analysis (PTA) is an appropriate method [73]. Furthermore, a measurement-based approach, potentially applied in code-snippet (region of source code) and combined with control flow information, is an appropriate method. This is because we speculate that faults, affecting the control flow, will have an impact on execution, and thus, a fine-grained analysis will be required. As a first step, the selected probabilistic timing analysis technique will be applied on the processor in order to obtain the distribution of fault-free WCET estimations. To achieve that, we will identify a set of scenarios, i.e., a sample of input states and initial hardware states that lead to measurements that expose representative executions of the system [73]. Application inputs lead to different executions in applications with multiple execution paths. The impact of core hardware state is expected to be low. Furthermore, the hardware state can be always initialised in the worst-case, i.e., reset the processor and flush the pipeline. Timing and structural analysis of the execution traces will identify the execution paths and input data that lead to the executions with high WCET.

Then, this approach will be combined with the reliability analysis framework, in order to inject faults and exploit their impact in WCET estimation. The probabilistic timing analysis will be leveraged to include the probabilities of fault occurrences. These probabilities will be derived from the spatial and temporal vulnerability of the instructions and the fault models, obtained from the reliability framework. For each input data, a set of error patterns will be injected to observe the impact on WCET estimations. However, not only random input data and error patterns should be analysed, but also the representative scenarios for WCET estimation. Simulating the execution with the representative scenarios, we will obtain representative pipeline traces. The reliability framework will be extended to use representative pipeline traces at the gate-level analysis. As a result, after gate-level fault injection, we will obtain instruction error patterns, error masking and error propagation factors, related to the application representative scenarios. At the microarchitecture-level, we can use the representative scenarios, along with the corresponding error patterns, to obtain the impact of soft errors on the WCET estimation.

To include the impact of interference in the fault-aware WCET estimations, we will be based on a graph model of the application as a graph, inspired from our work [C8], where each code snippet is characterized with fault-aware WCET estimations, without interferences, and Worst-Case Memory Accesses (WCMA), obtained by profiling code-snippet traces. The use of analytical methods will be explored in order to estimate the impact of interferences on the fault-aware WCET estimations, depending on the number of applications executed in parallel and the arbitration policy. We will focus on policies that provide a deterministic bound to the longest wait time for a request and support measurement-based probabilistic WCET techniques, such as round-robin and TDMA [45].

Mechanisms for time-guaranteed and reliable execution

We will design low-level mechanisms, with low area and low WCET overhead, that will eliminate faults when they occur, before they are propagated and expressed as functional errors and timing variations in the application execution.

Initially, we will use the aforementioned reliability analysis framework, combined with the probabilistic WCET estimation approach, to perform fault injection with the goal of identifying hardware and software sources that are most impacting, when faulty. Instructions use different parts of the processor for different durations, having different vulnerabilities and probabilities to be impacted by faults. Application executions have different timing behaviours, depending on the input data, which define the executed application paths. Depending on the position and duration of faults, they impact differently the application execution. We will identify the sources that impact the functional and timing behaviour of the application, when they are faulty. Such sources can be hardware parts of the processor and software parts of the application. Faults can be masked due to the core architecture, reducing the impact of some hardware parts. Faults may also be masked due to the application structure, reducing the impact of some software parts. Hardware and software parts will be classified as jitterless and jittery sources, based on whether they have an impact on execution time and WCET estimations. They will be characterized based on how probable a source is to be faulty, how many times a faulty source has affected the functional and timing behaviour, and the actual impact and the variations on the application execution time and WCET estimation.

Then, we selectively insert low-level fault-tolerance techniques, without or with negligible overhead in execution time and WCET, inside the core, in order to eliminate the impact of faults on the most vulnerable sources. Vulnerable sources are the sources that have high probability to get affected by faults, to lead to a high impact and to frequent impacts on the application execution. Hardware mechanisms are required as they can detect and correct faults that software approaches cannot. For instance, if the execution is stuck at the same memory address, the program counter evolution can be monitored by a hardware mechanism that checks the number of clock cycles spent on a single instruction [18]. We expect that low-level hybrid techniques will be effective and provide a high level of tolerance, e.g., low-level software approaches could be efficient for data-flow errors, while hardware approaches for control-flow errors. Initially, the sources that lead to functional interruptions will be protected, in order to allow the execution of the application to continue, improving system dependability. Then, sources that impact the timing behaviour and WCET estimation will be studied. We expect that the inserted low-level mechanisms will inherently protect the application output. Last, but not least, the remaining vulnerable sources will be protected. In order to avoid system over-design, run-time low-level mechanisms can be inserted, with low overhead in execution time and WCET, to deal with faults, if they occur. For instance, the core pipeline can be enhanced with hardware mechanisms that enable instruction roll-back. A exercising all possible application paths and fault occurrence is impractical, a hardware mechanism can monitor the execution cycles spent on a code-snippet to determine whether the code-snippet exceeded the WCET estimations and trigger a prevention action.

5.2 Real-time and reliable AI hardware accelerators

Context

Deep Neural Networks (DNN) [134] are currently one of the most intensively and widely used predictive models in the field of machine learning. DNN give very good results for many complex tasks

and applications, such as object recognition in images/videos, natural language processing, satellite image recognition, robotics, aerospace, smart healthcare, and autonomous driving. Nowadays, there is intense activity in designing custom AI hardware accelerators to support the energy-hungry data movement, speed of computation, and memory resources that DNNs require in order to realize their full potential [163].

In safety critical systems, there is a need for deterministic behavior and guarantees regarding the WCET on the target hardware [64]. Furthermore, AI hardware accelerators are subject to hardware faults, which can cause operational failures, potentially leading to important consequences, especially for safety-critical systems. AI hardware accelerators have some inherent resilience to faults, due to the learning process that can circumvent faults to a large extent. However, faults can still occur during the operation of the accelerators after training. Since some neurons are more critical than others, as they have a higher probability of propagating errors to the final DNN output [141], inference can be significantly affected, leading to DNN prediction failures that are likely to lead to a detrimental effect on the application [256, 217, 148]. Furthermore, although explaining AI decisions is highly desirable in order to increase the trust and transparency in AI, the role that faults can have in AI decisions has been neglected till now. Note that, if the hardware is affected by faults, leading to faulty decisions, then any attempt for explainability will be either inconclusive or misleading.

Therefore, ensuring real-time and reliable execution for AI hardware accelerators is crucial, especially when they are deployed in safety-critical systems.

State-of-the-art

Regarding reliability, common reliability analysis frameworks and standard fault-tolerance techniques used in traditional computing, such as TMR and ECC for memories, are not effective for AI hardware accelerators in general since they incur prohibitive overheads. Due to the large-sized DNN architectures and memories required for weight storage, these techniques turn out to be very inefficient. Furthermore, periodic re-training is impractical at chip-level, since the training set is too large to be stored on-chip and has to be communicated from the cloud. Besides, performing the training on-chip requires a large dedicated on-chip infrastructure, which increases prohibitively design complexity and area overhead.

Approaches to evaluate the DNN reliability are similar to traditional hardware and are typically based on fault injection. Fault injection at the application-level is the commonly used technique due to lower cost, fast execution and ease of deployment, while it is independent from the hardware architecture. For instance, a fault injection environment is built on an open-source DNN framework implemented in C and CUDA language to inject permanent faults on the weights of the neural network [37] and extended for reduced bit-width precision [217]. Fault injection frameworks are built on TensorFlow, where faults can be injected at the TensorFlow graph level by corrupting the outputs of the most common mathematical operators [30, 61, 62]. However, such approaches lack information of the underlying hardware platform, and thus, are less accurate. Fault injection at the hardware-level uses a model of the target hardware architecture running the DNN, being more accurate and close to the real hardware, but more time consuming. FIdelity framework is built on TensorFlow [108] and uses information obtained from architectural description to model soft errors in application level, improving accuracy. The propagation of soft errors is explored with open-source DNN simulator, where DNN hardware components are associated with simulator code [141]. Fault injection is performed at the RTL, considering both the application information (the DNN weights, inputs, and the intermediate values) and architectural information (the specific data representation

and the amount of computational resources) [205]. To reduce simulation time, the FI framework in [217] uses a pipeline mechanism that exploits the sequential execution of DNN layers. Overall, the majority of approaches use statistical fault injection, since exhaustive approaches are not possible, while few approaches propose smarter fault injections. Similar to traditional hardware, the timing impact of faults has not been explored yet.

Regarding real-time execution, there is still a large amount of work and proof needed to show the capability of computing a WCET for DNNs [74], as the majority of existing approaches focus on traditional embedded real-time benchmarks. Approaches focus on using measurement-based WCET, e.g., a probabilistic WCET (pWCET) of DNN based image classification models has been proposed, considering variation in the input size, using DNN inference time and extreme value theory [131]. Other approaches use static methods, e.g., a programming framework generates C code from offline trained DNNs in order to perform statistic WCET estimation [74]. Last, implementation based approaches remove variations, e.g., a DNN implementation with the same control flow regardless of the input [64]. However, existing approaches focus on COTS and fault-free architectures.

Future contributions

Our goal is to design real-time and reliable AI hardware accelerators. As a short-term goal, we focus on reducing the complexity to analyse and design reliable AI hardware accelerators. To achieve that, we will leverage our reliability analysis framework towards AI hardware accelerators. Then, we aim at designing low-cost, customized fault-tolerance strategies for AI hardware accelerators. This work will be performed in the RE-TRUSTING project (2022-2025). As a next goal, we will extend the obtained reliability analysis and protection mechanisms to not only take into account the functional, but also the timing behavior, of hardware accelerators. To achieve that, we will combine our findings regarding reliable AI hardware accelerators with the findings of timing analysis and protection of processors, presented in Section 5.1.

Reliability analysis framework for AI hardware accelerators

Fault severity is not considered in traditional computing hardware, since there is no intrinsic fault-tolerance and any fault is treated as a reliability hazard. However, classifying a fault as malignant is essential for AI hardware accelerators. To grade fault severity and identify the set of malignant faults through fault injection simulation poses a great challenge for AI hardware accelerators, since the number of fault locations explodes; every neuron and synapse is considered as a fault injection candidate, and each fault injection requires performing inference on the complete testing set to assess the fault effect, which is intractable from a computation point of view. Thus, a new paradigm is required that is specifically tailored to AI hardware accelerators.

To manage this complexity, we will leverage our cross-layer reliability analysis to AI hardware accelerators to enable fast fault exploration and identification of malignant faults. Initially, we will decompose the hardware architecture into elementary components, where detailed fault simulation can be performed at gate-level for each component separately. Note that, this is feasible since AI hardware accelerators have a lot of space redundancy and a repetitive architecture, i.e. replicated multiply-and-accumulate units, neurons, etc. We will collect and group the observed faulty behaviors in order to create fault models per such hardware component. The obtained fault models can be used at higher abstraction layer, where they will be injected through simulation at a high-level algorithmic description of the DNN, i.e., PyTorch, TensorFlow, Keras, etc.

Our framework will support single and multiple fault injection scenarios, at any point of the operation, with the goal of determining the set of malignant faults that mostly impact the accuracy of classification (or other DNN objectives, such as image segmentation) during the inference phase. However, DNN inference is very complex, especially on large models and for large datasets, and thus, a large number of faults have to be injected. Since the complexity of fault injection grows linearly with DNN inference complexity and the number of injected faults, we will leverage statistical and analytical methods to prune the fault space. We will rely on profiling DNN hyper-parameters, such as weight values distribution, neuron activation distributions, at the layer and even kernel level. The profiling will be done one time by running the whole testing dataset. Once the hyper-parameters distributions are determined, the next step will be to analyze potential correlations among them, linking the local sensitivity at the layer/kernel levels with the global accuracy. For instance, if a fault forces a weight value in the first layer to be away from the profiled distribution, but this local violation does not lead to a DNN inference failure. With the knowledge of such relations, it will be possible to immediately stop the fault injection process without the need of completing the full inference, thus saving a significant amount of time and improving high-level fault injection performance on larger datasets or more complex DNN models.

The fault injection and simulation framework will be further optimized by implementing the early stop criterion related to performance metrics. In order to further push the performances and reduce injection time, optimized C/C++ code can be exported and executed on GPU. Finally, direct execution of the network (or a portion) on the FPGA accelerators can also be used.

Selective fault tolerance

The next step is to develop the error correction and mitigation operation of the fault-tolerance scheme. The larger the set of target faults is, the higher the cost to implement fault tolerance. To reduce fault tolerance overheads, based on the fact that many faults will be probably benign, identifying the malignant faults and targeting only these in a fault tolerance scheme will reduce the implementation cost, i.e., there will be no over-test and the on-chip mechanisms for self-test and error correction can be more localized. Moreover, classical approaches for traditional computing hardware consider one fault at a time. In the case of AI hardware accelerators, this is not necessarily true since synaptic weights are constant data written only once in the memory. This means that external perturbations can have cumulative effects, leading to many faults affecting the hardware. Therefore, we will define fault tolerance strategies under multiple fault scenarios. More specifically, we will develop low-cost heterogeneous fault tolerance techniques for AI hardware accelerators, by combining passive and active fault tolerance strategies.

Passive fault tolerance is a proactive mean to deal with several anticipated fault locations and types. We will explore modern training algorithms, whose primary aim is to reduce over-fitting and improve the generalization ability of the network, such as removal of unnecessary nodes/weights, replication of critical neurons, evenly distributing/splitting synaptic weights, noise/fault injection during training, restricting weights to low values, etc. These techniques equalize the importance of neurons and synapses and this property will be key for reducing the impact of faulty neurons and synapses as well, and they have not been thoroughly evaluated yet for DNNs.

Active fault tolerance mechanisms will be used to deal with the rest of the faults. Such an active approach will be composed of self-test and error-correction mechanisms. Component-level BIST mechanisms will be explored, aiming at low area and power overhead and at transparency to the operation of the accelerator. Error correction mechanisms will be designed to deal with detected faults, such as the re-execution of the task of a faulty component, the dynamic reschedul-

ing/mapping of the DNN to the hardware accelerator, bypassing faulty components and replacing them with spare ones, weight-shifting to fault-free elements when faulty links are detected, selective low-precision TMR, most-significant bits reinforcement, standby sparing and correcting codes, etc. Reconfigurable fault tolerance mechanisms that can be tuned with respect to the algorithmic constraints (e.g., DNN hyper-parameters such as weights/inputs data representations) and the occurrence of "burst" faults, due to cumulative fault effects, will be also explored.

5.3 Workload-dependent aging and multiple reliability threats

Context

The accuracy of the hardware may degrade over time due to the aging effects on its components, due to several aging phenomena, such as TDDB, BTI, HCI, Ionizing-Radiation, etc [29]. These aging phenomena degrade the transistor's threshold voltage over the lifetime of the underlying circuit, resulting in slower transistors [133]. Such transistors can eventually lead to faulty operations, when the critical paths become longer than the cycle time. If such aging effects remain unexplored and uncontrolled, the system may have an unacceptable behavior. Furthermore, the importance of workload-dependent faults to the circuit aging has been recently highlighted in modern technologies [130]. With the further ongoing reduction of transistors size, system aging is becoming more and more sensitive to the workload [98, 130, 89, 58]. Different cores are subjected to different amount of stress as a result of varying workloads, leading to aging imbalance among cores [206]. Last, devices are susceptible to various fault sources, which occur concurrently and have inter-dependencies [10, 11].

In this context, dedicated WCET-aware and fault-aware approaches are required to efficiently deal with workload-dependent aging, not only for processors, but also for dedicated hardware accelerators and multicore systems.

State-of-the-art

Existing fault-aware approaches analyse the impact of aging sources, such as TDDB, BTI, HCI, radiation, for traditional hardware components. Typical approaches used for aging detection are based on monitoring temperature, critical path, clock frequency, workload, circuit state and voltage [122]. Aging mitigation techniques are based on dynamic voltage scaling, dynamic frequency scaling, aging compensation, body-bias adaptive and workload reduction [122]. Few real-time approaches deal with aging in memories, e.g., considering HCI and BTI for L1 caches [258], and register files [259].

Few approaches focus on exploring the workload-dependent nature of aging. A run-time mechanism re-executes faulty instructions, mitigates performance degradation by configuring the clock frequency, and performs resource allocation to the applications taking into account the performance degradation [211, 289]. A run-time adaptation pro-actively slows down aging when the first instructions starting to fail, by scheduling in dedicated functional units and balancing pipeline stages [181]. The critical path delay is monitored at run-time, and when the delay is increased, the critical path is cut in a way to minimize the accuracy loss [126]. Task graph retiming, ordering, assignment, and dynamic voltage selection is applied to extend the system lifetime [58]. At each abstraction layer, a set of fault tolerant mechanisms is selected to cover faults that escaped the lower layer [109]. Off-line cross-layer fault-tolerant solutions are used at run-time to adapt the fault-tolerance means [222]. Last, few approaches exist for aging mitigation of CNN accelerators, e.g.,

focusing on BTI in on-chip memories storing weights [102] and BTI and HCI in on-chip memories storing activations [133]. The lifetime of AI hardware accelerators has not been thoroughly investigated yet [102, 146]. However, existing approaches consider average execution, without focusing on WCET aspects.

Last, but not least, the majority of existing approaches focus on a single type of reliability threat. In reality, the system can have multiple degradation effects from different sources with inter-dependencies [10, 11]. Few works aim to jointly consider multiple reliability threats using failure-equivalent circuits and applying the sum-of-failure-rates rule, assuming that the considered degradation effects are independent. However, the causes of degradation effects are of physical origin, and they are interdependent from a physical point of view [11].

Future contributions

Our overall goal is to provide the means to design real-time and reliable systems considering workload-dependent aging and multiple reliability threats. Since applications are dynamic in nature, they lead to a dynamic workload affecting differently the stressing of the hardware components. Different reliability threats can be mitigated more efficiently at different layers, depending on their characteristics. Therefore, we believe that Cross-Layer Reliability (CLR) methods are required, since they can exploit application, deployment and hardware layers, opening up new opportunities for designing application-specific reliable embedded systems. Furthermore, CLR approaches need to be extended with WCET aspects to be applicable in safety-critical domains. As a first phase, we will analyse the workload-dependent nature of hardware aging, how WCET estimations can be adapted at run-time to take into account aging effects, and propose cross-layer approaches to mitigate workload-dependent aging effects under real-time constraints. Initially, we will focus on processors, and then, AI hardware accelerators. As a long-term goal, we will explore how fault models are correlated, when several reliability threats occur concurrently on the system, and propose analysis and cross-layer CLR methods for multiple reliability threats for real-time systems.

Workload-dependent aging real-time and reliability analysis

As a first step, we will characterize the workload-dependent nature of aging effects on the functional and timing behavior of a processor.

Toward this goal, we will apply a component-based approach, where we explore the impact of aging sources (e.g., TDDB, BTI, HCI) on major components of processor (e.g., multipliers, arithmetic and logic units, decoders) per instruction, under different scenarios through simulation. As the aged circuit delay depends on the operation conditions over lifetime and the workload, different scenarios can have different critical paths per aged component. Such scenarios can be identified based on the range of the operand values per instruction for different application contexts. Analytical methods will be also exploited to estimate the aging effects and critical paths. Through a parser, we can obtain the graph of the gate-level netlist and gather timing information of the cells. Aging fault models will be used to characterise aging for each cell type. The probability of each net in the design to have a specific logic value will be obtained, following approaches similar to the power analysis of programs based on the scenarios and/or worst-case analysis. Combining the probability of input vectors of a cell with its aging model, we can compute the probability of the cell's aging. Then, critical paths can be defined based on the forward cones of the component based on different input scenarios. With this analysis we can obtain aging models to be used in micro-architecture and application level, taking into account the hardware and software structure.

Furthermore, due to distinct path aging rates, which can occur due to local layout configurations of different gates, the critical paths of a component may change during its lifetime. However, existing WCET estimation approaches have not yet considered aging impact on the hardware components, which impacts the critical paths of computing elements. Since aging depends on the workload, we believe that parametric expressions will be appropriate for WCET estimation. Such expressions can support context-sensitive hardware and software timing effects and parameters regarding the application, e.g., parametric loop bounds depending on the input data, and the hardware, e.g., effects of aging at different components.

Our approaches will be adapted for AI hardware accelerators by performing a component-based analysis, e.g., on multiply-and-accumulate units and on-/off-chip memory in training and inference phases, and investigate the impacts of aging on the accuracy loss and timing behavior. Furthermore, by adapting fault models and combining the different reliability analysis frameworks, we will analyse the impact of multiple reliability threats on functional and timing behavior of the system.

Real-time cross-layer reliability methodologies

Real-time cross-layer reliability approaches are a promising solution towards workload-dependent aging and multiple faults, as they combine approaches from the application, deployment and hardware layers. In this context, we will propose design-time and low overhead run-time cross-layer approaches, so as to meet the real-time constraints under aging and multiple reliability threats.

At the application layer, the same application running with different input data (application context) does not always require results with the same level of QoS. For instance, aircraft collision detection systems compute trajectories using radar data. Trajectories for stationary obstacles are computationally simpler that those of moving obstacles [165]. Furthermore, applications have critical regions, for which correct execution is required, and forgiving regions, which can tolerate significant changes [210]. An exploration framework, based on simulation and profiling, will allow us to estimate the minimum acceptable QoS with respect to the application context and characterize the tolerance and WCET of the application regions. This information can be used to adapt the execution at run-time.

At the deployment-level, novel methodologies are required to map near-optimally the tasks onto the processors, and potentially hardware accelerators, considering different application contexts, provided by the application layer, aging effects and interference. Furthermore, reliability approaches at the task-level increase the workload, and thus, aging effects. Tasks running in parallel impact the interference, and thus, the WCET. Furthermore, the workload running on each core impacts the aging effects, leading to asymmetric aging of the processors. Such aging should be taken into account during task deployment both at design-time and at run-time.

At the hardware level, the processor, and potentially hardware accelerator, configurations impact several metrics, such as the execution time, the WCET, the reliability, and energy consumption. For instance, the higher the processor frequency, the lower the execution time, and thus, the WCET estimation. Furthermore, inserted fault-tolerance at the hardware level and dedicated function units may increase the components delays. Last, the connection of an accelerator as an external co-processor, connected to the communication network, increases interference which impacts WCET estimations.

Last, cross-layer run-time adaptation is required to deal with aging and multiple reliability threats. Mechanism for short-term adaptation are needed in order to deal with correcting workload-dependent faults and soft-errors. Any available time slack due to the actual execution of the tasks will be computed at run-time and check whether there is a need for creation of extra slack via

hardware adaptations, e.g., frequency increase, deployment adaptation, e.g., cycle stealing, or application adaptation, e.g., QoS reduction, in order to provide timing guarantees. Mechanisms for long-term adaptation are required in order to deal with circuit aging and its impact on system execution. The hardware can be configured to hide circuit aging, e.g., via voltage/frequency changes. This change will impact the WCET, and thus, such an impact needs to be estimated and taken into account for the timing guarantees of the system. Note that, the new WCET may turn the current task mapping infeasible. Therefore, an adaptation at the deployment layer, e.g., through a new mapping, or at the application layer, e.g., by reducing the QoS, is required.

5.4 Real-time and reliable emerging technologies

Context

Today's computing architectures are mainly based on CMOS technology, facing major limitations, which makes them unable to meet the growing requirements in performance and energy efficiency: Power Wall, Memory Wall and Instruction Level Parallelism Wall [190]. Manufacturing issues continue escalating for the current dominating CMOS technology and its scaling comes to an end, as feature sizes in the order of a single atom are approached. Due to the aforementioned limitations, alternative emerging technologies are explored in order to provide performance requirements with affordable costs. Such promising advances are technology-level solutions, developed in order to replace CMOS elements, and novel computing paradigms, such as PIM. Consequently, future computing systems will consist of heterogeneous architectures, merging general purpose computing elements (e.g., CPU/GPGPU) with application specific hardware accelerators, exploiting both the emerging technologies and novel computing paradigms [110].

In this context, design and analysis methodologies are required in order to incorporate emerging technologies and novel computing architectures in future computing architectures, especially for safety-critical systems, requiring real-time and reliable execution.

State-of-the-art

Regarding emerging memory devices, the majority of existing PIM approaches focuses on the design and modeling of PIM computation units [178]. Both volatile and non-volatile memory devices have been explored. Volatile memory devices correspond to the CMOS technology, i.e., static RAM and dynamic RAM. Examples of non-volatile memory devices are Phase Change materials (e.g., PCRAM) [270], Oxide-based (e.g., RRAMs) [271], Ferro-electric (e.g., FeRAM) [182] or Magnetic (e.g., Spin Transfer Torque (STT) - MRAM) [87]. A survey on memory-centric computer architectures can be found in [92]. Every type of emerging non-volatile memory have unique features, with different applications in the memory hierarchy, but also unique failure mechanisms that cannot be modelled with the traditional faults models [99]. Note that, the memory in memory-centric computer architectures has two configurations, i.e., storage and computing, which poses additional requirements in reliability analysis and testing. To be considered as a competent rival for conventional memories, their reliability characteristics require improvements [57]. Some works consider emerging devices with applications which are inherently tolerant to faults, such as AI inference [69, 176, 121].

Furthermore, as far as we know, few recent works focus on the use of emerging devices for safety-critical systems. They mainly focus on STT memories, e.g., the average system performance

and WCET implications are analysed for STT-MRAM [14] and partial WCETs are obtained and used for data allocation on STT-RAM with variable retention times [38].

Although works exist regarding the computational aspect of memory-centric computer architectures [92], PIM is a relatively new concept and work is still required in order to be incorporated to heterogeneous multicore architectures [178]. The architectural challenges of memory-centric architectures need to be revisited in order to harness the full potential of emerging memory technologies [92], whereas further research is required to develop appropriate analysis and design methodologies that can provide real-time guarantees for safety-critical systems.

Future contributions

Our aim is to support the design of heterogeneous multicore architectures combining emerging technologies and novel computing paradigms for safety-critical systems. Towards this direction, the analysis of functional and timing behavior for emerging memory devices and memory-centric computer architectures, is required. Furthermore, novel DSE approaches are needed to select and configure memory-centric architectures, determine which part of an application is executed on which memory computation unit, meeting real-time guarantees and providing reliable execution.

Timing and reliable analysis for emerging memory architectures

Emerging memory devises have different physical characteristics, while several memory-centric computer architectures exist which perform operations in different ways, impacting the timing and reliable execution of the applications.

As a first step, we will focus on identifying how characteristics of memory-centric computer architectures can affect the WCET estimations. First of all, the time and nature of possible operations is different among emerging memory devices [252] and architectures [92], which affects the execution time, and thus, the WCET estimations. Furthermore, due to the less reliable nature of emerging technologies, mitigation actions can be inserted with a potential impact on the WCET. For instance, Phase Change Memory (PCM) require high voltages for correct operation, provided by charge pumps, which accelerates aging. Discharging the stressed charge pump lowers the aging rate, but makes the neuromorphic hardware unavailable to perform computations during the discharging period [19]. Such operation unavailability is expected to impact the WCET estimations.

Emerging memory devises can be used for storing and processing of the data. Therefore, different timing delays are expected, depending on how data are using the memory. Furthermore, as applications typically have more complex logic functions than bit-wise operations, code-snippets will be offloaded on the emerging memory devices for efficient computation. As a result, the WCET will require to be estimated with models that are able to express parameters, such as the size and data computation of the offloaded code-snippet and the characteristics of memory-centric computer architectures. We believe that combining hybrid WCET approaches with parametric expressions can lead to WCET estimations for code-snippet offloading at emerging memory devises. Such methods will be required in order to provide timing guarantees during system deployment.

Furthermore, the reliability of emerging technologies is expected to be more dependent on the workload and the operating conditions, compared to CMOS technology. For instance, STT-RAM have high current densities that can lead to electromagnetic failures to the signal lines leading to asymmetric reliability [172] and RRAM variability can be affected by operating conditions, such as temperature and voltage [49]. We will explore methodologies to incorporate such reliability characteristics in order to perform timing and functional vulnerability analysis.

Last, but not least, sharing emerging memory devices among cores and tasks will open-up further challenges to be solved regarding timing guarantees, WCET aspects and reliability. For instance, exploration is needed regarding whether the memory banks can be shared or isolated, how this can be implemented and what are the impacts on WCET estimations.

Design Space Exploration for future heterogeneous architectures

Efficient DSE approaches will be required to efficiently design and use future heterogeneous architectures, consisting of multiple computing elements and different emerging technologies and architectures, while guaranteeing timing requirements and reliable execution.

Analysing the application in order to determine which part of an application is more efficient to be executed on a specific PIM architecture unit is not straightforward, due to the variety of technologies, architectures, different operations, available configurations and code complexity. We will focus on designing DSE approaches under real-time constraints, taking into account the different operations supported by different memory-centric computer architectures, the size, operations and regularity of instructions and code-snippets, the impact on the WCET estimations, and the interference. As a first step, we will design a methodology to characterise candidate code-snippets for offloading based on cost functions. Relevant cost functions and appropriate metrics will be defined, related to operation complexity, operation frequency, potential memory bandwidth savings, WCET estimation and interference impact. Then, an overall characterisation approach for the offloading candidates is required, taking into account the different granularities and sizes, regarding the eligibility, performance, energy consumption and precision of different memory units that perform processing. Last, we will design of methodologies to efficiently select the most beneficial offloading candidates and PIM architectures, taking into account dependencies and real-time constraints.

Furthermore, we will extend the proposed approaches to take into account the reliability aspects. The endurance of emerging memory technologies is the number of switching cycles a device can perform until it breaks down. It is related with data retention, i.e., the capability of retaining the information stored even when the power has been switched off. Endurance, and thus, retention, depend on the workload of the emerging memory device. Novel approaches will extended with the variability of memory-centric computer architectures and model how code-snippet offloading decisions will affect the lifetime of PIM devises. To take advantage of the emerging computing paradigm of PIM, efficient fault-detection and effective fault-tolerance techniques need to be explored across different abstraction levels [202].

Last, but not least, by adding computation capabilities in memory, there is a need of an in-memory scheduler able to deal with sharing of the memory among cores and tasks. Since the memory becomes a partial computation unit, a PIM scheduler will be required to deal with concurrent requests on the device. In this context, exploration is required to decide whether the execution of a code-snippet on a PIM device can be preempted or not, since such decision will impact the interference and WCET estimations, and how such preemptive/non-preemptive mechanism can be implemented.

5.5 Conclusions

The future directions of this research focus on providing the means to design, in a near-optimal and efficient way, real-time and reliable embedded systems for safety-critical domains, with unreliable components, under multiple reliability threats.

Our first research direction is the analysis of the timing impact of transient faults occurring on cores by leveraging our reliability analysis framework to incorporate the application timing behaviour and to include the impact of interference. WCET estimations will be enhanced with fault awareness and low cost fault tolerance techniques will be designed to protect the system. Our second direction is the design hardware accelerators for WCET-aware and fault-aware systems to extend homogeneous multicore systems towards domain specific heterogeneous multicore architectures. To achieve that, we will adapt reliability and WCET analysis frameworks and design real-time and fault tolerance techniques for hardware accelerators, such as accelerators dedicated for AI. With the further ongoing reduction of transistors size, system aging is becoming more and more sensitive to the workload. Thus, we will focus on dedicated cross-layer fault-aware and WCET-aware approaches to efficiently deal with workload-dependent aging faults for safety-critical systems. Furthermore, the systems are susceptible to multiple types of reliability threats, potentially correlated with each other. Therefore, we will leverage the proposed approaches to consider multiple sources for reliability threats. Last, but not least, we will propose novel approaches to provide timing and reliability analysis and DSE for new emerging technologies and novel computing paradigms to be used in safety-critical domains with real-time and reliable guarantees.

References

Author publications

[B2] A. Kritikakou, F. Catthoor and C. Goutis, "Scalable and near-optimal design space exploration for embedded systems", Springer, 2014

[J26] R. Psiakis, A. Kritikakou and O. Sentieys, "Cluster-Based Dynamic Fault-Tolerant VLIW Processor with Heterogeneous Function Units", Elsevier Microprocessors and Microsystems (MICPRO), 2022

[J25] M. Cui, A. Kritikakou, L. Mo, and E. Casseau, "Near-optimal Energy-Efficient Partial-Duplication Mapping of Real-Time Parallel Applications ", Elsevier Journal of System Architecture (JSA), 2022 (Minor revision)

[J24] M. Cui, A. Kritikakou, L. Mo, and E. Casseau, "Energy-aware Partial-Duplication Task Mapping under Real-Time and Reliability Constraints for multiple DVFS schemes", Springer International Journal of Parallel Programming (IJPP), 2022

[J23] R. Mercier, C. Killien, A. Kritikakou, Y. Helen, and D. Chillet, "BiSuT: A NoC-Based Bit-Shuffling Technique for Multiple Permanent Faults Mitigation", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (IEEE TCAD), 2021

[J21] A.Kritikakou, R. Psiakis, F. Catthoor, and O. Sentieys, "Binary Tree Classification of Rigid Error Detection and Correction Techniques" Journal of ACM Computed Surveys (ACM CS), 2020

[J16] L. Mo, A. Kritikakou and O. Sentieys, "Controllable QoS for Imprecise Computation Tasks on DVFS Multicores with Time and Energy Constraints", IEEE Journal on Emerging and Selected Topics in Circuits and Systems (IEEE JETCAS), 2018

[J15] L. Mo, A. Kritikakou and O. Sentieys, "Energy-Quality-Time Optimized Task Mapping for DVFS-enabled Real-Time Multicore", IEEE Transaction on Computer Aided Design (IEEE TCAD), 2018

[J14] A. Kritikakou, T. Marty and M. Roy, "DYNASCORE: DYNAmic Software COntroller to Increase REsource Utilization in Mixed-Critical Systems", ACM Trans. Design Automation of Electronic Systems (ACM TODAES), Vol. 23, Issue 2, Jan., 2018

[J6] A.Kritikakou, F. Catthoor, G.S. Athanasiou, V. Kelefouras and C. Goutis, "Near-optimal Microprocessor & Accelerators Co-Design with Latency & Throughput Constraints", ACM Trans. Architecture and Code Optimization (ACM TACO), Vol.10, No.2, May, 2013

[C26] A. Kritikakou, O. Sentieys, G. Hubert, Y. Helen, J.F. Coulon, and P. Deroux-Dauphin, "FLODAM: Cross-layer fault-tolerant reliability analysis flow for complex hardware designs", submitted in Design, Automation & Test in Europe (DATE), 2022

[C25] L. Mo, Q. Zhou, A. Kritikakou, and J. Liu, "Energy Efficient, Real-time and Reliable Task Deployment on NoC-based Multicores with DVFS", Design, Automation & Test in Europe (DATE), 2022

[C24] R. Mercier, C. Killian, A. Kritikakou, Y. Helen and D. Chillet, "A Region-Based Bit-Shuffling Approach Trading Hardware Cost and Fault Mitigation Efficiency", IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2021, Virtual conference

[C23] M. Cui, A. Kritikakou, L. Mo, and E. Casseau, "Fault-Tolerant Mapping of Real-Time Parallel Applications under multiple DVFS schemes", IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS),18-21 May, 2021, Virtual conference

[C22] R. Mercier, C. Killien, A. Kritikakou, Y. Helen, and D. Chillet, "Multiple Permanent Faults Mitigation through Bit-Shuffling for Network-on-Chip Architecture", International Conference on Computer Design (ICCD), 2020

[C21] S. Skalistis and A. Kritikakou, "Dynamic interference-sensitive run-time adaptation of Time-Triggered schedules", Euromicro Conference on Real-Time Systems (ECRTS), 2020

[C20] J. Paturel, A. Kritikakou, and O. Sentieys, "Fast Cross-Layer Vulnerability Analysis of Complex Hardware Designs", IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 6-8 July, 2020, Limassol, Cyprus

[C19] M. Cui, L. Mo, A. Kritikakou, and E. Casseau, "Energy-aware Partial-Duplication Task Mapping under Real-Time and Reliability Constraints", International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 5-9 July, 2020, Samos, Greece

[C18] S. Skalistis and A. Kritikakou, "Timely Fine-grained interference-sensitive run-time adaptation of time-triggered schedules", IEEE Real-Time Systems Symposium (RTSS), 2019

[C17] R. Psiakis, A. Kritikakou and O. Sentieys, "Fine-grained Hardware Mitigation for Multiple Long-Duration Transients on VLIW Function Units", Design, Automation & Test in Europe (DATE), 25-29 March, 2019, Florence, Italy

[C16] L. Mo, A. Kritikakou and O. Sentieys, "Approximation-aware Task Deployment on Asymmetric Multicore Processors", Design, Automation & Test in Europe (DATE), 25-29 March, 2019, Florence, Italy

[C15] R.Psiakis, A. Kritikakou, E. Casseau and O. Sentieys, "Run-Time Coarse-Grained Hardware Mitigation for Multiple Faults on VLIW Processors", Conference on Design and Architectures for Signal and Image Processing (DASIP), 16-18 October, 2019, Montreal, Canada

[C13] S. Derrien, I. Puaut, P. Alefragis, M. Bednaraz, H. Bucherx, C. David, Y. Debray, U. Durak,I. Fassi, C. Ferdinand, D. Hardy, A. Kritikakou, G. Rauwerda, S. Reder, M. Sicks, T. Stripf, K. Sunesen, T. Braak, N. Voros, J. Becker, "WCET-Aware Parallelization of Model-Based Applications for Multi-Cores: the ARGO Approach", Design, Automation & Test in Europe (DATE), 27-31 March 2017, Lausanne, Swiss

[C12] R.Psiakis, A. Kritikakou, and O. Sentieys, "NEDA: NOP Exploitation with Dependency Awareness for Reliable VLIW Processors", Computer Society Annual Symposium on VLSI (ISVLSI), July 3-5, 2017, Bochum, Germany

[C11] R.Psiakis, A. Kritikakou, and O. Sentieys, "Run-Time Instruction Replication for Permanent and Soft Error Mitigation in VLIW Processors", International New Circuits and Systems Conference (NEWCAS), 25-28 June, 2017 Strasbourg, France

[C10] L. Mo, A. Kritikakou and O. Sentieys, "Decomposed Task Mapping to Maximize QoS in Energy-Constrained Real-Time Multicores", International Conference on Computer Design (ICCD), 5-8 November, 2017, Massachusetts, USA

[C9] A. Kritikakou, T. Marty, C. Pagetti, C. Rochange, Michael Lauer and M. Roy, "Multiplexing Adaptive with Classic AUTOSAR? Adaptive Software Control to Increase Resource Utilization in Mixed-Critical Systems", Critical Automotive applications: Robustness & Safety (CARS), Sep 2016, Göteborg, Sweden

[C8] A. Kritikakou, O. Baldellon, C. Pagetti, C. Rochange and M. Roy, "Run-time Control to Increase Task Parallelism in Mixed-Critical Workloads", Euromicro Conference on Real-Time Systems (ECRTS), 8-11 July 2014, Madrid, Spain

[C7] A. Kritikakou, C. Pagetti, C. Rochange, M. Roy, Madeleine Faugère, Sylvain Girbal and Daniel Gracia Pérez, "Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems", International Conference on Real-Time Networks and Systems (RTNS), 8-10 October, 2014, Versailles, France

[W2] A. Kritikakou, and S. Skalistis, "Progress-aware dynamic slack exploitation in mixed-critical systems", WiP in Proc. International Conference on Embedded Software (EMSOFT), 20-25 Sep 2020, Virtual Conference

[W1] A. Kritikakou, O. Baldellon, C. Pagetti, C. Rochange, M. Roy, and F. Vargas, "Monitoring On-line Timing Information to Support Mixed-Critical Workloads", WiP in Proc. International Conference Real-Time Systems Symposium (RTSS), 3-6 Dec 2013, Vancouver, Canada

[PP7] M. Cui, A. Kritikakou, L. Mo, and E. Casseau, "Near-optimal Energy-Efficient Partial-Duplication Mapping of Real-Time Parallel Applications ", Presentation (2.0 publication model) in Int. Conf. Reliable Software Technologies (AEiC 2022), 2022, Ghent, Belgium

[PP5] R. Mercier, C. Killian, A. Kritikakou, Y. Helen and D. Chillet, "Tolerating Errors in NoC: A Lightweight Region-Based Fault-Mitigation Method", Presentation in Silicon Errors in Logic – System Effects (SELSE), 2022

Other publications

[1] Space Product Assurance: Techniques for Radiation Effects Mitigation in AASIC and FPGAs Handbook. Technical report, ESA Requirements and Standards Division, Sept. 2016.

[2] S. Abd Ishak, H. Wu, and U. Tariq. Energy-aware task scheduling on heterogeneous NoC-based MPSoCs. In *Int. Conf. Computer Design (ICCD)*, pages 165–168, 2017.

[3] J. Abella, F. Cazorla, E. Quiñones, et al. Towards improved survivability in safety-critical systems. In *Int. Symp. On-Line Testing and Robust System (IOLTS)*, pages 240–245, 07 2011.

[4] J. Abella, C. Hernandez, Ed. Quiñones, F.J. Cazorla, P.R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega. Wcet analysis methods: Pitfalls and challenges on their trustworthiness. In *Int. Symp. Industrial Embedded Systems (SIES)*, pages 1–10, 2015.

[5] J. Abella, M. Padilla, J. Del Castillo, and F.J. Cazorla. Measurement-based worst-case execution time estimation using the coefficient of variation. *ACM Trans. Design Automation of Electronic Systems (TODAES)*, 22, June 2017.

[6] J. Abella, E. Quiñones, F. Cazorla, et al. Rvc: A mechanism for time-analyzable real-time processors with faulty caches. pages 97–106, 01 2011.

[7] A. B. Ahmed, D. Fujiki, H. Matsutani, M. Koibuchi, and H. Amano. AxNoC: Low-power Approximate Network-on-chips Using Critical-path Isolation. In *Int. Symp. Networks-on-Chip (NOCS)*, number 6, pages 1–8. IEEE, Oct. 2018.

[8] S. Ainsworth and T.M. Jones. Parallel error detection using heterogeneous cores. In *Int. Conf. Dependable Systems and Networks (DSN)*, pages 338–349, 2018.

[9] Z. Al-bayati, J. Caplan, B. H. Meyer, and H. Zeng. A four-mode model for efficient fault-tolerant mixed-criticality systems. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 97–102, 2016.

[10] H. Amrouch, V.M. van Santen, T. Ebi, V. Wenzel, and J.g Henkel. Towards interdependencies of aging mechanisms. In *Int. Conf. Computer-Aided Design (ICCAD)*, pages 478–485, 2014.

[11] H. Amrouch, V.M. van Santen, and J. Henkel. Interdependencies of degradation effects and their impact on computing. *IEEE Design & Test*, 34(3):59–67, 2017.

[12] J. Anderson, S. Baruah, and B. Brandenburg. Multicore Operating-System Support for Mixed Criticality. In *Int. Workshop Mixed Criticality Systems (WMC)*, April 2009.

[13] Ma. Anwar, S. Furqan Qadri, and A. Sattar. Green computing and energy consumption issues in the modern age. *IOSR Journal of Computer Engineering*, 12:91–98, 01 2013.

[14] K. Asifuzzaman, M. Fernandez, P. Radojković, J. Abella, and F.J. Cazorla. Stt-mram for real-time embedded systems: Performance and wcet implications. In *Int. Symp. Memory Systems (MEMSYS)*, page 195–205, New York, NY, USA, 2019. Association for Computing Machinery.

[15] O. Astrachan. Bubble Sort: An Archaeological Algorithmic Analysis. In *Proc. ACM Tech. Symp. on Comput. Sci. Educ. (SIGCSE)*, SIGCSE'03, pages 1–5, New York, NY, USA, 2003. Association for Computing Machinery.

[16] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Real-Time Systems Symp. (RTSS)*, pages 95–105, 2001.

[17] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Optimal reward-based scheduling for periodic real-time tasks. *IEEE Trans. Computers (TC)*, 50(2):111–130, 2001.

[18] J.R. Azambuja, S. Pagliarini, M. Altieri, et al. A fault tolerant approach to detect transient faults in microprocessors based on a non-intrusive reconfigurable hardware. *Trans. on Nuclear Science (TNS)*, 59(4):1117–1124, 2012.

[19] A. Balaji, S. Song, A. Das, N. Dutt, J. Krichmar, N. Kandasamy, and F. Catthoor. A framework to explore workload-specific performance and lifetime trade-offs in neuromorphic computing. *IEEE Computer Architecture Letters*, 18(2):149–152, 2019.

[20] M. Baleani, A. Ferrari, L. Mangeruca, M. Peri, and S. Pezzini. Fault-Tolerant Platforms for Automotive Safety-Critical. In *Int. Conf. Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 170–177, May 2004.

[21] B. Barrois, K. Parashar, and O. Sentieys. Leveraging power spectral density for scalable system-level accuracy evaluation. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, page 6, Dresden, Germany, March 2016.

[22] S. Baruah, V. Bonifaci, G. D'Angelo, et al. Mixed-criticality scheduling of sporadic task systems. In *Annual European Symposium on Algorithms (ESA)*, 2011.

[23] S. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *Real-Time Systems Symp. (RTSS)*, 2011.

[24] S. Baruah, A. Burns, and Z. Guo. Scheduling mixed-criticality systems to guarantee some service under all non-erroneous behaviors. In *Euromicro Conf. Real-Time Systems (ECRTS)*, pages 131–138, 2016.

[25] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems (RTS)*, 50(1):142–177, January 2014.

[26] I. Bate, A. Burns, and R. I. Davis. A bailout protocol for mixed criticality systems. In *Euromicro Conf. Real-Time Systems (ECRTS)*, pages 259–268, 2015.

[27] R. Baumann. Soft errors in advanced computer systems. *IEEE Design Test of Computers*, 22(3):258–266, 2005.

[28] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numer. Math.*, 4(1):238–252, 1962.

[29] M.K. Bepary, B.M.S.B. Talukder, and M.T. Rahman. Dram retention behavior with accelerated aging in commercial chips. *Applied Sciences*, 12(9), 2022.

[30] M.l Beyer, A. Morozov, K. Ding, S. Ding, and K. Janschek. Quantification of the impact of random hardware faults on safety-critical ai applications: Cnn-based traffic sign recognition case study. In *Int. Symp. Software Reliability Engineering Workshops (ISSREW)*, pages 118–119, 2019.

[31] P.V. Bhanu, P.V. Kulkarni, and J. Soumya. Fault-tolerant network-on-chip design with flexible spare core placement. *ACM J. Emerg. Technol. Comput. Syst.*, 15(1), 2019.

[32] A. Bhat, S.l Samii, and R. Rajkumar. Practical task allocation for software fault-tolerance and its implementation in embedded automotive systems. In *Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 87–98, 2017.

[33] B. Bhowmik, S. Biswas, J. K. Deka, and B. B. Bhattacharya. A Low-Cost Test Solution for Reliable Communication in Networks-on-Chip. *J. of Electron. Testing*, 35(2):215–243, Apr. 2019.

[34] P. Bieber, F. Boniol, M. Boyer, E. Noulard, and C. Pagetti. New challenges for future avionic architectures. *Embedded Systems: Handbook*, pages 1–10, January 2012.

[35] J. Bin, S. Girbal, D. Gracia Perez, A. Grasset, and A. Merigot. Studying co-running avionic real-time applications on multi-core cots architectures. In *European Congress Embedded Real-Time Systems (ERTS)*, pages 1–10, February 2014.

[36] C. Bolchini. A software methodology for detecting hardware faults in VLIW data paths. *IEEE Trans. Reliability (TR)*, 52(4):458–468, December 2003.

[37] A. Bosio, P. Bernardi, A. Ruospo, and E. Sanchez. A reliability analysis of a deep neural network. In *Latin American Test Symp. (LATS)*, pages 1–6, 2019.

[38] R. Bouziane, E. Rohou, and A. Gamatié. Energy-Efficient Memory Mappings based on Partial WCET Analysis and Multi-Retention Time STT-RAM. In *RTNS: Real-Time Networks and Systems*, pages 148–158, Poitiers, France, October 2018.

[39] S. Boyd, A. Ghosh, and A. Magnani. Branch and bound methods. *Notes for EE364b, Stanford University*, pages 1–11, 2007.

[40] A. Burns and B. Baruah. Towards a more practical model for mixed criticality systems. In *Real-Time Systems Symp. (RTSS)*, 2013.

[41] A. Burns and S. Baruah. Timing faults and mixed criticality systems. In *Dependable and Historic Computing*, LNCS. Springer, 2011.

[42] R.A. Camponogara-Viera, R. Possamai Bastos, J.-M. Dutertre, O. Potin, M.-L. Flottes, G. Di Natale, and B. Rouzeyre. Validation Of Single BBICS Architecture In Detecting Multiple Faults. In *Asian Test Symposium (ATS)*, Mumbai, India, November 2015.

[43] F. Catthoor and G. Groeseneken. Will Chips of the Future Learn How to Feel Pain and Cure Themselves? *IEEE Design Test*, 34(5):80–87, October 2017.

[44] P. Li Cavoli, G. Hubert, and J. Busto. Study of atmospheric muon interactions in si nanoscale devices. 12(12):P12021–P12021, dec 2017.

[45] F.J. Cazorla, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and T. Vardanega. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Computing Surveys (CS).*, 52(1), February 2019.

[46] J.P. Cerrolaza, R. Obermaisser, J. Abella, F.J. Cazorla, K. Grüttner, I. Agirre, H. Ahmadian, and I. Allende. Multi-core devices for safety-critical systems: A survey. *ACM Computing Surveys (CS)*, 53(4), aug 2020.

[47] C.K. Chang, S. Lym, N. Kelly, et al. Hamartia: A fast and accurate error injection framework. In *Int. Conf. on Dependable Systems and Networks Workshops (DSN-W)*, pages 101–108, 2018.

[48] N. Chatterjee, S. Paul, and S. Chattopadhyay. Fault-tolerant dynamic task mapping and scheduling for network-on-chip-based multicore platform. *ACM Trans. Embed. Comput. Syst.*, 16(4), 2017.

[49] A. Chen and M.-R. Lin. Variability of resistive switching memories and its impact on crossbar array performance. In *Int. Reliability Physics Symposium*, pages MY.7.1–MY.7.4, 2011.

[50] C. Chen, J Panerati, I. Hafnaoui, and G. Beltrame. Static probabilistic timing analysis with a permanent fault detection mechanism. In *Int. Symp. Industrial Embedded Systems (SIES)*, pages 1–10, 2017.

[51] C. Chen, J. Panerati, M. Li, and G. Beltrame. Probabilistic timing analysis of timed-randomised caches with fault detection mechanisms. *IET Computers and Digital Techniques*, 13, 01 2019.

[52] C. Chen, L. Santinelli, J Hugues, and G. Beltrame. Static probabilistic timing analysis in presence of faults. In *Int. Symp. Industrial Embedded Systems (SIES)*, pages 1–10, 2016.

[53] G. Chen, N. Guan, K. Huang, and W. Yi. Fault-tolerant real-time tasks scheduling with dynamic fault handling. *Journal of Systems Architecture (JSA)*, 102:101688, 2020.

[54] G. Chen, K. Huang, and Al. Knoll. Energy optimization for real-time multiprocessor system-on-chip with optimal DVFS and DPM combination. *ACM Trans. Embedded Computing Systems (TECS)*, 13(3):111:1–111:21, 2014.

[55] T. Chen, Q. Guo, K. Tang, O. Temam, Z. Xu, Z. Zhou, and Y. Chen. ArchRanker: A ranking approach to design space exploration. In *Int. Symp. Computer Architecture (ISCA)*, pages 85–96, June 2014.

[56] X. Chen, Z. Lu, Y. Lei, Y. Wang, and S. Chen. Multi-Bit Transient Fault Control For NoC Links Using 2D Fault Coding Method. In *Int. Symp. Networks-on-Chip (NOCS)*, pages 1–8. IEEE/ACM, Aug. 2016.

[57] Y. Chen, I. Bayram, and E. Eken. Recent technology advances of emerging memories. *IEEE Design & Test*, PP:1–1, 03 2017.

[58] Y.-G. Chen, I.-C. Lin, and J.-T. Ke. Road: Improving reliability of multi-core system via asymmetric aging. In *Int. Conf. Computer-Aided Design (ICCAD)*, pages 1–8, 2019.

[59] Y.-Y Chen et al. An integrated fault-tolerant design framework for vliw processors. In *Int'l Symp. Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 555–562, Nov 2003.

[60] Y.-Y. Chen and K.-L. Leu. Reliable data path design of VLIW processor cores with comprehensive error-coverage assessment. *Microprocessors and Microsystems: Embedded Hardware Design*, 34(1):49 – 61, 2010.

[61] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben. Binfi: An efficient fault injector for safety-critical machine learning systems. In *Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, New York, NY, USA, 2019. Association for Computing Machinery.

[62] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben. Tensorfi: A flexible fault injection framework for tensorflow applications. In *Int. Symp. on Software Reliability Engineering (ISSRE)*, pages 426–435, 2020.

[63] Z. Chen, Y. Zhang, Z. Peng, and J. Jiang. A Deterministic-Path Routing Algorithm for Tolerating Many Faults on Wafer-Level NoC. In *Des. Automat. Test in Europe Conf. Exhib. (DATE)*, pages 1337–1342, Mar. 2019.

[64] S. Chichin, M. Brundler, D. Portes, and V. Jegu. Capability to embed deep neural networks: Study on cpu processor in avionics context. In *Embedded Real-Time Systems (ERTS)*, 2020.

[65] H. Chishiro and N. Yamasaki. Practical imprecise computation model: Theory and practice. In *Int. Symp. Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 198–205, 2014.

[66] L. A. Cortes, P. Eles, and Z. Peng. Quasi-static assignment of voltages and optional cycles in imprecise-computation systems with energy considerations. *IEEE Trans. Very Large Scale Integr. Syst. (TVLSI)*, 14(10):1117–1129, 2006.

[67] L. Cucu-Grosjean. Probabilistic Approaches for Time Critical Embedded Systems. In *Int. Works. Verification and Evaluation of Computer and Communication Systems (VECoS)*, September 2015.

[68] D. d. Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Real-Time Systems Symp. (RTSS)*, 2009.

[69] T. Dalgaty, N. Castellani, D. Querlioz, and E. Vianello. In-situ learning harnessing intrinsic resistive memory variability through markov chain monte carlo sampling. *CoRR*, abs/2001.11426, 2020.

[70] W.J. Dally and B.P. Towles. *Principles and Practices of Interconnection Networks*. Elsevier, Mar. 2004.

[71] A. Das, A. Kumar, and B. Veeravalli. A Survey of Lifetime Reliability-Aware System-Level Design Techniques for Embedded Multiprocessor Systems, 2014.

[72] R. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CS)*, 43(4), oct 2011.

[73] R. Davis and L. Cucu-Grosjean. A survey of probabilistic schedulability analysis techniques for real-time systems. 6:04:1–04:53, 05 2019.

[74] I. De Albuquerque Silva, A. Carle, T.sand Gauffriau, and C. Pagetti. ACETONE: Predictable Programming Framework for ML Applications in Safety-Critical Systems. In *Euromicro Conf. Real-Time Systems (ECRTS)*, volume 231 of *Leibniz Int. Proceedings in Informatics (LIPIcs)*, pages 3:1–3:19, 2022.

[75] D. de Niz and L. T. X. Phan. Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms. In *Real-time and embedded Technology and Applications Symp. (RTAS)*, 2014.

[76] A. DeOrio, D. Fick, V. Bertacco, D. Sylvester, D. Blaauw, J. Hu, and G. Chen. A Reliable Routing Architecture and Algorithm for NoCs. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 31(5):726–739, May 2012.

[77] B. Deveautour, A. Virazel, P. Girard, S. Pravossoudovitch, and V. Gherman. Is aproximate computing suitable for selective hardening of arithmetic circuits? In *Int. Conf. Design Technology of Integrated Systems In Nanoscale Era (DTIS)*, pages 1–6, 2018.

[78] J.-F. Deverge and I. Puaut. Safe measurement-based WCET estimation. In Reinhard Wilhelm, editor, *Int. Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 1 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2007. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[79] S. Di Mascio, A. Menicucci, E. Gill, G. Furano, and C. Monteleone. Leveraging the openness and modularity of risc-v in space. *Journal of Aerospace Information Systems*, 16(11):454–472, 2019.

[80] A. Dixit and A. Wood. The impact of new technology on soft error rates. In *Int. Reliability Physics Symp. (IRPS)*, pages 5B.4.1–5B.4.7, April 2011.

[81] E. Dubrova. Fault tolerant design: an introduction. In *Springer*, 2008.

[82] E. Dubrova. *Fault-Tolerant Design*. Fault-Tolerant Design. Springer, 2013.

[83] M. Ebrahimi, M. Daneshtalab, J. Plosila, and H. Tenhunen. Minimal-Path Fault-Tolerant Approach Using Connection-Retaining Structure in Networks-on-Chip. In *Int. Symp. Networks-on-Chip (NOCS)*, pages 1–8. IEEE/ACM, Apr. 2013.

[84] A. Esper, G. Nelissen, V. Nélis, and E. Tovar. An industrial view on the common academic understanding of mixed-criticality systems. *Real-Time Systems (RTD)*, 54(3):745–795, 2018.

[85] M. Fischetti, F. Glover, and A. Lodi. The feasibility pump. *Math. Program.*, 104(1):91–104, 2005.

[86] T. Fleming and A. Burns. Extending mixed criticality scheduling. In *Real-Time Systems Symp. (RTSS)*, 2013.

[87] X. Fong, Y. Kim, K. Yogendra, D. Fan, A. Sengupta, A. Raghunathan, and K. Roy. Spin-transfer torque devices for logic and memory applications : Prospects and perspectives. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 35:1–1, 01 2015.

[88] B. Fu, Y. Han, H. Li, and X. Li. ZoneDefense: A Fault-Tolerant Routing for 2-D Meshes Without Virtual Channels. *IEEE Trans. Very Large Scale Integration (TVLSI) Systems*, 22(1):113–126, Jan. 2014.

[89] F. Gabbay and A. Mendelson. Asymmetric aging effect on modern microprocessors. *Microelectronics Reliability*, 119:114090, 2021.

[90] Y. Gao, L. Han, J. Liu, et al. Minimizing energy consumption for real-time tasks on heterogeneous platforms under deadline and reliability constraints. Research Report RR-9403, Inria - Research Centre Grenoble – Rhône-Alpes, 2021.

[91] P. Gargini. Roadmap Past, Present and Future.

[92] A. Gebregiorgis, H.A. Du Nguyen, J. Yu, R. Bishnoi, M. Taouil, F. Catthoor, and S. Hamdioui. A survey on memory-centric computer architectures. *J. Emerg. Technol. Comput. Syst.*, jun 2022. Just Accepted.

[93] N.J. George, C. Elks, B. Johnson, et al. Transient fault models and avf estimation revisited. In *Int. Conf. Dependable Systems and Networks (DSN)*, pages 477 – 486, 08 2010.

[94] D. Gizopoulos, M. Psarakis, S. V. Adve, P. Ramachandran, S. K. S. Hari, D. Sorin, A. Meixner, A. Biswas, and X. Vera. Architectures for online error detection and recovery in multicore processors. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 1–6, March 2011.

[95] C. Gou, A. Benoit, M. Chen, et al. Reliability-aware energy optimization for throughput-constrained applications on MPSoC. In *Int. Conf. Parallel and Distributed Systems (ICPADS)*, pages 1–10, 2018.

[96] Y. Guo, D. Zhu, and H. Aydin. Reliability-aware power management for parallel real-time applications with precedence constraints. In *Int. Green Computing Conf. and Workshops (ICGSET)*, pages 1–8, 2011.

[97] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R.d Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Int. Workshop on Workload Characterization (IWWC)*, pages 3–14, 01 2002.

[98] S. Hamdioui, D. Gizopoulos, G. Guido, M. Nicolaidis, A. Grasset, and P. Bonnot. Reliability challenges of real-time systems in forthcoming technology nodes. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 129–134, March 2013.

[99] S. Hamdioui, P. Pouyan, H. Li, Y. Wang, A. Raychowdhur, and I. Yoon. Test and reliability of emerging non-volatile memories. In *Asian Test Symposium (ATS)*, pages 175–183, 2017.

[100] J. Han, M. Lin, D. Zhu, and L.T. Yang. Contention-aware energy management scheme for NoC-based multicore real-time systems. *IEEE Trans. Parallel Distrib. Syst.*, 26(3):691–701, 2015.

[101] L. Han, L. C. Canon, J. Liu, et al. Improved energy-aware strategies for periodic real-time tasks under reliability constraints. In *Real-Time Systems Symp. (RTSS)*, pages 17–29, 2019.

[102] M.A Hanif and M. Shafique. Dnn-life: An energy-efficient aging mitigation framework for improving the lifetime of on-chip weight memories in deep neural network hardware architectures. In *Design, Automation and Test in Europe Conf. Exhibition (DATE)*, pages 729–734, 2021.

[103] M. A. Haque, H. Aydin, and D. Zhu. On reliability management of energy-aware real-time systems through task replication. *Trans. on Parallel and Distributed Systems (TPDS)*, 28(3):813–825, 2017.

[104] D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *Real-Time Systems Symp. (RTSS)*, pages 456–466, 2008.

[105] D. Hardy and I. Puaut. Static probabilistic worst case execution time estimation for architectures with faulty instruction caches. *Real-Time Systems (RTS)*, 51(2):128–152, March 2015.

[106] D. Hardy, I. Puaut, and Y. Sazeides. Probabilistic WCET estimation in presence of hardware for mitigating the impact of permanent faults. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 91–96, March 2016.

[107] O. He, S. Dong, W. Jang, J. Bian, and D.Z. Pan. UNISM: Unified scheduling and mapping for general networks on chip. *IEEE Trans. Very Large Scale Integr. Syst.*, 20(8):1496–1509, 2012.

[108] Y. He, P. Balaprakash, and Y. Li. Fidelity: Efficient resilience analysis framework for deep learning accelerators. In *Int. Symp. Microarchitecture (MICRO)*, pages 270–281. IEEE, 2020.

[109] J. Henkel, L. Bauer, H. Zhang, S. Rehman, and M. Shafique. Multi-layer dependability: From microarchitecture to application level. In *Design Automation Conference (DAC)*, pages 1–6, June 2014.

[110] J.L. Hennessy and D.A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, jan 2019.

[111] B. Hu, K. Huang, P. Huang, et al. On-the-fly fast overrun budgeting for mixed-criticality systems. In *Int. Conf. Embedded Software (EMSOFT)*, pages 1–10, 10 2016.

[112] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. Irwin. Compiler-assisted Soft Error Detection Under Performance and Energy Constraints in Embedded Systems. *ACM Trans. Embedded Computing Systems (TECS)*, 8(4):27:1–27:30, July 2009.

[113] J. Huang, J. O. Blech, A. Raabe, C. Buckl, and A. Knoll. Reliability-Aware Design Optimization for Multiprocessor Embedded Systems. In *Euromicro Conf. Digital System Design (DSD)*, pages 239–246, August 2011.

[114] K. Huang, X. Jiang, X. Zhang, et al. Energy-efficient fault-tolerant mapping and scheduling on heterogeneous multiprocessor real-time systems. *IEEE Access*, 6:57614–57630, 2018.

[115] P. Huang, H. Yang, and L. Thiele. On the scheduling of fault-tolerant mixed-criticality systems. In *Design Automation Conference (DAC)*, pages 1–6, 2014.

[116] W. H. Huang and J. J. Chen. Self-suspension real-time tasks under fixed-relative-deadline fixed-priority scheduling. In *Design, Automation & Test in Europe (DATE)*, pages 1078–1083, 2016.

[117] G. Hubert et al. A generic platform for remote accelerated tests and high altitude SEU experiments on advanced ICs: Correlation with MUSCA SEP3 calculations. In *Int. On-Line Testing Symposium (IOLTS)*, pages 180–180, June 2009.

[118] E. Ibe, H. Taniguchi, Y. Yahagi, et al. Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule. *IEEE Trans. on Electron Devices (TED)*, 57(7):1527–1538, 2010.

[119] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Scheduling of Fault-Tolerant Embedded Systems with Soft and Hard Timing Constraints. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 915–920, March 2008.

[120] J. Johnson, W. Howes, M.J. Wirthlin, et al. Using duplication with compare for on-line error detection in fpga-based designs. *EEE Aerospace Conference*, pages 1–11, 2008.

[121] V. Joshi, M. Le Gallo, S. Haefeli, et al. Accurate deep neural network inference using computational phase-change memory. *Nature Communications*, 11(1), may 2020.

[122] L.R. Juracy, M.T. Moreira, A.M. Amory, and F.G. Moraes. A survey of aging monitors and reconfiguration techniques. *arXiv preprint arXiv:2007.07829*, 2020.

[123] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, et al. Differential fault injection on microarchitectural simulators. In *Int. Symp. Workload Characterization (ISWC)*, pages 172–182, 2015.

[124] S. Kang, H. Yang, L. Schor, I. Bacivarov, S. Ha, and L. Thiele. Multi-objective mapping optimization via problem decomposition for many-core systems. In *Symposium on Embedded Systems for Real-time Multimedia (ESTMedia)*, pages 28–37, October 2012.

[125] K. Khalil, O. Eldash, A. Kumar, and M. Bayoumi. Self-Healing Hardware Systems: A Review. *Microelectronics Journal*, 93:104620, Nov. 2019.

[126] J. Kim, H. Kim, H. Amrouch, J. Henkel, A. Gerstlauer, and K. Choi. Aging Gracefully with Approximation. In *Int. Symp. Circuits and Systems (ISCAS)*, pages 1–5, May 2019.

[127] N. Kim, B. C. Ward, M. Chisholm, C. Fu, J. H. Anderson, and F. D. Smith. Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. In *Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 1–12, April 2016.

[128] J.S. Klecka, W.F. Bruckert, and R.L. Jardine. Error self-checking and recovery using lock-step processor pair architecture, 2002. U.S. Patent 6,393,582 B1.

[129] Y. Ko, S. Kim, H. Kim, and K. Lee. Selective code duplication for soft error protection on vliw architectures. *Electronics*, 10(15), 2021.

[130] D. Kraak, M. Taouil, S. Hamdioui, P. Weckx, F. Catthoor, A. Chatterjee, A. Singh, H. Wunderlich, and N. Karimi. Device aging: A reliability and security concern. In *European Test Symp. (ETS)*, pages 1–10, May 2018.

[131] A. J.R. Kumar. Statistical analysis of wcet estimation on dnns, 2018.

[132] S. Kundu and S. Chattopadhyay. *Network-on-Chip: The Next Generation of System-on-Chip Integration*. Taylor & Francis, 2014.

[133] N. Landeros Muñoz, A. Valero, R. Gran Tejero, and D. Zoni. Gated-cnn: Combating nbti and hci aging effects in on-chip activation memories of convolutional neural network accelerators. *Journal of Systems Architecture*, 128:102553, 2022.

[134] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[135] J. Lee, Y. Ko, K. Lee, J. Youn, and Y. Paek. Dynamic Code Duplication with Vulnerability Awareness for Soft Error Detection on VLIW Architectures. *ACM Trans. Architecture and Code Optimization (TACO)*, 9(4):48:1–48:24, January 2013.

[136] F. Lemonnier, P. Millet, G. M. Almeida, M. Hübner, J. Becker, S. Pillement, O. Sentieys, M. Koedam, S. Sinha, K. Goossens, C. Piguet, M. Morgan, and R. Lemaire. Towards future adaptive multiprocessor systems-on-chip: An innovative approach for flexible architectures. In *Int. Conf. Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 228–235, July 2012.

[137] D. Li, J.S. Vetter, and W. Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *Int. Conf on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2012.

[138] D. Li and J. Wu. Energy-aware scheduling for frame-based tasks on heterogeneous multiprocessor platforms. *Int. Conf. Parallel Processing (ICPP)*, 2012.

[139] D. Li and J. Wu. Minimizing energy consumption for frame-based tasks on heterogeneous multiprocessor platforms. *IEEE Trans. Parallel Distrib. Syst. (TPDS)*, 26(3):810–823, 2015.

[140] D. Li and J. Wu. Minimizing energy consumption for frame-based tasks on heterogeneous multiprocessor platforms. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 26(3):810–823, 2015.

[141] G. Li, S.K.S. Hari, M.l Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S.W. Keckler. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. In *Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, New York, NY, USA, 2017. Association for Computing Machinery.

[142] H. Li and S. Baruah. Global Mixed-Criticality Scheduling on Multiprocessors. In *Euromicro Conf. Real-Time Systems (ECRTS)*, pages 166–175, July 2012.

[143] H.-T. Li, C.-Y. Chou, Y.-T. Hsieh, W.-C. Chu, and A.-Y. Wu. Variation-aware reliable many-core system design by exploiting inherent core redundancy. *Trans. Very Large Scale Integration (VLSI) Systems*, 25(10):2803–2816, 2017.

[144] J. W. S. Liu, W. K. Shih, K. J. Lin, R. Bettati, and J. Y. Chung. Imprecise computations. *Proc. IEEE*, 82(1):83–94, 1994.

[145] M. Liu and B.H. Meyer. Bounding error detection latency in safety critical systems with enhanced execution fingerprinting. In *Int. Symp. Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 47–52, 2016.

[146] W. Liu and C.-H. Chang. Analysis of circuit aging on accuracy degradation of deep neural network accelerator. In *Int. Symp. Circuits and Systems (ISCAS)*, pages 1–5, 2019.

[147] G. Liva, L. Gaudio, T. Ninacs, and T. Jerkovits. Code Design for Short Blocks: A Survey. *Computing Research Repository (CoRR) - arXiv*, Oct. 2016.

[148] A. Lotfi, S. Hukerikar, K. Balasubramanian, P. Racunas, N. Saxena, R. Bramley, and Y. Huang. Resiliency of automotive object detection networks on gpu architectures. In *Int. Test Conf. (ITC)*, pages 1–9, 2019.

[149] A. Löfwenmark and S. Nadjm-Tehrani. Fault and timing analysis in critical multi-core systems: A survey with an avionics perspective. *Journal of Systems Architecture (JSA)*, 87:1–11, 2018.

[150] N. N. Mahatme, S. Jagannathan, T. D. Loveless, L. W. Massengill, B. L. Bhuva, S.-J. Wen, and R. Wong. Comparison of combinational and sequential error rates for a deep submicron process. *IEEE Trans. Nuclear Science (TNS)*, 58(6):2719–2725, 2011.

[151] C. Maiza, H. Rihani, J. Rivas, J. Goossens, S. Altmeyer, and R. Davis. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Computing Surveys (CS)*, 52(3):56:1–56:38, June 2019.

[152] R. Mancuso, R. Pellizzoni, N. Tokcan, and M. Caccamo. WCET Derivation under Single Core Equivalence with Explicit Memory Budget Assignment. In *Euromicro Conf. Real-Time Systems (ECRTS)*, volume 76, pages 3:1–3:23, 2017.

[153] S. Martinez, D. Hardy, and I. Puaut. Quantifying wcet reduction of parallel applications by introducing slack time to limit resource contention. In *Int. Conf. Real-Time Networks and Systems (RTNS)*, pages 188–197. ACM, 2017.

[154] P. Meloni, G. Tuveri, L. Raffo, E. Cannella, T. Stefanov, O. Derin, L. Fiorin, and M. Sami. System Adaptivity and Fault-Tolerance in NoC-based MPSoCs: The MADNESS Project Approach. In *Euromicro Conf. Digital System Design (DSD)*, pages 517–524, Sep. 2012.

[155] R. Mercier. *Multiple Fault Mitigation in Network-on-Chip Architectures Through A Bit-Shuffling Method.* Theses, Université de Rennes 1, December 2021.

[156] M. Micheletto, R. Santos, and J. Orozco. Using bioinspired meta-heuristics to solve reward-based energy-aware mandatory/optional real-time tasks scheduling. In *Brazilian Symposium on Computing System Engineering (SBESC)*, pages 132–135, 2015.

[157] T. Mithun Haridas, V. Ananthanarayanan, R. Naveen, and A. Rajeswari. Reliable and Affordable Embedded System Solution for Continuous Blood Glucose Maintaining System with Wireless Connectivity to Blood Glucose Measuring System. In *Amrita Int. Conf. Women in Computing (AICWIC)*, January 2013.

[158] K. Mitropoulou, V. Porpodas, and M. Cintra. CASTED: Core-Adaptive Software Transient Error Detection for Tightly Coupled Cores. In *Int. Symp. Parallel and Distributed Processing (ISPA)*, pages 513–524, May 2013.

[159] S. Mittal. A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Computing Surveys (CS).*, 48(3):45:1–45:38, 2016.

[160] H. J. Mohammed, W. N. Flayyih, and F. Z. Rokhani. Tolerating Permanent Faults in the Input Port of the Network on Chip Router. *Journal of Low Power Electronics and Applications*, 9(1):1–11, Feb. 2019.

[161] A.h Mokhtarpour, A.M. Monazzah, and H. Farbeh. Pb-ifmc: A selective soft error protection method based on instruction fault masking capability. pages 1–9, 01 2020.

[162] M. Mollison, J. Erickson, J. Anderson, S. Baruah, and J. Scoredos. Mixed-Criticality Real-Time Scheduling for Multicore Systems. In *Int. Conf. Computer and Information Technology (CIT)*, pages 1864–1871, 2010.

[163] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst. 5 envision : A 0 . 26-to-10 tops / w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28 nm fdsoi. In *Int. Solid-State Circuits Conf. (ISSCC)*, 2017.

[164] C. Moreno and S. Fischmeister. Accurate measurement of small execution times—getting around measurement errors. *IEEE Embedded Systems Letters*, 9(1):17–20, March 2017.

[165] A. Moses. *RADAR Based Collision Avoidance for Unmanned Aircraft Systems.* PhD thesis, Dep. Engineering, University of Denver, January 2013.

[166] A. Mukherjee and A. S. Dhar. Triple Transistor Based Triple Modular Redundancy With Embedded Voter Circuit. *Microelectronics Journal*, 87:101 – 109, May 2019.

[167] S. Mukherjee, J. Emer, and S. Reinhardt. The soft error problem: an architectural perspective. In *Int. Symp. High-Performance Computer Architecture (HPCA)*, pages 243–247, February 2005.

[168] O.B. Mutlu, G.K. Gioiosa, J. Manzano, O. Unsal, S.Chatterjee, and S. Krishnamoorthy. Characterization of the impact of soft errors on iterative methods. 4 2018.

[169] O.B. Mutlu, G. Kestor, A. Cristal, O. Unsal, and S. Krishnamoorthy. Ground-truth prediction to accelerate soft-error impact analysis for iterative methods. In *Int. Conf. High Performance Computing, Data, and Analytics (HiPC)*, pages 333–344, 2019.

[170] L.H. Mutuel. Single Event Effects Mitigation Techniques Report. *Federal Aviation Administration*, 15(62):470, Feb. 2016.

[171] I. Méndez-Díaz, J. Orozco, R. Santos, and P. Zabala. Energy-aware scheduling mandatory/optional tasks in multicore real-time systems. *Int. Trans. Operational Research*, 24(1-2):173–198, 2017.

[172] S.M. Nair, M. Mayahinia, M.B. Tahoori, et al. Workload-aware electromigration analysis in emerging spintronic memory arrays. *IEEE Trans. Device and Materials Reliability (TDMR)*, 21(2):258–266, 2021.

[173] A. Najafi, L. Bamberg, A. Najafi, and A. Garcia-Ortiz. Integer-Value Encoding for Approximate On-Chip Communication. *IEEE Access*, 7:179220–179234, Dec. 2019.

[174] Y. Nakamura and K. Hiraki. Heterogeneous functional units for high speed fault-tolerant execution stage. In *Pacific Rim Int. Symp. Dependable Computing (PRDC)*, pages 260–263, 2007.

[175] A. Namazi, M. Abdollahi, S. Safari, and S. Mohammadi. A majority-based reliability-aware task mapping in high-performance homogenous NoC architectures. *ACM Trans. Embedded Computing Systems (TECS)*, 17(1), 2017.

[176] S. R. Nandakumar, Ma. Le Gallo, C. Piveteau, et al. Mixed-precision deep learning based on computational memory. *CoRR*, abs/2001.11773, 2020.

[177] N. Navet and F. Simonot-Lion. Fault tolerant services for safe in-car embedded systems. *Embedded Systems: Handbook*, January 2005.

[178] H. Nguyen, J. Yu, M. Lebdeh, M. Taouil, S. Hamdioui, and F. Catthoor. A classification of memory-centric computing. *J. Emerg. Technol. Comput. Syst.*, 16(2), jan 2020.

[179] J. Nowotsch and M. Paulitsch. Quality of Service Capabilities for Hard Real-time Applications on Multi-core Processors. In *Int. Conf. Real-Time Networks and Systems (RTNS)*, pages 151–160, 2013.

[180] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In *Euromicro Conf. Real-Time Systems (ECRTS)*, pages 109–118, July 2014.

[181] F. Oboril and M. Tahoori. Cross-Layer Approaches for an Aging-Aware Design Space Exploration for Microprocessors. In *Workshop on Early Reliability Modeling for Aging and Variability in Silicon Systems (ERMAVSS)*, 2016.

[182] I. O'Connor, M. Cantan, C. Marchand, B. Vilquin, S. Slesazeck, E.T. Breyer, H. Mulaosmanovic, T. Mikolajick, B. Giraud, J.-P. Noël, A. Ionescu, and I. Stolichnov. Prospects for energy-efficient edge computing with integrated hfo2-based ferroelectric devices. In *Int. Conf. Very Large Scale Integration (VLSI-SoC)*, pages 180–183, 2018.

[183] K. Pang, V. Fresse, and S. Yao. Communication-aware branch and bound with cluster-based latency-constraint mapping technique on network-on-chip. *Journal of Supercomputing (JS)*, 72(6):2283–2309, 2016.

[184] M. Paolieri, J. Mische, S. Metzlaff, M. Gerdes, E. Quiñones, S. Uhrig, T. Ungerer, and F. Cazorla. A Hard Real-time Capable Multi-core SMT Processor. *ACM Trans. Embedded Computing Systems (TECS)*, 12(3):79:1–79:26, April 2013.

[185] A. Papadopoulos, E. Bini, S. Baruah, et al. AdaptMC: A Control-Theoretic Approach for Achieving Resilience in Mixed-Criticality Systems. In *Euromicro conf. Real-Time Systems (ECRTS)*, pages 14:1–14:22, 2018.

[186] M. K. Papamichael and J. C. Hoe. CONNECT: Re-examining Conventional Wisdom for Designing Nocs in the Context of FPGAs. In *Int. Symp. Field-Programmable Gate Arrays (FPGA)*, pages 37–46. ACM/SIGDA, Feb. 2012.

[187] A. Pathak and V. K. Prasanna. Energy-efficient task mapping for data-driven sensor network macroprogramming. *IEEE Trans. Comput.*, 59(7):955–968, 2010.

[188] R. Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems (RTS)*, 50, 07 2014.

[189] R. Pathan. Real-time scheduling algorithm for safety-critical systems on faulty multicore environments. *Real-Time Systems (RTS)*, 53, 01 2017.

[190] D. A. Patterson. Future of computer architecture, 2008.

[191] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A Predictable Execution Model for COTS-Based Embedded Systems. In *Real-time and embedded Technology and Applications Symp. (RTAS)*, pages 269–279, April 2011.

[192] Z. Peng. Building reliable embedded systems with unreliable components. In *Int. Conf. Signals and Electronic Circuits (ICSES)*, pages 9–13, Sep. 2010.

[193] M. Pignol. Dmt and dt2s: two fault-tolerant architectures developed by cnes for cots-based spacecraft supercomputer. In *Int. On-Line Testing Symp. (IOLTS)*, pages 10 pp.–, July 2006.

[194] B. Pratt, M. Fuller, and M. Wirthlin. Reduced-precision redundancy on fpgas. *Int. Journal of Reconfigurable Computing (JRC)*, 24(6):10–20, 2011.

[195] R. Psiakis. *Performance Optimization Mechanisms for Fault-Resilient VLIW Processors.* Theses, Université de Rennes 1, December 2018.

[196] G. Psychou, D. Rodopoulos, M. M. Sabry, T. Gemmeke, D. Atienza, T. G. Noll, and F. Catthoor. Classification of resilience techniques against functional errors at higher abstraction layers of digital systems. *ACM Computing Surveys (CS)*, 50(4):50:1–50:38, October 2017.

[197] X. Qi, D. Zhu, and H. Aydin. Global scheduling based reliability-aware power management for multiprocessor real-time systems. *Real Time Systems (RTS)*, 47(2):109–142, 2011.

[198] L. P. Qian, Y. J. A. Zhang, Y. Wu, and J. Chen. Joint base station association and power control via Benders' decomposition. *IEEE Trans. Wireless Communications (TWC)*, 12(4):1651–1665, 2013.

[199] Y. Qin, G. Zeng, R. Kurachi, Y. Matsubara, and H. Takada. Execution-variance-aware task allocation for energy minimization on the big. little architecture. *Sustainable Computing: Informatics and Systems (SCIS)*, 22:155–166, 2019.

[200] G. Quan and V. Chaturvedi. Feasibility analysis for temperature-constraint hard real-time periodic tasks. *IEEE Trans. on Industrial Informatics*, 6(3):329–339, 2010.

[201] M. Radetzki, C. Feng, X. Zhao, and A. Jantsch. Methods for fault tolerance in networks-on-chip. *ACM Computing Surveys (CS)*, 46(1), 2013.

[202] S. Rai, M. Liu, A. Gebregiorgis, D. Bhattacharjee, K. Chakrabarty, S. Hamdioui, A. Chattopadhyay, J. Trommer, and A. Kumar. Perspectives on emerging computation-in-memory paradigms. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1925–1934, 2021.

[203] J. Ramkumar and M. Tulika. Temperature aware task sequencing and voltage scaling. In *Int. Conf. Computed Aided Design (ICCAD)*, pages 618–623, 2008.

[204] C. D. Randazzo and H. P. L. Luna. A comparison of optimal methods for local access uncapacitated network design. *Annals of Op. Res.*, 106(1):263–286, 2001.

[205] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Conf. Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016.

[206] S. Rehman, M. Shafique, and J. Henkel. *Reliable Software for Unreliable Hardware: A Cross Layer Perspective*. Springer Publishing, 2016.

[207] G.A. Reis, J. Chang, R. Vachharajani, N.and Rangan, and D.I. August. Swift: Software implemented fault tolerance. In *Int'l Symp. Code Generation and Optimization (ISCGO)*, pages 243–254, 2005.

[208] M. Riera, R. Canal, J. Abella, et al. A detailed methodology to compute soft error rates in advanced technologies. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 217–222, 2016.

[209] H. Rihani, M. Moy, C. Maiza, R. Davis, and S. Altmeyer. Response Time Analysis of Synchronous Data Flow Programs on a Many-Core Processor. In *Int. Conf. Real-Time Networks and Systems (RTNS)*, pages 67–76, 2016.

[210] M. Rinard. Obtaining and Reasoning About Good Enough Software. In *Design Automation Conference (DAC)*, pages 930–935, 2012.

[211] D. Rodopoulos, S. Corbetta, G. Massari, S. Libutti, F. Catthoor, Y. Sazeides, C. Nicopoulos, A. Portero, E. Cappe, R. Vavrík, V. Vondrák, D. Soudris, F. Sassi, A. Fritsch, and W. Fornaciari. HARPA: Solutions for dependable performance under physically induced performance variability. In *Int. Conf. Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 270–277, July 2015.

[212] D. Rodopoulos, G. Psychou, M.M. Sabry, et al. Classification framework for analysis and modeling of physically induced reliability violations. *ACM Computing Surveys (CS)*, 47(3), February 2015.

[213] S. Rokicki, D. Pala, J. Paturel, and O. Sentieys. What you simulate is what you synthesize: Designing a processor core from c++ specifications. In *Int Conf Computer-Aided Design (ICCAD)*, pages 1–8, November 2019.

[214] B. Rouxel, S. Derrien, and I. Puaut. Tightening Contention Delays While Scheduling Parallel Applications on Multi-core Architectures. *ACM Trans. Embedded Computing Systems (TECS)*, 16(5s):164:1–164:20, September 2017.

[215] B. Rouxel, S. Skalistis, S. Derrien, and I. Puaut. Hiding Communication Delays in Contention-Free Execution for SPM-Based Multi-Core Architectures. In *Euromicro Conf. on Real-Time Systems (ECRTS)*, pages 25:1–25:24, 2019.

[216] Y. Rui, C. Qinqin, L. Zengwu, and S. Yanmei. Multi-objective evolutionary design of selective triple modular redundancy systems against SEUs. *Chinese Journal of Aeronautics*, 28(3):804 – 813, 2015.

[217] A. Ruospo, A. Bosio, A. Ianne, and E. Sanchez. Evaluating convolutional neural networks reliability depending on their data representation. In *Euromicro Conf. Digital System Design (DSD)*, pages 672–679, 2020.

[218] C. Rusu, R.i Melhem, and D. Mossé. Maximizing rewards for real-time applications with energy constraints. *ACM Trans. Embed. Comput. Syst. (TECS)*, 2(4):537–559, 2003.

[219] C. A. Rusu, R. Melhem, and D. Mosse. Maximizing the system value while satisfying time and energy constraints. *IBM Journal of Research and Development*, 47(5/6):689–702, 2003.

[220] SAE. Aerospace recommended practices 4754a - development of civil aircraft and systems, 2010. SAE.

[221] S. Safari, M. Ansari, G. Ershadi, et al. On the scheduling of energy-aware fault-tolerant mixed-criticality multicore systems with service guarantee exploration. *Trans. on Parallel and Distributed Systems (TPDS)*, 30(10):2338–2354, 2019.

[222] S. Sahoo, B. Veeravalli, and A. Kumar. A Hybrid Agent-based Design Methodology for Dynamic Cross-layer Reliability in Heterogeneous Embedded Systems. In *Design Automation Conference (DAC)*, pages 38:1–38:6, 2019.

[223] S.S. Sahoo, B. Ranjbar, and A. Kumar. Reliability-aware resource management in multi-/many-core systems: A perspective paper. *Journal of Low Power Electronics and Applications*, 11:7, 01 2021.

[224] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. Dinechin. The Shift to Multicores in Real-Time and Safety-Critical Systems. In *Int. Conf. Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, October 2015.

[225] M. Salehi, A. Ejlali, and B. M. AI-Hashimi. Two-phase low-energy n-modular redundancy for hard real-time multi-core systems. *Trans. on Parallel and Distributed Systems (TPDS)*, 27(5):1497–1510, 2016.

[226] M. Salehi, M. K. Tavana, S. Rehman, et al. DRVS: Power-efficient reliability management through dynamic redundancy and voltage scaling under variations. In *Int. Symp. Low Power Electronics and Design (ISLPED)*, pages 225–230, 2015.

[227] C. E. Salloum, M. Elshuber, O. Höftberger, H. Isakovic, and A. Wasicek. The ACROSS MPSoC – A New Generation of Multi-core Processors Designed for Safety-Critical Embedded Systems. In *Euromicro Conf. Digital System Design (DSD)*, pages 105–113, September 2012.

[228] A. Sánchez-Macián, P. Reviriego, and J. A. Maestro. Hamming SEC-DAED and Extended Hamming SEC-DED-TAED Codes Through Selective Shortening and Bit Placement. *IEEE Trans. Device and Materials Reliability*, 14(1):574–576, Mar. 2014.

[229] F. Santy, L. George, P. Thierry, et al. Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp. In *Euromicro Conf. Real-Time Systems (ECRTS)*, 2012.

[230] P. Saraswat, P. Pop, and J. Madsen. Task Migration for Fault-tolerance in Mixed-criticality Embedded Systems. *SIGBED Rev.*, 6(3):6:1–6:5, October 2009.

[231] P. K. Saraswat, P. Pop, and J. Madsen. Task Mapping and Bandwidth Reservation for Mixed Hard/Soft Fault-Tolerant Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 89–98, April 2010.

[232] A. Sartor, A. Lorenzon, L. Carro, F. Kastensmidt, S. Wong, and A. Beck. A Novel Phase-Based Low Overhead Fault Tolerance Approach for VLIW Processors. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 485–490, July 2015.

[233] A. Sartor, A. Lorenzon, L. Carro, F. Kastensmidt, S. Wong, and A. Beck. Exploiting Idle Hardware to Provide Low Overhead Fault Tolerance for VLIW Processors. *ACM Trans. Journal on Emerging Technologies in Computing (JETC)*, 13(2):13:1–13:21, January 2017.

[234] A. Sartor, A. Lorenzon, S. Kundu, I. Koren, and A. Beck. Adaptive and Polymorphic VLIW Processor to Optimize Fault Tolerance, Energy Consumption, and Performance. In *Int. Conf. Computing Frontiers (CF)*, 2018.

[235] A. Sartor, S. Wong, and A. Beck. Adaptive ILP control to increase fault tolerance for VLIW processors. In *Int. Conf.Application-specific Systems, Architectures and Processors (ASAP)*, pages 9–16, July 2016.

[236] A.L. Sartor, P.H. Becker, and A.C. Beck. A fast and accurate hybrid fault injection platform for transient and permanent faults. 2018.

[237] M. N. S. M. Sayuti and L. S. Indrusiak. Real-time low-power task mapping in networks-on-chip. In *Proc. IEEE ISVLSI*, pages 14–19, 2013.

[238] M. Schözel. HW/SW co-detection of transient and permanent faults with fast recovery in statically scheduled data paths. In *Design, Automation and Test in Europe Conf. Exhibition (DATE)*, pages 723–728, March 2010.

[239] N. Seifert, B. Gill, S. Jahinuzzaman, et al. Soft Error Susceptibilities of 22 nm Tri-Gate Devices. *IEEE Trans. Nuclear Science (TNS)*, 59, 2012.

[240] H. Shah, A. Coombes, A. Raabe, K. Huang, and A.s Knoll. Measurement based wcet analysis for multi-core architectures. In *Int. Conf. Real-Time Networks and Systems (RTNS)*, RTNS '14, page 257–266, New York, NY, USA, 2014. Association for Computing Machinery.

[241] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Inte. Conf. Dependable Systems and Networks (DSN)*, pages 389–398, June 2002.

[242] F. Siegle, T. Vladimirova, J. Ilstad, and O. Emam. Mitigation of Radiation Effects in SRAM-Based FPGAs for Space Applications. *ACM Computing Surveys (CS)*, 47(2):37:1–37:34, January 2015.

[243] M.T. Sim and Y. Zhuang. A dual lockstep processor system-on-a-chip for fast error recovery in safety-critical applications. In *Industrial Electronics Society Conf. (IECON)*, pages 2231–2238, 2020.

[244] S. Skalistis, F. Angiolini, A. Simalatsar, and G. De Micheli. Safe and Efficient Deployment of Data-Parallelizable Applications on Many-Core Platforms: Theory and Practice. *IEEE Design & Test*, 35(4):7–15, 2018.

[245] S. Skalistis and A. Simalatsar. Worst-case execution time analysis for many-core architectures with NoC. In *Int. Conf. Formal Modelling and Analysis of Timed Systems (FORMATS)*, August 2016.

[246] S. Skalistis and A. Simalatsar. Near-optimal deployment of dataflow applications on many-core platforms with real-time guarantees. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 752–757, March 2017.

[247] M. Slijepcevic, L. Kosmidis, J. Abella, et al. Timing verification of fault-tolerant chips for safety-critical applications in harsh environments. *IEEE Micro*, 34:8–19, 11 2014.

[248] M. Slijepcevic, L. Kosmidis, J. Abella, E. Quiñones, and F.J. Cazorla. Dtm: Degraded test mode for fault-aware probabilistic timing analysis. In *Euromicro Conf. Real-Time Systems (ECRTS)*, pages 237–248, 2013.

[249] H. Su, T. Lu, C. Feng, and L. Chen. Triple module redundancy reliability framework design based on heterogeneous multi-core processor. In *Int. Conf. Information and Communication Technology (ICICT)*, pages 504–511, 2020.

[250] H. Su, D. Zhu, and S. Brandt. An elastic mixed-criticality task model and early-release edf scheduling algorithms. *ACM Trans. Des. Autom. Electron. Syst. (TODAES)*, 22(2), December 2016.

[251] H. Tabkhi, M. Sabbagh, and G. Schirner. Power-efficient real-time solution for adaptive vision algorithms. *IET Computers Digital Techniques*, 9(1):16–26, 2015.

[252] B. Taylor, Q. Zheng, Z. Li, S. Li, and Y. Chen. Processing-in-memory technology for machine learning: From basic to asic. *IEEE Trans. Circuits and Systems II: Express Briefs*, 69(6):2598–2603, 2022.

[253] Texas Instruments. TMS320C6678 Multicore fixed and floating-point digital signal processor. Technical Report SPRS691D, TI, 2013.

[254] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real Time Systems (RTS)*, 18(2/3):157–179, 2000.

[255] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Int. Conf. Parallel Architectures and Compilation Techniques (PACT)*, pages 365–376. ACM, 2010.

[256] C. Torres-Huitzil and B. Girau. Fault and error tolerance in neural networks: A review. *IEEE Access*, 5:17322–17341, 2017.

[257] S. Tosun. Energy- and reliability-aware task scheduling onto heterogeneous MPSoC architectures. *Journal of Supercomputing (JS)*, 62(1), 2012.

[258] D. Trilla, C. Hernandez, J. Abella, and F.J. Cazorla. Aging assessment and design enhancement of randomized cache memories. *Trans. Device and Materials Reliability (TDMR)*, 17(1):32–41, 2017.

[259] I. Tuzov, P. Andreu, L. Medina, T. Picornell, A. Robles, P. Lopez, J. Flich, and C. Hernández. Improving the robustness of redundant execution with register file randomization. In *Int. Conf. Computer Aided Design (ICCAD)*, page 1–9. IEEE Press, 2021.

[260] I. Tuzov, D. de Andrés, and J. Ruiz. Accurate Robustness Assessment of HDL Models Through Iterative Statistical Fault Injection. In *European Dependable Computing Conf. (EDCC)*, pages 1–8, September 2018.

[261] T. Ungerer, C. Bradatsch, M. Gerdes, F. Kluge, et al. parMERASA – Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability. In *Euromicro Conf. Digital System Design (DSD)*, pages 363–370, September 2013.

[262] A. Vallero, A. Savino, A. Chatzidimitriou, et al. Syra: Early system reliability analysis for cross-layer soft errors resilience in memory arrays of microprocessor systems. *Trans. Computers (TC)*, 68(5):765–783, 2019.

[263] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symp. (RTSS)*, 2007.

[264] P. Voudouris, P. Stenström, and R. Pathan. Bounding the execution time of parallel applications on unrelated multiprocessors. *Real-time Systems (RTS)*, 2021.

[265] N.J. Wang, A. Mahesri, and S.J. Patel. Examining ace analysis reliability estimates using fault-injection. In *Int. Symposium on Computer Architecture (ISCA)*, page 460–469, 2007.

[266] S. Wasly and R. Pellizzoni. A Dynamic Scratchpad Memory Unit for Predictable Real-Time Embedded Systems. In *Euromicro Conf. Real-Time Systems (ECRTS)*, pages 183–192, July 2013.

[267] T. Wei, J. Zhou, K. Cao, P. Cong, M. Chen, G. Zhang, X. S. Hu, and J. Yan. Cost-constrained QoS optimization for approximate computation real-time tasks in heterogeneous MPSoCs. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst (TCAD).*, 37(9):1733–1746, 2018.

[268] R. Wilhelm, J. Engblom, A. Ermedahl, and othersr. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst. (TECS)*, 7, 01 2008.

[269] M. Wilkening, V. Sridharan, S. Li, et al. Calculating architectural vulnerability factors for spatial multi-bit transient faults. In *Int. Symposium on Microarchitecture (MICRO)*, pages 293–305, 2014.

[270] H.-S. Wong, S. Raoux, S. Kim, J. Liang, J. Reifenberg, B. Rajendran, M. Asheghi, and K. Goodson. Phase change memory. *Proceedings of the IEEE*, 98, 12 2010.

[271] P. Wong, H.Y Lee, S. Yu, Y.H. Chen, Y. Wu, P. Chen, B. Lee, F. Chen, and M.-J. Tsai. Metal–oxide rram. *Proceedings of the IEEE*, 100:1951, 06 2012.

[272] D. Xiang, K. Chakrabarty, and H. Fujiwara. A Unified Test and Fault-Tolerant Multicast Solution For Network-on-Chip Designs. In *IEEE Int. Test Conf. (ITC)*, pages 1–9, Nov. 2016.

[273] G. Xie, Y. Chen, Y. Liu, et al. Resource consumption cost minimization of reliable parallel applications on heterogeneous embedded systems. *IEEE Trans. on Industrial Informatics (TII)*, 13(4):1629–1640, 2017.

[274] G. Xie, Y. Chen, X. Xiao, et al. Energy-efficient fault-tolerant scheduling of reliable parallel applications on heterogeneous distributed embedded systems. *IEEE Trans. Sustainable Computing (TSC)*, 3(3):167–181, 2018.

[275] G. Xie, Y. Chen, X. Xiao, et al. Energy-efficient fault-tolerant scheduling of reliable parallel applications on heterogeneous distributed embedded systems. *IEEE Trans. on Sustainable Computing (TSC)*, 3(3):167–181, 2018.

[276] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Reliability-aware Co-synthesis for Embedded Systems. *Journal of VLSI Signal Processing Systems*, 49(1):87–99, October 2007.

[277] H. Yu, Y. Ha, and B. Veeravalli. Quality-driven dynamic scheduling for real-time adaptive applications on multiprocessor systems. *IEEE Trans. Computers (TC)*, 62(10):2026–2040, 2013.

[278] H. Yu, B. Veeravalli, and Y. Ha. Dynamic scheduling of imprecise-computation tasks in maximizing qos under energy constraints for embedded systems. In *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, pages 452–455, 2008.

[279] H. Yu, B. Veeravalli, Y. Ha, and S. Luo. Dynamic scheduling of imprecise-computation tasks on real-time embedded multiprocessors. In *Int. Conf. Computer Science and Education (ICCSE)*, pages 770–777, 2013.

[280] Q. Yu and P. Ampadu. Adaptive Error Control for NoC Switch-to-Switch Links in a Variable Noise Environment. In *Int. Symp. Defect and Fault-Tolerance in VLSI Systems (DFT)*, pages 352–360. IEEE, Oct. 2008.

[281] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality. In *Euromicro Conf. Real-Time Systems (ECRTS)*, pages 299–308, July 2012.

[282] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 55–64, April 2013.

[283] L. Zhang, K. Li, K. Li, et al. Joint optimization of energy efficiency and system reliability for precedence constrained tasks in heterogeneous systems. *Int. Journal of Electrical Power &Energy Systems*, 78:499–512, 2016.

[284] B. Zhao, H. Aydin, and D. Zhu. On maximizing reliability of real-time embedded applications under hard energy constraint. *IEEE Trans. Industrial Informatics (TII)*, 6(3):316–328, 2010.

[285] S. Zhao, X. Dai, I. Bate, et al. Dag scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency. In *Real-Time Systems Symp. (RTSS)*, pages 128–140, 2020.

[286] J. Zhou, J. Yan, T. Wei, M. Chen, and X. S. Hu. Energy-adaptive scheduling of imprecise computation tasks for QoS optimization in real-time MPSoC systems. In *Design, Automation & Test in Europe (DATE)*, pages 1402–1407, 2017.

[287] D. Zhu, R. Melhem, and D. Mosse. The effects of energy management on reliability in real-time embedded systems. *Int. Conf. Computer Aided Design (ICCAD)*, pages 35–40, 2004.

[288] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee. FlexPRET: A processor platform for mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 101–110, April 2014.

[289] N. Zompakis, M. Noltsis, L. Ndreu, Z. Hadjilambrou, P. Englezakis, P. Nikolaou, A. Portero, S. Libutti, G. Massari, F. Sassi, A. Bacchini, C. Nicopoulos, Y. Sazeides, R. Vavrik, M. Golasowski, J. Sevcik, V. Vondrak, F. Catthoor, W. Fornaciari, and D. Soudris. HARPA: Tackling physically induced performance variability. In *Design, Automation Test in Europe Conf. Exhibition (DATE)*, pages 97–102, March 2017.