Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
Dottorato di Ricerca in Ingegneria dell'Informazione

Towards the Definition of a Methodology for the Design of Tunable Dependable Systems

Doctoral Dissertation of:
Matteo Carminati

Advisor:
Prof. Cristiana Bolchini
Tutor:
Prof. Donatella Sciuto
Supervisor of the Doctoral Program:
Prof. Carlo Fiorini

2014 – XXVII

Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
Piazza Leonardo da Vinci, 32 I-20133 — Milano

Abstract

The problem of guaranteeing the correct behavior in digital systems even when faults occur has been investigated for several years. However, the researchers' efforts have been mainly devoted to safety- and mission-critical systems, where the occurrence of faults (both transient and permanent) can be extremely hazardous. Nowadays, the need to provide reliability also for non-critical application environments is gaining a lot of momentum, due to the pervasiveness of embedded systems and their increasing susceptibility due to technology scaling. While in critical applications the budget devoted to reliability is almost unlimited and it is not to be compromised, in non-critical scenarios the limited available budget used to guarantee the best performance and energy consumption is to be shared for providing reliability as well.

In the past, great effort has been devoted to provide strict reliability management. This led to the shared belief that reliability is to be considered from the early stages of the embedded systems design process. In fact, as this process is becoming more and more complex, approaches that do not consider reliability throughout all the design steps may lead to expensive or not-optimized solutions. Moreover, considering reliability in a holistic way allows to drive the several decisions by exploiting the synergy of both the most classical aspects and reliability-oriented ones. Postponing the reliability assessment to the later phases of the design flow on a system prototype is not appealing, because failure in achieving the desired level of reliability would be detected too late. However, the complexity of managing reliability, performance and power/energy consumption all at the same time grows exponentially, especially when several decision variables are available in the considered system. For this reason it is not possible to envision a system able to properly react to any possible scenario on the basis of decisions precomputed at design time; a new paradigm based on self-adaptability is to be designed. Self-adaptive systems are becoming quite common when dealing with such complex systems: relevant examples are available in literature if performance management is considered.

Given these motivations, we argue that the self-adaptive paradigm is to be implemented when designing embedded systems with the aim of considering reliability as a driving dimension. In particular, in this thesis

we propose a comprehensive management framework for dealing with reliability in multi/manycore embedded systems. Reliability is considered both for permanent/transient faults management and components aging mitigation. This framework implements a well-known control loop where the status of the system and the environment are sensed (observe), adaptation is defined through decisions made at runtime to meet the specified goals and constraints (decide), and the values of the system parameters are modified accordingly (act). The designed framework is integrated in a two-layer heterogeneous multi/manycore architecture which is considered as the reference hardware platform. Self-adaptability is enabled through independent control layers implemented on top of the reference architecture. At the top-level, the manycore one, a control cycle ensures the combined optimization of the overall architecture lifetime and of the energy consumption due to communication among nodes. On the other hand, within each node, two control cycles have been designed: one for dealing with faults occurrence (through the selection of the proper scheduling technique) and another one for mitigating components aging and minimizing energy consumption (thanks to smart mapping algorithms). Moreover, in the considered optimization process, performance is always considered as a constraint, in the form of applications deadlines, in order to guarantee a satisfying quality of service.

In addition to the above contributions, two more ones deserve to be considered. First, the definition of a model to describe and classify self-adaptive computing systems on the basis of their driving dimensions (goals and constraints), the available measures and acting parameters. Second, a tool for estimating the reliability function and the expected lifetime of complex multi/manycore architectures, able to tolerate multiple failures and considering varying workloads.

Sommario

Il problema di garantire il corretto funzionamento dei sistemi digitali è da molti anni oggetto di ricerca accademica. Tuttavia, gli sforzi dei ricercatori si sono concentrati soprattutto su sistemi il cui fallimento causa gravi ripercussioni sulla salute delle persone o sulla buona riuscita di progetti particolarmente onerosi. Al giorno d'oggi, invece, sta diventando sempre più importante la necessità di rendere affidabili anche quei sistemi che non possono essere definiti critici, ma che hanno raggiunto una grande diffusione e risultano sempre più proni alla rottura a causa delle ridotte dimensioni dei loro componenti. Mentre in scenari critici le risorse allocate per far fronte a problemi di affidabilità sono pressochè illimitate e non si deve scendere a compromessi, in scenari che non presentano criticità le risorse a disposizione sono limitate e devono essere sfruttate per ottimizzare il funzionamento del dispositivo stesso, anche in termini di prestazioni ed energia consumata.

Le ricerche svolte in passato hanno portato alla luce la necessità di considerare i requisiti di affidabilità del sistema sin dal principio del processo di progettazione. Infatti, a causa della sempre crescente complessità di questo processo, approcci che non considerano il livello di affidabilità richiesto in tutti i passi di progettazione rischiano di portare a soluzioni eccessivamente costose e/o non ottimizzate. Considerare, invece, gli aspetti di affidabilità come parte integrante di tutto il processo di progettazione permette trarre giovamento sia da tecniche classiche che da tecniche esplicitamente orientate all'affidabilità. Postporre la verifica del livello di affidabilità raggiunto al termine del processo di prototipazione non è invece una pratica saggia, poichè, in caso di fallimento, può causare la perdita di tempo e denaro. Questa necessità di integrazione causa però un aumento della complessità di progetto, in particolar modo quando i parametri su cui è possibile agire sono molti. Per questo motivo non è pensabile un sistema capace di reagire correttamente ad ogni possibile situazione sulla base di decisione pre-calcolate; è necessario adottare un paradigma di progettazione che preveda l'auto adattamento del sistema a tempo di esecuzione. I sistemi auto-adattativi per la gestione di sistemi complessi stanno infatti diventando sempre più comuni, soprattutto nell'ambito dell'ottimizzazione delle prestazioni.

Sulla base di queste motivazioni, sosteniamo che il paradigma di pro-

gettazione basato su auto-adattatività debba essere impiegato anche quando si ha a che fare con sistemi embedded che richiedano un determinato livello di affidabilità. In particolare, in questa tesi si propone un'infrastruttura completa per la gestione dell'affidabilità in sistemi embedded composti da multi/manycore. L'affidabilità viene considerata sia in termini di gestione dei guasti (sia transitori che permanenti) sia come mitigazione dell'invecchiamento dei componenti. L'infrastruttura proposta realizza un ciclo di controllo proposto in letteratura e composto da tre fasi fondamentali: l'osservazione dello stato interno del sistema e dell'ambiente in cui vive (osserva), l'adattamento attraverso decisioni prese a tempo di esecuzione per rispettare i vincoli e gli obiettivi precedentemente specificati (decidi) e l'attuazione delle decisioni prese attraverso la modifica dei parametri del sistema (agisci). L'infrastruttura è stata progettata in modo da integrarsi all'interno dell'architettura di riferimento, composta da due livelli: ad un livello più alto una piattaforma manycore, in cui ogni nodo, a livello più basso, è composto da un sistema multicore. Il sistema è reso auto-adattativo tramite la realizzazione di cicli di controllo indipendenti che lavorano ai diversi livelli di astrazione. Ad alto livello, un primo ciclo di controllo garantisce l'ottimizzazione congiunta dell'invecchiamento dei nodi e del consumo di energia dovuto alla comunicazione tra gli stessi. All'interno di ogni nodo sono stati realizzati altri due cicli di controllo: il primo si occupa di gestire le occorrenze di guasti (attraverso la selezione della tecnica di scheduling più adatta tra quelle disponibili), mentre il secondo di bilanciare l'invecchiamento dei componenti e di minimizzare l'energia consumata (grazie ad algoritmi di mapping intelligenti). Nel corso di questo processo di ottimizzazione, le prestazioni del sistema vengono sempre considerate come un vincolo da rispettare, nella forma di deadline temporali, in modo tale che il sistema garantisca sempre una soddisfacente qualità del servizio erogato.

Altri due contributi di questa tesi meritano di essere citati, a complemento di quelli già descritti in precedenza. In primo luogo, la definizione di un modello per descrivere e classificare i sistemi auto-adattativi sulla base delle dimensioni che guidano il loro adattamento, dei dati disponibili e dei parametri sui quali è possibili agire. In secondo luogo, uno strumento per stimare l'affidabilità e il tempo di vista atteso di un'architettura complessa come quella di riferimento, sottoposta a carichi di lavoro variabili nel tempo e capace di tollerare la rottura di alcuni suoi componenti.

Contents

1 Introduction **17**
- 1.1 Thesis Statement . . . 19
- 1.2 Overview of Research . . . 20
- 1.3 Contributions . . . 22
- 1.4 Publications . . . 24
- 1.5 Thesis Organization . . . 25

2 Background & Preliminaries **27**
- 2.1 Background . . . 27
  - 2.1.1 Multi/Manycore Systems . . . 27
  - 2.1.2 Dependable Systems . . . 32
  - 2.1.3 Self-Adaptive Systems . . . 35
- 2.2 Working Scenario . . . 37
  - 2.2.1 Reference Platform . . . 38
  - 2.2.2 Baseline . . . 39
  - 2.2.3 Key Performance Indicators . . . 40
  - 2.2.4 Case Study . . . 41

3 Self-Adaptive Systems Design **43**
- 3.1 Orchestrator . . . 43
  - 3.1.1 Monitoring Infrastructure . . . 44
  - 3.1.2 Decision Policies . . . 46
  - 3.1.3 Actuating Elements . . . 47
- 3.2 A Model for Self-Adaptive Computing Systems . . . 47
  - 3.2.1 The Context Concept . . . 50
  - 3.2.2 Formalization . . . 55
  - 3.2.3 Validation . . . 57
- 3.3 Final Remarks . . . 60

4 Runtime Transient Fault Management **61**
- 4.1 Background . . . 61
- 4.2 State of the Art . . . 62
- 4.3 Fault Tolerance through Self-Adaptiveness . . . 64
  - 4.3.1 Fault Management Mechanisms . . . 65
  - 4.3.2 Layer Internals . . . 67

4.3.3 Orchestrator Design 68
4.4 Experimental Results 76
4.4.1 Experimental Set-up 77
4.4.2 First Experimental Session 78
4.4.3 Second Experimental Session 81
4.5 Final Remarks 84

5 Runtime Aging Management **87**
5.1 Background 87
5.2 Aging Evaluation 91
5.2.1 State of the Art 91
5.2.2 The proposed framework 92
5.2.3 Experimental evaluation 97
5.3 Aging Mitigation through Self-Adaptiveness 104
5.3.1 Single Node 107
5.3.2 Multi Node and Computation Energy 115
5.3.3 Multi Node and Communication Energy 118
5.3.4 Putting it all together 128
5.4 Final Remarks 131

6 Conclusions & Future Works **133**

List of Figures

1.1 A graphical overview of the proposed system composition. 21

2.1 Representation of the defined architectural concepts. . . . 28
2.2 A simple application represented by a task-graph. 31
2.3 The Observe-Decide-Act (ODA) control loop. 36
2.4 Graphical representation of the reference platform. 39

3.1 Overview of the proposed framework. 45
3.2 Graphical representation of the context meta-model; each rectangle represents a context dimension, those with rounded corners are the driving dimensions. Dotted rectangles indicate dimensions that might not exist in some contexts. Segments represent relations among dimensions. 54
3.3 Context meta-model and dimension domains for self-adaptive computing systems. 54
3.4 Context dimensions and domains for METE research project. 59
3.5 Context dimensions and domains for Into The Wild project. 59
3.6 Context dimensions and domains for Metronome project. . 60

4.1 Context dimensions and domains for the fault tolerance management self-adaptive system. 61
4.2 System structure overview: the FM layer acts between the applications and the OS. 64
4.3 Duplication with Comparison (DWC) technique applied to all the tasks of the sample application of Figure 2.2. . . 65
4.4 Triplication (TMR) technique applied to all the tasks of the sample application of Figure 2.2. 66
4.5 Duplication with Comparison and Re-execution (DWCR) technique applied to all the tasks of the sample application of Figure 2.2. 67
4.6 The Observe–Decide–Act control loop. 68
4.7 Application's task–graph hardened at the two different levels of granularity: the colored dashed tasks represent the voter nodes added to make fault mitigation possible. . 71

4.8 If PE0 is permanently faulty, the presented mapping of the tasks on three processing cores, PE0, PE1 and PE2, causes TMR to fail. 72
4.9 A FSM representation of the decision process in the case reliability is the constrained dimension. 75
4.10 Task-graph for the edge detector application. 78
4.11 Overall execution times for the edge detector on the various architectures stimulated by each fault list. 80
4.12 Metrics computed over the overall experiment execution. . 82
4.13 Throughput for the edge detector on the architecture with 12 processing cores, stimulated with a fault list presenting a variable failure rate and a permanent fault. 84

5.1 Model of a PE's temperature profile over time. 88
5.2 Reliability curve considering temperature changes. 89
5.3 Workflow of the proposed framework. 93
5.4 Example of application execution and thermal profile. . . 95
5.5 Simulations execution times w.r.t. number of failures k. . 102
5.6 Iterations number w.r.t. different architectures topologies. 102
5.7 Coefficient of variation w.r.t. number of failures k. 102
5.8 Comparison vs. past works for constant workloads. 103
5.9 Reliability curve for different workload change's periods. . 105
5.10 Context dimensions and domains for the aging mitigation self-adaptive system. 105
5.11 Energy consumption vs. lifetime optimization. 107
5.12 Block diagram for the reference energy optimization framework. 110
5.13 Block diagram for the proposed lifetime and energy optimization framework. 110
5.14 Extension of the proposed framework to a mult-node architecture. 110
5.15 The proposed approach compared with the baseline framework in a *single node* architecture. 115
5.16 Control flow chart for the dispatching algorithm. 116
5.17 The proposed approach compared with the baseline framework in a *multi-node* architecture. 118
5.18 Overview of the proposed methodology. 122
5.19 MTTF performance of the proposed approach. 126
5.20 MTTF performance with multi-application and multi-throughput scenarios. 128
5.21 Communication energy performance for the proposed technique. 129

5.22 Overview of the proposed approach when both communication and computation energy consumptions are optimized. 131

6.1 A graphical representation of the research contribution of this thesis, organized in *self-adaptive layers*. 134

List of Tables

2.1 Baseline summary. 40

3.1 Model constraints for the METE self-adaptive system. . . 58

4.1 Qualitative performance comparison between TMR and DCR techniques. The comparison refers to the same level of granularity. 70
4.2 Qualitative evaluation of the described techniques with reference to reliability aspects (FD: Fault Detection – FT: Fault Tolerance). 73
4.3 Execution times for voting and checking various amount of data. 77

5.1 Comparison of the state the art frameworks (*Architecture*: single core (S), multicore (M) – *Aging Model*s: exponential distribution (E), Weibull distribution (W) – *Workload*: average temperature (Avg T), average aging factor (Avg α) during the considered period). 92
5.2 Energy performance of the proposed dynamic approach with MTTF optimization only 127
5.3 Parameters for multi-application and multi-throughput . . 127

1 Introduction

The trend of building new complex systems by integrating low-cost, inherently unreliable Commercial Off-The-Shelf (COTS) components is one of today's challenges in the design, analysis and development of modern computing systems [55]. In fact, the last decade has seen the complexity of electronic systems grow faster and faster, because of the decrease of the components' size and cost. However, this harsh technology scaling has led to an increase of the susceptability to both permanent and transient faults due to the variations in the manufacturing process and to the exposition of devices to radiations and noise fluctuations [55]. More precisely, the aggressive CMOS scaling exploited to boost computational power, generates higher temperatures, causing physical wear-out phenomena that increase the probability for permanent faults due to components aging to appear. This susceptability is further exacerbated when considering systems constituted by several computational units (to achieve high performance) because of the additional environmental interactions between the hardware and the software of the different units.

A permanent fault, also known as *hard fault*, indicates the permanent going out of commission of a component; on the other hand, a transient fault, or *soft fault*, does not damage a component permanently, but causes glitches in the elaboration, randomly corrupting either the computed data or the control flow being executed, thus jeopardizing the output results [61]. Radiations are among the main causes of transient faults [40]; they are particularly frequent in space, but are becoming an issue also at ground level [77]. Permanent faults are, instead, closely related to devices usage and wear-out phenomena, which are strictly dependent on temperature, operating frequency, voltage and current-density.

All these considerations, particularly hazardous in safety- and mission-critical systems (such as automotive devices and controls, railways, aerospace systems and plant control systems), seriously affect ordinary devices as well, when considering the embedded system's pervasiveness in today's life (consumer electronics and home appliances) [100]. Therefore, *reliability* is increasingly adopted as one of the main optimization goals, together with *performance* and *energy optimization*, and it needs to be taken into account from the initial phases of the design process.

In non-critical environments, where the available budget to be spent for reliability is limited, it must be leveraged in order not to introduce too high costs and/or stringent requirements. This scenario is further complicated by the current shift towards parallel architectures, such as multicores and manycores. In the last years indeed, a lot of attention has been devoted to the design of architectures integrating multiple cores on a single die, to benefit from the relatively low cost of the computational power, while offering high performance through the parallel execution of different applications [14]. The number of computing resources and their power/performance profiles characterize the overall architecture, that can be classified as *multicore* (several units) or *manycore* (several tens of units), and *homogeneous* (all identical units) or *heterogeneous* (units with different profiles). However, this opportunity increases the difficulty of managing the available resources, considering the complexity of selecting the most appropriate resource mixture where to run the workload, and the fact that, even when homogenous in terms of power/performance, each resource will have its own history, making it heterogeneous from other points of view (e.g. aging and wear-out levels).

Another level of complexity is introduced by the high dynamism of the working scenario the considered systems live in. In particoular: i) optimization goals may vary at runtime (performance, energy consumption, reliability), ii) the workload may not be known in advance, and iii) permanent and transient faults may occur, dynamically affecting the behavior of the system.

As mentioned, in many scenarios the need for reliability may change during the system's activity, depending on the specific working scenario, or it cannot be foreseen at design-time due to the unpredictability of the environment. Moreover, knobs allowing the runtime configuration of the cores' working conditions (as for instance, dynamic voltage/frequency scaling – DVFS) are usually available. For these reasons, a new way to dynamically *tune* the reliability management based on the working scenario is needed, taking into consideration the incidence of both permanent and transient faults. The challenge in identifying a suitable solution for this problem is given by the need for finding a satisfying trade-off between benefits and costs at runtime. Since the overall reliability problem is not new, although becoming more and more relevant, literature offers a wide set of reliability-oriented approaches; however, most of them tackle the problem of permanent and transient faults singularly or do not take into consideration the possibility for the system requirements to vary at runtime, thus needing the initial solution to be recomputed.

1.1 Thesis Statement

The previous discussion highlighted the necessity for a new paradigm to be employed in embedded system design, where reliability is to be considered as a leading dimension of the design process from its early stages. It would be unfeasible for the designer to manually evaluate all the constraints and optimize the system for a wide range of scenarios: conditions change constantly, rapidly, and unpredictably. For this reason, the resulting solutions space would be too wide for an exhaustive exploration, even at design-time. It would be desirable to have the system able to autonomously adapt to the mutating environment at runtime. *Self-adaptiveness* proved to be the answer to most of the problems previously described. A self-adaptive system is able to adapt its behavior to autonomously find the best configuration to accomplish a given goal (e.g. a performance level, an energy budget) according to the changing environmental conditions and given constraints. However, the design of a self-adaptive system able to dynamically adapt, taking away this burden from the user, is a complex engineering problem. Such a system needs to monitor itself and its context, discern significant changes, determine how to react, and execute decisions. Runtime data are exploited to better perform the hard problem of tuning all the system's parameters (which hardening technique to select, how to map and schedule applications on the available cores, etc.). Similarly, different types of quantities can be considered and monitored in order to make the system aware of itself and able to better perform (cores temperature, power consumption, applications performance, etc.). By coupling one (or more) resource(s) with one (or more) quantity(ties), many different aspects of self-adaptiveness can be implemented.

An entity implementing such an intelligence has been envisioned and dubbed *orchestrator*. As its name suggests, this component takes care of managing (orchestrate) all the self-adaptive aspects of the system: it gathers information about its status, makes decisions about the best values for the system's parameters, and sets the chosen value through the system's knobs. Since the effects of each knob are not necessarily independent from one another, the orchestrator must also be able to identify possible disruptive interactions and unexpected side effects, and to solve them. All these tasks are to be performed at runtime to have updated information about the system status and prune the complex solution space; it would be unfeasible for the problem to be tackled at design time both because of the huge dimension of a solution space considering any possible scenario and because of the lack of certain information, which can be retrieved only when the system is actually executing. However, the

runtime execution requires the orchestrator to have a reduced overhead for the advantages introduced by self-adaptiveness not to be wasted.

1.2 Overview of Research

In this context, the research developed in these years and presented in this thesis addresses permanent and transient faults and attempts to provide a unified reliability framework to support and achieve fault tolerance and aging mitigation capabilities for multi/manycore architectures. It is important to highlight that reliability features are to be incorporated in the system while meeting possible performance or energy/power constraints. Even when constraints are not specified for such dimensions, reliability must be leveraged and trade-offs are to be exploited to minimize costs and overheads.

In recent years, the idea of self-adaptive (computing) systems has received a lot of attention [60], however none of these initial concepts included reliability in the picture. Moreover, often self-adaptiveness has been designed and implemented implicitly without a foundational approach. Thus, we studied and developed a model for describing and classifying self-adaptive computing systems, clearly showing the goals and the constraints of the modeled system. It describes the available knobs and how these knobs can interact with the goal/constraint dimensions; it also analyzes the sensing portion of the system at different abstraction levels: from raw data to how they are processed and aggregated in order to extract knowledge from them. Even if developed for analyzing self-adaptive reliable systems, the model permits to formally describe, classify and analyze self-adaptive computing systems in general.

The developed model allowed us to focus and convey our efforts in the design of a central entity able to coordinate all the aspects of a self-adaptive system and governing the available resources, the *orchestrator*. As mentioned, the orchestrator is a runtime manager that makes smart decisions on how to set the system's parameters on the basis of the information gathered during the execution. Again, even if actually implemented for dealing with reliability, such a component is potentially usable in any kind of scenario.

The problem of dealing with both transient faults and wear-out phenomena simultaneously is a hard one, mainly for the different timescales at which they are to be managed. The occurrence of faults (both soft and hard ones) requires a prompt reaction to prevent the propagation of faulty results; on the other hand, architecture lifetime extension is

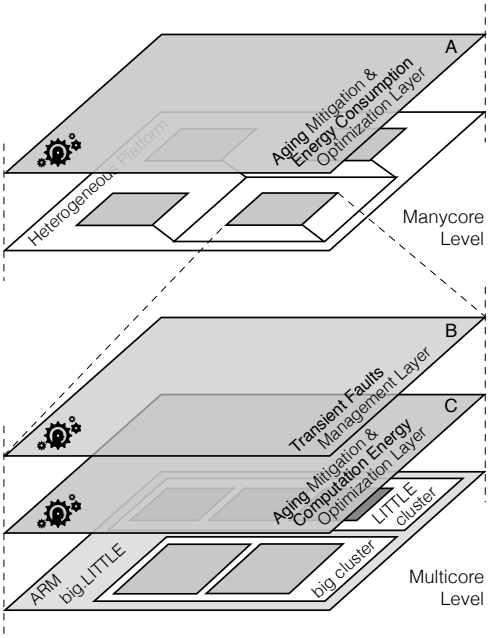


Figure 1.1: A graphical overview of the proposed system composition.

obtained through a wise use of the resources based on algorithms that proactively distributes the workload on the basis of the past components wear-out. For these reasons the problem has been tackled separately on two different architectural levels; Figure 1.1 gives an overview of the proposed system composition. The first tackled problem is the combined optimization of lifetime reliability and energy consumption at the many-core level (layer A in Figure 1.1). Energy consumption is considered both in terms of communication and computation energy. The designed orchestrator is in charge of selecting the best initial mapping of the workload and to dynamically adapt it to achieve the user's specified goals.

At the multicore level, a two step optimization takes place. First, transient faults are managed at a higher abstraction level (layer B in Figure 1.1), by selecting the best reliable scheduling technique at runtime, to optimize the reliability/performance trade-off. The decisions made by this layer are further optimized (layer C in Figure 1.1) for taking components aging and computation energy consumption into consideration. This two step optimization process does not lead to a suboptimal result since the two approaches are perfectly complementary. In fact, the fault management layer, which selects the reliability technique to be applied, the number of replicas to be created and the need for executing a vot-

ing or checking task, lays at a higher abstraction than the one where the orchestrator designed for aging and energy optimization acts. This orchestrator can make decisions on tasks mapping and DVFS without having any information about what each task actually represents (e.g. the original version of the task, a replica, or even a voter or a checker). On the other hand, being at a higher abstraction level, the decisions taken by the fault management layer are not influence by the actual mapping of the tasks on the PEs or by the frequency at which the clusters are running.

Thus, in the overall picture, in this thesis it will be presented a system able to deal with both transient faults as well as aging and wear-out phenomena, by autonomously adapting to the evolving scenario, without negatively impacting performance and energy consumption.

1.3 Contributions

The research has addressed the various aspects related to the design and (prototype) implementation of a self-adaptive reliable computing system. Indeed, the main innovative contributions are summarized in the following.

A model for self-adaptive computing systems. A deep investigation on the concept of *context-awareness* and *self-adaptiveness* in the field of computing systems. While in different research areas, such as database and software engineering, they have received a lot of attention, we witness a direct exploitation of an adaptive, context-aware behavior, implicitly resorting to a model of context without its precise introduction. The research effort brought to the definition of a model for self-adaptive computing systems able to express the elements affecting their behavior and triggering adaptation, including the relationships and constraints that exist among them. The model is the first building block of a framework for modeling, developing and supporting the implementation of self-adaptiveness in computing systems.

Self-adaptive fault tolerance for transient faults in multicore architectures. Study and definition of a novel approach in the design of multicore with self-adaptive fault tolerance, acting at the thread scheduling level. The adoption of fault management strategies at such abstraction level is a well known problem; however, it has been tackled by the existing approaches at design-time. A fault management layer has been introduced at the operating system level, implementing a strategy for

dynamically adapting the achieved reliability. This self-adaptive system has been designed by exploiting the models described in the previous point. It consists of a runtime manager (the orchestrator) in charge of making decisions concerning the adaptation of the system to changing resource availability. More precisely, the methodology we envision is suited for application scenarios where the reliability requirements need to be enforced only in specific working conditions (e.g., when an emergency situation arises) and/or for limited time-windows. In these cases, the system can adapt its behavior and expose robust properties with possibly limited performance overhead, or the other way around. This is mainly due to the fact that performance and reliability are usually conflicting goals. The role of the orchestrator is to balance this trade-off, by making decisions on task scheduling.

A tool for estimating multi/manycore lifetime. An in-depth study of wear-out mechanisms and their effects on multi/manycore architectures. Because components aging is hard to be measured due to the lack of sensors on commercial platforms, stochastic models have been built to estimate the reliability of a system. When the considered system is a multi/manycore one and multiple failures can be tolerated, things are more complicated, both from a theoretical and a computational point of view. The contribution on this topic is twofold: on one hand we provide a clear formalization of the problem and, on the other hand, we present techniques for making the computation feasible. The outcome is a framework for the lifetime estimation of multicore architectures, based on Monte Carlo simulations and random walks.

Techniques for lifetime extension in multi/manycore architectures. Integration of the proposed lifetime reliability evaluation framework with a self-adaptive system for lifetime extension in multi/manycore architectures. Overall system lifetime can be improved by selecting which among the available healthy cores to use and how intensively; an orchestrator has been designed to make decisions about applications mapping, aiming at maximizing architecture lifetime, while considering the aging status of the architecture in each instant of time. Together with reliability optimization, other dimensions have been taken into consideration, such as energy consumption and performance. This research direction was carried out in collaboration with the National University of Singapore - School of Computing, in the person of Prof. Tulika Mitra.

1.4 Publications

Most of the ideas described in this thesis have been presented at international conferences and appear in the associated proceedings or have been published in other international journals. Other papers are presently submitted for possible publication. The list of the related publications follows.

- C. Bolchini, M. CARMINATI, A. Miele and E. Quintarelli, *A Framework to Model Self-Adaptive Computing Systems*, in Proceedings of NASA/ESA Conference on Adaptive Hardware and Systems - June, 2013 - pp. 71–78.

- C. Bolchini, M. CARMINATI, and A. Miele, *Towards the Design of Tunable Dependable Systems*, in Proceedings of Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale - June, pp. 2012 - 17–21.

- C. Bolchini, M. CARMINATI and A. Miele, *Self-Adaptive Fault Tolerance in Multi-/Many-Core Systems*, Journal of Electronic Testing: Theory and Applications - Volume 29 Issue 2, April 2013 - pp. 159–175.

- C. Bolchini and M. CARMINATI, *Multi-Core Emulation for Dependable and Adaptive Systems Prototyping*, in Proceedings of Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale - March 2014 - pp. 1–4.

- C. Bolchini, M. CARMINATI, M. Gribaudo and A. Miele, *A lightweight and open-source framework for the lifetime estimation of multicore systems*, in Proceedings of International Conference on Computer Design - October, 2014 - pp. 1–7.

- C. Bolchini, M. CARMINATI, A. Miele, A. Das, A. Kumar and B. Veeravalli, *Run-Time Mapping for Reliable Many-Cores Based on Energy/Performance Trade-Offs*, in Proceedings of Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems - October, 2013 - pp. 58-64.

- C. Bolchini, M. CARMINATI, T. Mitra and T. Somu Muthukaruppan, *Combined On-line Lifetime-Energy Optimization for Asymmetric Multicores*, Technical Report, submitted to Conference on Design, Automation and Test in Europe, 2015.

1.5 Thesis Organization

The ideas at the basis of the proposed research are presented in this thesis as follows. Chapter 2 provides a short overview of the context, presenting some background knowledge (namely, multi/manycore architectures, reliability, and self-adaptive systems) and defining the working scenario in terms of case studies, baseline for comparison and evaluation of the results (key performance indicators – KPIs).

The model for self-adaptive computing systems is presented in Chapter 3. The concepts here introduced allow us to describe the proposed approach from a high abstraction level and to focus on the orchestrator design and implementation. Then, the subsequent two chapters will dwelve in the detailed presentation of how self-adaptive systems for transient faults management and aging mitigation have been implemented.

Chapter 4 describes transient fault management: the state of the art on fault-tolerance for multi/manycore architectures will be presented and its main common points with the proposed research highlighted. The need for self-adaptivity will be discussed by showing that a solution computed at the time the system is designed might not identify the optimal working condition with respect to the active context. As a consequence, the advantages of having designed a centralized entity able to manage the available resources at the time of execution, the orchestrator, is shown. In this context, it is in charge of selecting among the available reliable task scheduling techniques, which provide increasing levels of reliability at different costs. The best scheduling technique is to be chosen at each instant of time, according to the context in which the system is executing and to the required reliability level.

The design of the self-adaptive system for dealing with lifetime reliability is presented in Chapter 5. Differently from what has been done in the previous chapter, the identification of proper metrics for the evaluation of different wear-out effects required a deep and detailed study. This is mainly due to the stochastic nature of the concept of failure: it is hard to quantify the probability for a given component to fail at a certain time. It is even more difficult when a complex system, consisting of many components interfering each other, is considered. Task mapping algorithms for aging mitigation on multi/manycore architecture are presented step by step. First, the single node scenario is considered; in this case the orchestrator is in charge of making decisions about how to move tasks inside the node and how to manage DVFS. The attention is then moved to the multi-node scenario, where the application mapping problem is tackled as well. The decision making process is mainly led by the aging accumulated by each component and their thermal interaction.

Last, Chapter 6 draws some conclusions and identifies possible future research directions are discussed.

2 Background & Preliminaries

The aim of this chapter is twofold. First, we introduce the background knowledge at the basis of the work proposed in this thesis. Second, we present the considered working scenario, the baseline against which we will compare the proposed solutions in terms of architecture and case studies.

2.1 Background

This background section is divided into three main parts: multi/manycore systems, dependable systems, and self-adaptive ones. Each of them will be described in the following sections.

2.1.1 Multi/Manycore Systems

In the last years a lot of attention has been devoted to the design of architectures integrating multiple cores on a single die, moving away from the old paradigm of having a single core, extremely powerful and power-hungry. This direction change allows the new architectures to benefit from the relatively low cost of the computational power, while offering high performance through the parallel execution of different applications.

The number of computing resources and their power/performance profiles characterize the overall architecture, that can be classified as *multi-core* (several units) or *manycore* (several tens of units). The properties of the units that compose the architecture allow us to classify them under another point of view: *homogeneous* or *symmetric*, when all units are identical, and *heterogeneous* or *asymmetric*, if units have different profiles. This asymmetry can be further classified as performance or functional one. *Performance asymmetry* means that cores support identical instruction set architectures (ISAs), but exhibit power-performance heterogeneity because of differences in their hardware design or configuration. Examples of such an architecture are the ARM big.LITTLE [3] and the NVIDIA Tegra [78]. *Functional asymmetry* occurs when a subset of cores has different computational capabilities, exposed, for example, through ISA extension; this is the case, for example, of hardware accelerators.

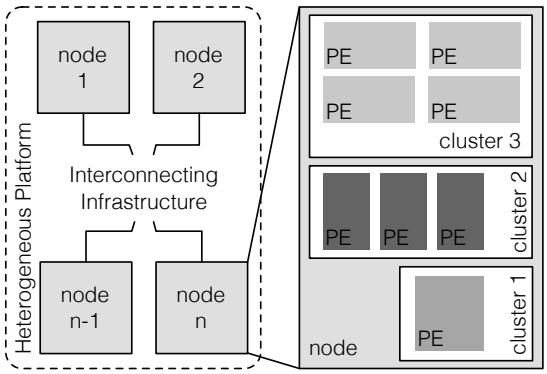


Figure 2.1: Representation of the defined architectural concepts.

Regardless of the number of units in the architecture and their type, each one of them is generically referred to as *processing element* (PE). Homogeneous PEs characteristics are usually grouped into *clusters*, which usually have their own parameters (such as the voltage-frequency working point) and represent the lowest aggregation level. At a higher level, clusters can be organized into *nodes*; within an architecture, these nodes can be homogeneous or not. Thus, a system can be defined as homogeneous if looking at the nodes only, while it is heterogeneous if considering the PEs; Figure 2.1 provides a graphical representation of the expressed concepts.

Several communication, programming, and execution models are typical of multicore or manycore architectures and will be discussed in the following paragraphs.

Communication Models

When dealing with a small number of processing elements communication is usually implemented through a *bus*-like infrastructure. However, when the number of connected elements approaches ten, the bus system will produce a performance bottleneck problem [56]. An alternative communication solution is a fully *crossbar* system. It is a non-blocking switch, connecting multiple inputs to multiple outputs in a matrix manner ("non-blocking" means that other concurrent connections do not prevent connecting an arbitrary input to any arbitrary output). However, as the number of components increases, the complexity of the wires could be dominant over the logical parts.

Finally, the *Network on Chip* (NoC) interconnection system has been

introduced as the solution to these problems [10]. NoC entails a unified solution to on-chip communication and the possibility to design scalable systems at supportable levels of power consumption. NoCs can provide a flexible communication infrastructure, consisting of three main components on a NoC-based system [39]: i) the Network Interface Controller (NIC), which implements the interface between each component and the communication infrastructure, ii) the router (also called switch), in charge of forwarding data to the next queue according to the implemented routing protocol, and iii) the links, the specific connections that provide communication between components.

NoCs can be classified according to the implemented switching method: circuit switching and packet switching. In the former, a path from source to destination is reserved before the information is emitted through the NoC components; all data are sent following the reserved path, that is released after the transfer has been completed. In the latter, there is not a reserved path; instead, data are forwarded hop by hop using the information contained in the packet. Thus, in each router the packets are buffered before being forwarded to the next router. Since NoC interconnection systems have replaced traditional bus interconnection systems, many topologies have been proposed. In this thesis regular (mesh-like) and direct (where all nodes are attached to a core) topologies only will be considered.

Programming Models

Having a scalable communication system is not sufficient to achieve full scalability: it is mandatory for the programmer to have tools to design applications that will run efficiently on multi/manycore systems. The programming model is the necessary mean to abstract the logic of applications and translate it to the hardware platform system. It must provide scalability, that is, to an increase in the system hardware resources it must correspond an increase in its performance. The main programming models relevant for this work will be now briefly presented; for a more comprehensive discussion, the reader can refer to [59, 92].

In the *shared-memory* model, communication occurs implicitly through a global address space accessible to all cores. This model forces the programmer to explicitly handle data coherence and synchronization. Systems based on this model can suffer of a performance bottleneck due to the congested accesses to the memory hierarchy.

This bottleneck can be avoided by explicitly moving data between sender and receiver: this kind of model is called *message passing*. Message passing implies a set of cores with no shared address and explicit

collaboration between sender and receiver. The most common primitives used for communication in this model are *send* and *receive*, and always a send operation must match a receive one. This model could overcome the non-determinism and the scalability limits that cache coherence protocols introduce in shared memory architectures. The main drawbacks of message passing are that the programmer must explicitly implement parallelism and data distribution, dealing with data dependencies and inter-process communication and synchronization.

Other programming models that explicitly take parallelism into consideration are the *thread*-based and the *data*-based ones. The former considers a process as composed by multiple threads. Each thread represents an independent execution domain and can run independently from the others; thread-based parallelism focuses on distributing *execution threads* across the available PEs. On the other hand, data parallelism focuses on distributing *data* across the available PEs, i.e. the parallelism is determined by the data partitioning. A more detailed presentation of these models is available in [64], however additional aspects useful to follow the discussion will be introduced when necessary.

Execution Models

The smallest independent execution unit is usually referred to as *application*. An application can be run potentially infinite times with different input data; each one of these run is an application *instance*. The introduction of tens or hundreds of cores on the same chip pushes the *sequential* application execution to be replaced by a *parallel* one. To make the execution parallel, the application model must be further detailed in order to identify the code sections that do not show any dependency and can be simultaneously executed.

A widely adopted way of describing parallel applications is the *fork-join* model, used for instance by Open-MP [95]. The key element in this programming model is the *thread*; each thread corresponds to a different portion of the application's static code. In addition to their code, threads are characterized by their input and output data, i.e. data that is used and produced by the thread, respectively. An application is composed by a master thread that may fork to create child threads; when it does, the master execution is blocked by means of a barrier mechanism waiting for the termination of all its children; only when this occurs, it resumes. Similarly, each child thread can fork to create other threads and will be blocked until their termination. Such an application can be represented by means of a direct acyclic graph, as the example shown in Figure 2.2.

Each node in this graph represents the execution of a *task*, a segment

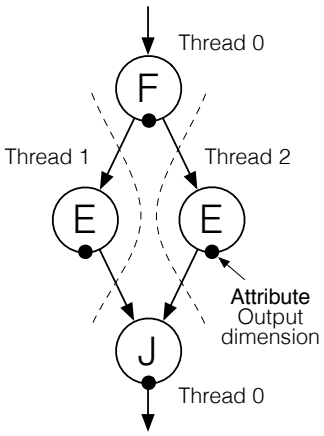


Figure 2.2: A simple application represented by a task-graph.

of the program code (a thread or a part of it), while the arrows represent data dependency among the nodes. More precisely, the fork-join graph is composed by four different types of tasks, characterized on the basis of the topology of the task-graph and the thread primitive called in the associated code segment: *elaborate* (E), *fork* (F), *join* (J), and *join_fork* (FJ). For the sake of completeness, it is worth noting that a fork node always generates child tasks belonging to different threads, while the join node is part of the same thread the fork node belongs to. Hence, in our example, node F and node J belong both to the same master thread that generates the two children threads containing the E nodes. Each node in the task-graph is annotated with the amount of produced data for the final result. In fact, as the example in Figure 2.2 shows, in the considered applications, data processing is split into several parallel elaboration tasks, while the other nodes have a synchronization and control functionality.

As it will be later discussed when introducing the proposed case studies, the considered application scenario is the one of image and video processing applications. This means that the overall algorithm represented by the task-graph is continuously repeated in a cyclic fashion in order to process each received data chunk; this execution model is usually referred to as *data-driven multithreading* (DDM). This is a nonblocking multithreading execution model that tolerates inter-node latency by scheduling threads for execution based on data availability. The synchronization part of a program is separated from the computation part, that represents the actual instructions of the program executed by the PE.

The synchronization part contains information about data dependencies among threads and is used for thread scheduling. In the DDM model, scheduling of threads is determined at runtime based on data availability, i.e., a thread is scheduled for execution only if all of its input data is available in the local memory. As with all dataflow models, DDM major benefit is the fact that it exploits implicit parallelism. Effectively, scheduling based on data availability can tolerate synchronization and communication latencies.

2.1.2 Dependable Systems

Dependability is a general term indicating the property of a (computer) system such that reliance can justifiably be placed on the service it delivers [66]. Under the umbrella of this definition, a plethora of properties has been better formalized: reliability, availability, safety, integrity, maintainability, testability, etc.. *Reliability*, $R(t)$, is defined as the probability for a given system to operate continuously and correctly, in a specified environment, in the time interval $[0, t]$. On the other hand, *availability*, $A(t)$, is the average fraction of time over the same $[0, t]$ interval during which the system is performing correctly [61]. It is usually meaningful to refer to the reliability of a system when, in the considered scenario, even a momentary disruption can prove costly; when continuous performance is not vital but it would be expensive to have the system down for a significant amount of time, availability is the measure which to refer to.

System reliability is closely related to *Mean Time To Failure* (MTTF) and *Mean Time Between Failures* (MTBF). The former is defined as the average time the system operates until a failure occurs, considering a non-repairable system, which would thus fail as soon as a fault occurs. MTBF is the average time between two consecutive failures, a measure usually adopted when dealing with repairable systems. The difference between the two is the time needed to repair the system following a failure (*Mean Time To Repair* – MTTR). Given these definitions, the availability of a system can also be written as:

$$A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR}.$$

These quantities will be computed or evaluated for characterizing the dependability of a system, taking into account the specific adopted fault models, presented in the following.

Fault Models

A comprehensive classification of faults has been initially presented in [61]; we here recall the relevant elements for our discussion. In the context of this thesis, a *fault* is defined as a hardware defect. An *error* is, instead, a manifestation of such fault, i.e. a deviation of the system from the required operation. Finally, a *failure* happens when the system fails to perform its required function. Hardware faults can be further classified according to several aspects. If the duration is considered, it is possible to distinguish between: *permanent* faults, when the system permanent goes out of commission, *transient* faults, when the malfunctioning is reduced in time and after that time the system functionality is fully restored, and *intermittent* faults, which oscillate between being quiescent and active. Permanent faults are usually due to hardware design defects or to components wear-out and aging. On the other hand, the source of transient and intermittent faults is usually considered to be random: a common example of random fault origin is radiations [8].

Another classification of hardware faults is into benign and malicious ones. A fault that just causes the system to switch-off is called *benign*. Even if counter-intuitive, such faults are the easiest ones to deal with. In fact, far more insidious are the faults, called *malicious* or Byzantine, that cause a unit to produce reasonable-looking, but incorrect, outputs.

Given the abstraction level adopted in the characterization of the architecture and applications, the fault model adopted in this thesis tries to capture the effect of physical faults in a processing core executing a task. The fault model is referred to as *processor failure* and it may be caused by transient, permanent or intermittent faults. In particular, when a fault affects one processing core, the tasks executed on such core will exhibit an incorrect behavior. Moreover we adopt the *single processor failure* assumption: at each time instant, only one failure can affect the architecture (independently from the number of physical faults causing it), and a subsequent failure will occur only after an amount of time that allows the detection of the previous one. This is a commonly adopted assumption, that does not impose particular restrictions. In fact, the cardinality of the physical faults is not limited, but rather the area of their impact is restricted to a single core of the architecture. Furthermore, more cores can fail however not all the same time instant. In case of a transient fault, the task being executed on the affected PE will exhibit an incorrect behavior and a re-execution of the task should produce no errors. In the case of a permanent or intermittent fault, all the tasks running on the faulty PE produce erroneous data or behave incorrectly.

Hardening Techniques

When designing a dependable system, fault management requirements are expressed, to determine how the faults should be dealt with [24]:

- fault *ignore*: no fault management capability is required;
- fault *detection*: the occurrence of any error has to be detected;
- fault *mitigation*: the correctness of the result has to be guaranteed.

The three classes are ordered and, in particular, fault mitigation includes fault detection. Fault *diagnosis* can also be included within these dependability requirements, even if it can be combined with both fault detection and mitigation. It describes the system capability of identifying the faulty component. This can be done by running further diagnosis task, or by analyzing historical data.

Literature offers a wide set of techniques that can be used to provide any of the presented levels of dependability to the system; all of them are based on properly managing and exploiting *redundancy* to detect or mask the errors. It is possible to identify several different forms of redundancy; the interested reader can refer to [61] for an exhaustive dissertation on the topic. Since the hardening techniques exploited in this work are at the processor level, only some of them proved to be useful: namely space and time redundancy. Space redundancy is provided by more resources (using more PEs) than strictly required to be able to identify the occurrence of a failure or mitigates its effects. For example, *replicas* of the same task are created and executed on different PEs. Fault detection requires the output of at least two replicas to be compared. If a mismatch in the output data is observed, then a fault has occurred. *Duplication with comparison* (DC) uses two replicas and provides the system with fault-detection capabilities, but does not allow to straightforwardly diagnose which of the two PE is faulty. Given the described fault model, diagnosis is made easier if three replicas and a majority voter are employed. This technique, dubbed *Triple Modular Redundancy* (TMR), provides fault mitigation as well; in fact, when a fault occurs, at least two out of three replicas will provide the correct matching output. These are example of static redundancy, since redundancy is exploited regardless the appearance of a fault. A different form of space redundancy is dynamic redundancy, where spare replicas are activated upon the failure of a currently active component. An example of such a category is *Duplication with Comparison and Re-execution* (DWCR), where the third replica is issued only in case an error is detected within the first two executions.

When the various replicas are executed on the same PE in subsequent time frames, time redundancy is exploited. This kind of approach effective for systems with no hard real-time constraints and against transient faults. Compared with the other forms of redundancy, time redundancy has much lower hardware and software overhead, but incurs in high performance penalty [61].

Information redundancy is a third redundancy technique that is not directly exploited in this thesis, but will be mentioned with reference to error detection and correction coding. Here, additional information (e.g. bits) is added to the original data so that an error can be detected or even corrected. Error-detecting codes (EDCs) and error-correcting codes (ECCs) are widely used in memory units and various storage devices.

2.1.3 Self-Adaptive Systems

In the last few years, the growing trend in the architecture complexity, as in the number of processing element per chip and their specialization, has raised the need for a further abstraction layer between the hardware and the programmer to hide the complexity of the underlying system by introducing the self-management of its resources. Taking into consideration self-adaptation capabilities while designing new computing systems allows developers to ignore to a certain extent parallelism, energy efficiency, reliability and predictability issues. Implementing self-adaptivity implies adopting the *dynamic optimization paradigm*. Dynamic approaches can adapt the behavior of the system they govern to cope with evolving environments (changing resource availability, unknown application scenario, varying requirements and constraints, etc.). Their main drawback is that they usually suffer from execution time overhead or partial solution space exploration (which means sub-optimal solutions). On the other hand, the usual lack of a complete description for the forthcoming execution scenario represents the key limitation of static approaches. Moreover, even in the case in which all the possible execution scenarios are known in advance, it could be unfeasible (in terms of execution time) to exhaustively explore the whole solution space.

In order to build computing systems that show self-adaptive capabilities, a specific design paradigm is to be adopted. This paradigm is a classic close loop approach; a computing system designed and implemented around this solution is said to harness an autonomic control loop. In fact, it is characterized by a recurrent sequence of actions that consists in gathering information about the internal state of the system and the environmental conditions (achieving self- and context-awareness), detecting possible issues with respect to the objectives and constraints,

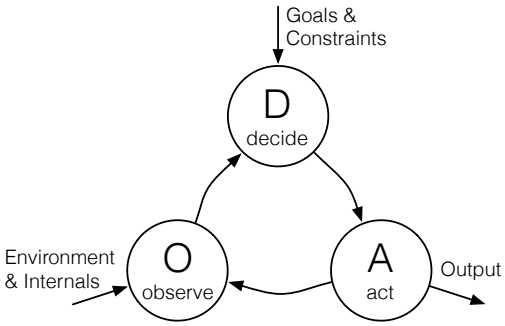


Figure 2.3: The Observe-Decide-Act (ODA) control loop.

eventually deciding a set of corrective actions to be performed and then applying them.

Various definitions and descriptions of this control loop can be found in literature: from the one given in [46] emphasizing the separation between the detection and decision phases, to the one dubbed MAPE-K (Monitor-Analyze-Plan-Execute with Knowledge) [60], which focuses on the shared knowledge each phase of the control loop contributes to build. The Observe Decide Act (ODA) loop [86], shown in Figure 4.6, represents a third version of the autonomic control loop. Even if it can be considered equivalent to the other interpretations, the ODA loop better captures the essence of autonomic computing, by clearly dividing the system design in three simple and sharply distinct stages.

Observe-Decide-Act

The Observe-Decide-Act abstraction defines a protocol consisting of three phases guiding the design of a self-adaptive system.

- The *observation* phase consists in sensing both the external environment and the internal behavior of all the sub-systems in order to maintain and update information about the state of the system. The sensing task is accomplished by monitors; it can be done directly (through thermometers, voltmeters or any other device able to directly measure the dimension of interest) or indirectly (when a direct measure is not available and the dimension is to be estimated trough a mathematical model).

- The information gathered in the observation phase must be exploited to create knowledge and make smart choices on how to

correct system's parameters: this is the aim of the *decision* phase. This phase is performed taking into account the data obtained by the monitors, possibly considering the past iterations of the autonomic control loop, and an high-level goal. The knowledge of the goal guides the logic of the system in coming up with a suitable decision, a set of actions, which should approach the state of the system to the desired one. The decision making process is usually carried out by adaptation policies, based on high-level objectives and constraints.

- Once the decision has been taken, it is put into practice in the *action* phase through the actuators. Actuators, that can be either physical or virtual, are able to modify some system parameters, or knobs, in order to alter its behavior.

A central entity in charge of coordinating the three phases is also defined. Throughout all this thesis it will be named *orchestrator*, but it is also known in literature as coordinator or central governor [79]. The aim of the coordinator is to extract knowledge from the data gathered by the monitors; to understand whether the current status of the system reflects the desired one or not; if this last is the case, it must identify which are the best actions to be taken to bring the system closer to the desired state, possibly exploiting the results achieved in previous iterations.

2.2 Working Scenario

In this section we exploit the previously introduced background concepts to define the working scenario. Furthermore, we will introduce the baseline for comparing the outcomes of our proposal, in terms of a set of case studies presented and a list of key performance indicators (KPIs).

The aim of this work is to improve dependability of multi/manycore architectures through system-level techniques, considered both in terms of availability and reliability. Availability is mainly analyzed (but not only) when coping with transient and intermittent faults, that cause system's components to fail for limited intervals of time; in this case, redundancy techniques are exploited to detect and mitigate faults, not to impact on the required reliability level. On the other hand, system-level techniques can be exploited to extend components lifetime, i.e. the overall system's reliability. Both availability and reliability will be analyzed with respect to the entire system, being it a multi/manycore architecture.

2.2.1 Reference Platform

The reference architecture we adopted is a manycore one consisting of varying number of nodes (e.g. 2×2, 3×3, 3×4 depending on the experiments) connected through a NoC infrastructure. All the nodes in the architecture are homogeneous, i.e. they expose the same hardware characteristics. Such a model is implemented by several real examples, such as the ST/CEA P2012 platform [93] or the Teraflux one [94]. The computational resources within a node are heterogeneous and organized as in the ARM big.LITTLE [3]: a *LITTLE* cluster, consisting of two Cortex-A7 cores, and a *big* one, consisting of two Cortex-A15 cores. Communication among cores and clusters within a node is made possible through a bus and the shared-memory programming model is considered.

The reference architecture is assumed to be equipped with per-cluster energy monitors (to derive energy and power consumption) and per-core temperature and wear-out sensors [43]. We also assume dynamic voltage and frequency scaling (DVFS) to be available per-cluster; this means that all the cores within a cluster run at the same frequency [3]. The reference architecture can seamlessly migrate applications across the clusters within a node during runtime, thanks to the shared memory architecture. This allows the dynamic mapping of applications to the available cores at runtime. According to the adopted fault model, we assume that data belonging to different threads of execution are organized in separate segments of the memory and no thread can access the segment of any other thread; watchdogs are used for terminating tasks in infinite execution loop.

The whole system's activity is coordinated by a special node, named *fabric controller*. This node is in charge of dispatching the applications to the available nodes and it is connected to the NoC infrastructure through a special link. We assume it is hardened by design so to achieve fault tolerant properties, and it is assumed not to influence the thermal profile of the system. Each node contains also a *controller core*, provided with a (minimal) operating system: it functions as an interface towards the rest of the platform and performs the boot of the issued applications (the execution of an application is currently limited within one node). The controller manages thread synchronization and scheduling as well. In specific scenarios, the role of the controller core can be played by a regular core or even distributed among the other cores. A graphical representation of the overall reference platform is given in Figure 2.4.

Even if validated on the described platform, the reliability framework proposed in this thesis has been designed to be adaptable to the underlying architecture. A two-layer heterogeneous architecture allows to obtain

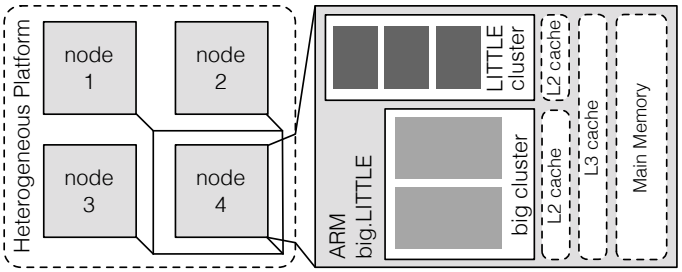


Figure 2.4: Graphical representation of the reference platform.

the best from our framework. However, it can be exploited to optimize the reliablity/performance/energy consumption trade-off even when the available architecture misses some features. For example, a single-layer architecture won't implement the communication energy optimization layer, but it will benefit from the remaining multicore optimization layers. Again, in a double-layer homogeneous architecture it won't be possible to select the proper execution resource on the basis of the application phase, but it will be still possible to optimize the energy consumption by minimizing NoC consumption at the top level and through the dynamic selection of the resources execution frequency within the nodes.

2.2.2 Baseline

The definition of a basis for comparison is pivotal to properly evaluate the advantages of any proposed solution. In the context of this work, two baselines are defined for dealing with transient faults and aging phenomena, respectively.

The baseline for transient faults management is a multicore system with a single node platform, a bus-based communication model and a shared-memory architecture. The controller core is in charge of managing applications mapping and scheduling and to provide fault management capabilities to the system. An off-line exploration phase is run for selecting the most suitable reliable technique to be applied, based on the information available at design-time. Since a safety-critical scenario is considered, the reliable technique chosen for the baseline system is TMR, which guarantees a complete fault coverage. This technique is applied during the whole system's lifetime to all the running tasks, without the possibility to adapt it at run-time.

When considering mapping and scheduling techniques for aging and wear-out mitigation, the baseline to be considered is a two-level architec-

Managed Aspect	Architecture	Baseline Technique
Transient faults	Multicore	TMR on every task
Wear-out & Aging	2-level Manycore	Ubuntu Linaro Scheduler

Table 2.1: Baseline summary.

ture with several nodes, each node consisting of an ARM big.LITTLE processor. The running operating system implements a Linux scheduler supporting multicore architectures, such as the one included in the Ubuntu Linaro distribution [68]. Such a scheduler is aware of the applications performance and schedules them to optimize the exploitation of performance/power consumption trade-off. However, it is not aware of the cores aging status and its scheduling decisions are not made to maximize the architecture lifetime.

The characteristics of the chosen baseline are summarized in Table 2.1.

2.2.3 Key Performance Indicators

Key performance indicators define a set of values against which to measure the effectiveness of the proposed methodology if compared to the given baseline. The indicators chosen for evaluating the work described in this thesis are listed below, together with a brief description.

Applications Throughput Throughput is defined as the amount of data processed by the system in a unit of time and represents a metric for performance estimation. In particular, throughput is computed as the ratio between the amount of processed data and the time elapsed between the execution start and end times. Thus, it is measured in bits per second $[bit/s]$. The throughput of an application should be as close as possible to a defined threshold: a lower throughput can cause the subsequent data processing tasks to stop or poor outcomes to be produced; a throughput higher than strictly required, on the other hand, means an unnecessary use of computational power and of energy. While in the past performance has been adopted as an optimization goal, as the number of processing resources increases, making it usually not too difficult to achieve good performance, throughput has become a constraint to be fulfilled while optimizing other aspects (e.g. reliability and/or energy consumption).

Architecture Lifetime Architecture lifetime estimates how long a system can work while meeting the given constraints; lifetime is usually

measured in years [y]. We are interested in evaluating the expected lifetime of the overall architecture, not the one of single cores.

Error Rate The error rate the frequency at which the considered system produces incorrect results. We will measure it as the ratio between the number of wrong bits in the output and the overall number of output bits. Being a ratio, it is a dimensionless number representing the percentage of incorrect result.

Architecture Energy Consumption Energy consumption measures how much energy is used by the considered architecture and is measured in Joules [J]. The used Joules is the integral over time of the instantaneous power consumption.

2.2.4 Case Study

The evaluation of the proposed solutions will be carried out by considering a workload consisting of several applications with different performance/reliability constraints. The applications are kernels performing image/video processing. We envision this kind of workloads running on (embedded) systems used in (as two examples) 1. the medical environment (in a surgery room) with stringent reliability requirements, 2. scientific computing where eventually reliability requirements may vary based on the user's profile. A brief description of the basic kernels follows, while a more specific characterization of the workloads characteristics will be introduced later on in the thesis.

- *Edge Detection:* is the name for a set of mathematical methods that aim at identifying points in a digital image at which the image brightness has discontinuities; these points are typically organized into a set of curved line segments termed edges.
- *Motion Detection:* is the process of detecting a change in position of an object relative to its surroundings or the change in the surroundings relative to an object.
- *Erosion Filter:* is a morphological filter that changes the shape of objects in an image by reducing, or eroding, the boundaries of bright objects, and enlarging the boundaries of dark ones; it is often used to reduce, or eliminate, small bright objects.
- *RGB to Grey:* is a simple algorithm for converting RGB color images into greyscale ones.

- *Gaussian Filter:* is a two dimensional convolution operator that uses a Gaussian function for calculating the transformation to apply to each pixel in the image; it is usually used to blur images and remove detail and noise.
- *Template Matching:* is a technique in digital image processing for finding small parts of an image which match a given template image.

These tasks are usually characterized by high parallelization, since the processing of each pixel is independent from the other ones, or depends only on the very near ones. These basic operations can be individually used or combined to obtained more complex functionalities. Both the cases are considered in this work.

In this chapter we have introduced the basics related to dependability and self-adaptiveness. The next chapter focuses on the system architecture we propose to define a self-adaptive dependable system, formalizing the model to design and implement such kind of systems.

3 Self-Adaptive Systems Design

This chapter introduces the core of a self-adaptive system, the orchestrator, in charge of managing the available resources in order to achieve the goals set by the user. Based on its functionality we developed a formal to define, design and implement self-adaptive systems.

3.1 Orchestrator

In a self-adaptive system, the orchestrator is the entity that implements the observe-decide-act paradigm. It plays a central and fundamental role in allowing the system to have self-adaptive capabilities. A graphical representation of the orchestrator and of the context it works in is shown in Figure 5.13.

The orchestrator can perform its job provided that some input data are available. In particular:

- a description of the underlying architecture, in terms of quantity and quality of the processing elements, and its topology. For each PE, its operating points (frequency – voltage) are to be listed and a characterization of its performance and energy consumption is to be provided. The topology of the architecture is to be described as well: the coordinates of each node in the network (if any) and its composition (how many clusters and PEs in each cluster). This kind of information is available at design-time, even if it can be necessary to update it at runtime in case, for example, if a occurs and a resource is temporarily or permanently damaged;

- a characterization of the workload that the architecture will execute. It is not realistic to assume that the arrival times of all the applications are known a-priori; however, it is reasonable to assume that the applications that are going to be executed are known in advance. This allows the system to perform an off-line profiling phase and get an insight of the duration of the application's tasks and possibly other information. Thus, part of these data is supposed to be available at design-time (by means of a profiling phase), while other will be gathered at runtime (e.g., the applications' arrival times);

- a set of user's defined goals and constraints. Goals are those quantities that the system must optimize during its execution (e.g., performance); constraints are, on the other hand, quantities that the system must keep above or below a certain threshold (e.g., throughput, energy consumption). It is the user who will set these and they can change at runtime.

All the other information regarding the system's status and the surrounding environment will be gathered during the observation phase through the *monitoring infrastructure*. Knowledge is to be extracted from these data and manipulated using suitable *decision policies*; indeed, another task of the orchestrator is to select the best decision policy to properly exploit the available *actuating elements* and obtain the desired results. These three categories will be further investigated in the following paragraphs.

3.1.1 Monitoring Infrastructure

The monitor infrastructure allows the observation phase to take place. It consists of a set of monitors and the infrastructure needed to communicate the sensed values to the orchestrator. A *monitor* is a component that is in charge of periodically sensing the value of a specific quantity. Monitors are classified as *real* or virtual *ones*. The former ones are physical components that can directly read the value of the quantity they measure; a thermometer is a real monitor for measuring temperature, voltmeter is a real monitor for electrical voltage. When real monitors are not available or cannot be used, mathematical models can be exploited to estimate the value of the parameter of interest. As an example, temperature sensors are often virtual, as they are not typically available on boards. Rather temperature is computed starting from a model that takes into consideration the component self-activity and the neighbors load. While real monitors provide direct measures, virtual monitors represent an indirect measure since they rely on different quantities to compute the value of the desired one. As said, mathematical models for temperature estimation are widely used in literature [37], as well as aging ones [90].

Performance, power/energy consumption, reliability and aging are the quantities of interest for this research. Depending on the execution scenario, *performance* will be measured through a real or a virtual monitor. Application Heartbeat [45] can be considered as a real monitor: it can measure the progress of an instrumented application based on the number of heartbeats it emits; the higher the number of heartbeats, the higher the performance of the application. In other contexts, where it

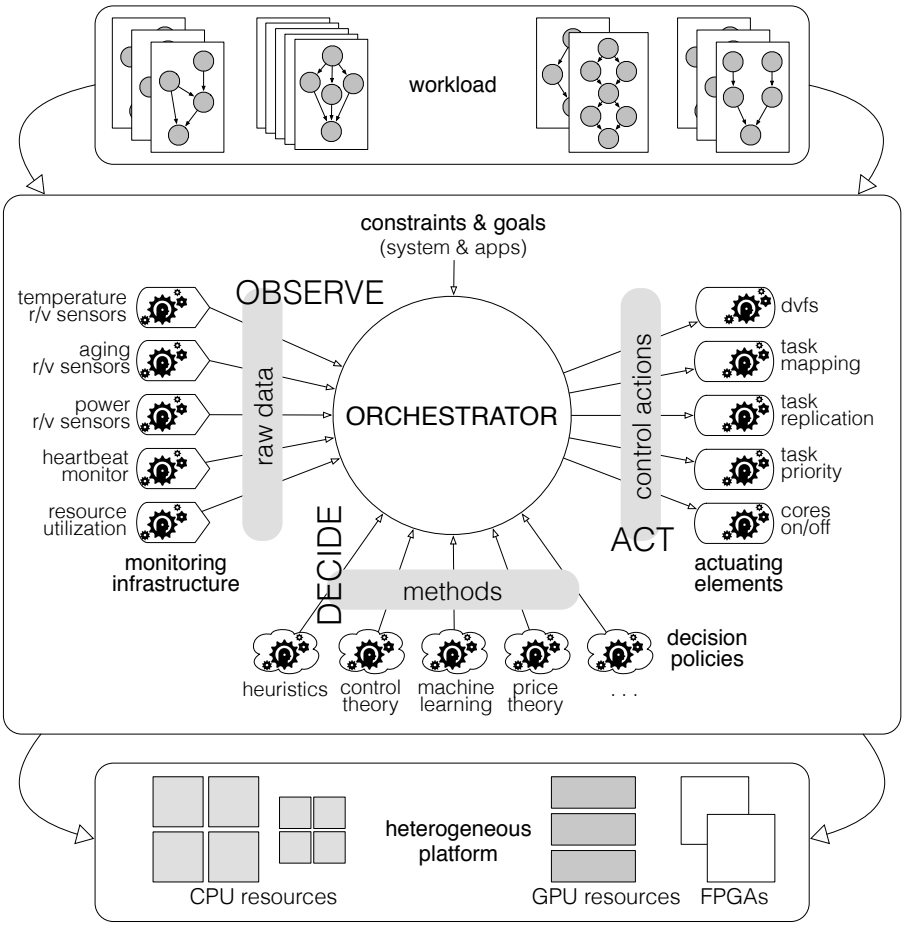


Figure 3.1: Overview of the proposed framework.

has not been possible to exploit the heartbeat mechanism, performance has been measured indirectly by looking at the throughput, i.e., the amount of data processed by the application in a unit of time.

Power and *energy* are usually measured through real sensors. In fact, it is quite common for development boards (as the ones used in the experimental sessions [3]) to be equipped with sensors for measuring power consumption. Since power consumption is defined as energy used per unit of time, energy consumption can be directly and straightforwardly computed from the power data.

There are no real sensors for measuring *reliability*, but several metrics for its estimation exist. We measured reliability as error rate, i.e., the percentage of erroneous data in the output over the overall processed data.

Components *aging* represents the hardest quantity to be evaluated. Even if real aging sensors have been designed [15], their diffusion is so limited that virtual sensors are often used. The work described in this thesis relies on a aging model taken from literature [90] and capable of describing the aging of each component, given the phenomena of interest and its temperature.

3.1.2 Decision Policies

Several techniques have been used in literature to enable the self-adaptive computing systems to make smart decisions; all of them are considered in the decision policies groups. The orchestrator can choose among the available techniques how to manage the available actuating elements to pursue the set goals and meet the constraints. A brief overview of the most common decision policies, taken from [70], is provided in the following paragraphs.

Heuristic solutions start from a guess about the system behavior and subsequently adjust this guess. They are designed for computational performance or simplicity at the cost of potential loss in accuracy or precision. Such solutions generally cannot be proven to converge to the optimum or desired value.

Control-based solutions employ canonical models, such as discrete-time linear models and discrete event systems, and apply standard control techniques such as Proportional Integral (PI) controllers or Proportional Integral and Derivative (PID) controllers. Assuming the model to be correct, some properties may be enforced, among which stability and convergence time are probably the most important ones, thereby providing formal performance guarantees. More complex control-based solutions exist, requiring complex models and advanced control techniques,

but they will not be exploited in this thesis.

Another interesting category of decision policies is based on *machine learning* techniques; they describe a framework that can be exploited to learn system behavior and adjust tuning points online. This solutions are extremely appealing, in particular in the case of self-adaptive systems; they are not directly treated in this work, but represent an attractive future development.

3.1.3 Actuating Elements

The last category of elements the orchestrator must interact with is the actuating elements one, which corresponds to the act phase of the ODA loop. In a self-adaptive systems, the available actuating elements, also known as *knobs*, are strictly dependent on the underlying architecture characteristics and the implemented software layers.

Architecture dependent knobs that will be exploited in the proposed self-adaptive systems include: DVFS, which allows to dynamically change the processors working points in terms of voltage and frequency, and the possibility to power on and off single PEs or whole clusters. More knobs are available at a software level: tasks and applications mapping allows to select the PEs on which workloads are executed; the selection of the reliability technique to be applied allows to exploit the reliability/performance trade-off according to the current needs.

The overview of the system provided in the first part of this chapter highlights the complexity of the elements composing a self-adaptive computing system and thus the need for defining a model of context in order to deal with them in a more meticulous way; this is the aim of the remainder part of the chapter. In particular, the presented model has been proposed in [21].

3.2 A Model for Self-Adaptive Computing Systems

While context/self-awareness and self-adaptation are common denominators for some of the most recent research topics in the computing systems domain, the idea of rigorously modeling the concepts at the basis of these topics have often been neglected or overlooked, leading to several different partial definitions. Indeed, while in different research areas (e.g., database [6] and the software engineering [44] areas) the concept of context and self-adaptiveness has received a lot of attention

in other communities we witness a direct exploitation of an adaptive, context-aware behavior, implicitly resorting to a model of context without its precise introduction. In fact, it is possible to find in literature many examples of approaches where the context plays a fundamental role without being formalized. This is the case for research projects related to the reduction of power consumption in mobile devices [85], to the optimization of resource usage in various computing systems [86], to the thermal control of sensitive environments [5], to the smart management of reliability in critical systems [18], and so forth. On the other hand, a notable contribution within the self-aware/adaptive area is presented in [9], where the authors identify the fundamental elements of a self-aware system as *public self-awareness*, *private self-awareness* and *self-expression*. The work poses the basis for an architectural framework for the representation and design of self-adaptive systems, but it only provides a preliminary general overview, not followed by a rigorous and systematic definition.

Even if the lack of a formal, rigorous model may not represent a limitation when the number of possible contexts is somehow small and when the real adaptive features are only a few, as the number of elements determining the *context* grows and the adaptive behaviors become more and more refined, the quest for a flexible and powerful support arises. Indeed, it is paramount to define what situations may arise, which one(s) should be avoided, what actions need to be taken (for both normal adaptation and/or self-healing management), and what configurations should be adopted in each situation. Part of these issues are informally analyzed at design time, implicitly, while defining the adaptive part of the system; others may pass unnoticed. Thus, last but not less relevant, a rigorous model can contribute to specification and documentation. Therefore, the designer could benefit from a rigorous approach in the modeling phase. Moreover, we envision the model at the basis of a framework to be used as a design/validation instrument to support the designer in his/her choices and in defining the skeleton of such self-adaptive systems.

When considering a self-adaptive computing system, it would be interesting to be able to describe it in a complete, correct, and robust way. The first aspects to be considered are the *elements* that come into play, paying particular attention to the ODA step we are considering.

- The *Observation* phase is in charge of observing high-level elements, that may represent an aggregation of lower-level quantities computed on raw direct/virtual observations of system and environment.
- The *Decide* phase must know which are the aspects to reason on,

i.e., the constrained elements and the ones to be optimized.

- For the *Act* phase, it is necessary to know the parameters on which the system can act and the strategies to manage them.

Elements alone do not suffice for providing a description of the system as the one we desire: *relations* between them are equally crucial. In fact, when making a decision on how to react to changes in the context, it is important to know the quantitative values of the observed elements, *and* the effects of the planned actions. The possibility to have indirect relations must also be taken into consideration.

The envisioned framework should be able to capture the contextual information typical of as many self-adaptive computing systems scenarios as possible. To this end, a suitable model must support the representation of all information that the self-adaptive system will use to (re)act, also highlighting the relation between the various parameters and data, and possible requirements and constraints. Do note that the flexibility of the model is related to the number of different contexts it is able to capture and represent, not to the number of contexts a given system will possibly experience/manage. Moreover, while we will aim at providing a model that is as complete as possible, the adopted approach should be flexible enough to allow the addition of elements that have been omitted or that might become of interest in the future. As a result, the output of the modeling process is the means to:

- help the computing systems designer in understanding what resources are needed to be able to pursue its goal (e.g., performance optimization, energy minimization, ...);
- document the system to provide a common and systematic classification of self-adaptive computing systems;
- describe, at run-time, the current *context*;
- provide an active support to system development by automating some crucial step (e.g., the creation of standard rule-based decision engines).

To exemplify the proposed model, we first introduce a running example, for an immediate application of the introduced concepts. Among the various self-adaptive systems proposed in literature, a "standard" one has been selected to exemplify the context model concepts. *METE* [84] implements a framework for QoS improvement in multi-core platforms through system-wide resource management. The aim of this system is to

satisfy a performance constraint specified by each application, by adapting to the changing context of the system. Self-adaptation consists in dynamically provisioning sufficient on-chip resources to meet the constraint: cores, shared caches and off-chip bandwidth. METE employs a feedback-based system to manage these knobs, designed as a Single-Input, Multiple-Output (SIMO) controller with an Auto-Regressive-Moving-Average (ARMA) model to capture the behavior of different applications, pursuing an optimized resource utilization goal.

3.2.1 The Context Concept

Let us first look at the *model* from an informal point of view, to identify all elements that contribute to the self-adaptive behavior, and then formally define the context meta-model. This definition derives from the analysis of the peculiar aspects characterizing existing self-adaptive systems, from the investigation of existing context models from different application areas, and from the significant elements identified by other studies in this same area. The *meta-model* is also to be defined: it captures and allows to represent all and only the relevant information for the definition and realization of a self-adaptive computing system.

Context categories

A *context dimension*, or simply dimension, is defined as any element such that when a change occurs in its values, the system will change its behavior and adapt (as in [23]). Eventually, when envisioning a proactive self-adaptive behavior, the system may also anticipate changes in the context dimensions and thus evolve autonomously as a look-ahead adaptation. In this perspective, all elements referring both to the system itself and to the environment (including the user's inputs and the application workload) are included in the unique *context* concept.

The three categories of context defined in [83] have been extended to four, for a more refined differentiation. The *functional* context includes all those elements characterizing the overall primary functionality the system has to provide, in a self-adaptive fashion, that is the possible *goals* the system tends to, the *requirements* or constraints it has to fulfill, and eventually some additional variables it can observe, to have a better understanding of its context.

The *application* context is related to the characterizing elements of the workload being executed on the system, which might change over time, in terms of the kind of applications, their criticality (i.e., the request for fault tolerant execution), and the size of the workload itself. There are

situations where the application context is rather "static" (such as the embedded systems application scenario) and others where the workload is not known in advance and varies a lot (such as in today "on-demand computing" scenario).

The *architectural* context covers all those elements related to the system itself, in terms of the characteristics that may change during the system lifetime and such changes would affect the system behavior. These aspects are mainly related to the resources constituting the system architecture and their status; examples are the amount of used resources, their availability due to the occurrence of failures or to a dynamic architecture where the number of nodes is not fixed a-priori (a cloud providing a number of resources based on the workload/user).

Finally, the *environmental* context gathers all relevant elements characterizing the ambient the system interacts with, that have impact on its behavior. This context is crucial when considering cyber-physical systems (which usually interact with other systems) or dangerous environment (e.g., radiations exposure).

Context Dimensions, Domains and Relations

The envisioned context dimensions belong to one of the above mentioned categories, and cover all elements having an impact on a self-adaptive system behavior; in the following a brief explanation for each one of them is provided, introducing their relations as well.

- *goals.* When designing a self-adaptive system, the user may express one or more goals to be pursued in terms of an optimization function (performance) or as a trade-off between different contrasting objectives (e.g., high performance vs. energy consumption). With respect to the selected running example, METE pursues an optimal resource exploitation goal.

- *requirements.* This dimension models the functional constraints the system must satisfy; it can be both a strict constraint (e.g., throughput must not decrease below a given threshold) or a soft one (e.g., throughput should be within a given window). In the METE example, there is a requirement on the system performance, set on the weighted number of Instructions Per Cycle (IPC).

- *observations.* An additional dimension has been introduced to model those elements that constitute an interesting facet of the context, worth observing although it might not directly fall into one of the first two dimensions, since they are neither optimized

nor constitute a constraint. In the running example no observation elements are identified.

The next set of dimensions captures the information on the collected data from real as well as virtual sensors, that are opportunely aggregated, weighted and abstracted to be feed the self-adaptive engine acting on the perceived part of the context. They represent the data sensing and elaboration sphere and include all elements constituting the private and public self-awareness defined in [9].

- *raw data.* This dimension gathers all collected information, represented at the lowest abstraction level, directly coming from the observation of a physical phenomenon through real and virtual monitors. The METE scenario collects information on the number of instructions being executed and on the resources being allocated to each application. Do note that the identification of the raw data to be collected allows the system designer to focus the attention of the necessary sensors that should be available in the architecture in order to access that specific kind of information. In the running example no specific physical sensor is necessary.

- *measures.* This dimension models the first refinement of raw data. The same raw data can be aggregated to define different measures, that constitute a more abstract view of the observed variables. The measures adopted in METE are the number of Instructions Per Cycle (IPC) and the overall number of allocated resources. A measure uses a number of raw data that varies from 1 to n: if more than one data is combined, then *all* of them are necessary to compute the measure. This aspect will be clarified soon.

- *metrics.* This dimension represents a further abstraction of the measures and the raw data, to model how the contextual information is weighted and combined to define either the optimization function or the constraints exploited by the goals, requirements and observations dimensions. Indeed, the same measures combined in a different metric lead to a different behavior, therefore this dimension can further refine the contextual characteristics of the system being designed. Some metrics may be particularly simple and directly exploit raw data, without any specific aggregation/manipulation. Again, a metric can refer to 1 to n measures; if more measures are specified, then *all* of them are necessary. The metrics adopted in METE are the weighted IPC and resource utilization.

The last two dimensions model the actions the self-adaptive system can control and the methods or strategies that are exploited to drive such actions. These dimensions, with respect to the EPiCS framework [9], pertain to the self-expression of the overall system.

- *methods.* This dimension models the general policies the self-adaptive engine may enact to pursue a goal or to fulfill a requirement. These methods need be provided and be made available. With reference to the running example, the exploited method is the feedback loop made up of the SISO controller and an ARMA model.

- *control actions.* A control action is the elementary action that can be performed by acting on a knob, expressing the physical or virtual actuators that are available (or need to be) in the system architecture. In METE the feedback loop method takes as input the value of the QoS metric and outputs three control actions on the available system knobs: number of cores assigned to an application, cache dimension and off-chip bandwidth size. The same control actions may be exploited by different methods, as well as a method may need more than one (at least one) control action.

It is worth highlighting a few considerations. Given a self-adaptive system, not all dimensions might be of interest, such that in the specific scenario one dimension may not exist or not have an impact on the behavior, either because there is no cause-effect relation or the system is not able to observe such aspect/phenomenon or is not designed to react to it. There might be other dimensions that have not been here listed; the proposed model is flexible and supports any number of dimensions. Dimensions goals and requirements are the most important ones in the design of the system, where everything else is exploited in order to pursue the goal and satisfy requirements. Therefore they have been identified as *driving dimensions*, referring to the others as *secondary dimensions*. The self-adaptive engine implementing the ODA loop and the necessary intelligence, although having a state itself is not part of the context model.

The set of presented dimensions actually defines a *context meta-model* for a self-adaptive computing system; it provides a general-purpose framework to represent any system according to the listed dimensions (or to a subset of them). A graphical representation of the meta-model is shown in Figure 3.2, where the driving dimensions are shown with rounded corner rectangles. Indeed, once we consider a given self-adaptive computing system, such as METE, we have specific values for the dimensions,

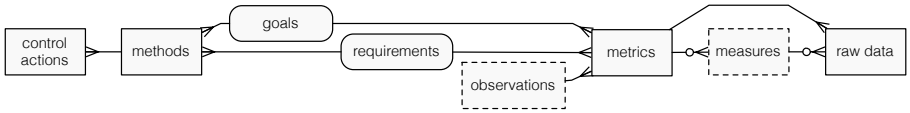


Figure 3.2: Graphical representation of the context meta-model; each rectangle represents a context dimension, those with rounded corners are the driving dimensions. Dotted rectangles indicate dimensions that might not exist in some contexts. Segments represent relations among dimensions.

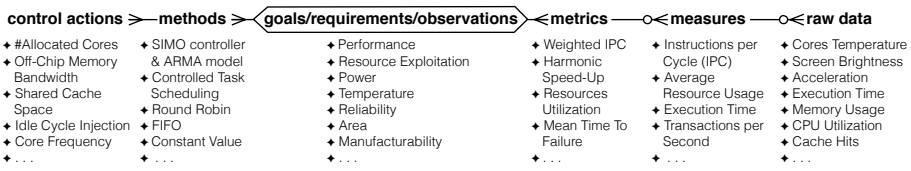


Figure 3.3: Context meta-model and dimension domains for self-adaptive computing systems.

those we mentioned through the presentation. Therefore, for a given self-adaptive computing system, the *context model* is defined as a set of (dimension,value) pairs.

Dimensions domain

Each one of the identified dimensions, as well as any additional one the designer might introduce, has a domain of values whose selection will define the specific context model. A number of self-adaptive systems in the literature has been surveyed (namely, [84, 85, 86, 5, 18]) to extract some of the most common values of the domains for the listed dimensions, presented in Figure 3.3. Each domain is extensible according to the peculiarities of each working scenario. It is worth noting that the domains for the goals, requirements and observations coincide (and for clarity reasons have not been replicated in Figure 3.3, but listed once, in a shared fashion). In fact, the specific goal being pursued in a self-adaptive computing system (e.g., performance optimization) can be a requirement for another self-adaptive computing system (e.g., a minimum performance level).

Relations among dimensions

As shown in Figure 3.2, relations exist among the dimensions, expressing dependencies among values, such that "given a system, when a dimension value exists, a related value, or set of values, must exist in another dimension for the system to be consistent". For example, in METE the IPC measure is function of the number of instructions and the elapsed time, that are raw data. In particular, both data are necessary to compute the measure; an AND operator characterizes the relations between measures and raw data dimension values, as well as between metrics and measures. As another example, the same method adopted for pursuing a goal can use one or more control actions, therefore in this case, an OR operator characterizes the relations between the methods and control actions dimension values.

As a result, the context meta-model allows for the representation of relations among dimensions, and the context model for relations among dimension values, also supporting the AND/OR constraint definition. In the graphical representation, relations are segments, with a single end on one side and a multi-point end on the other, expressing a 1 to $n \geq 1$ cardinality; a circle on the segment characterizes an AND constraint.

All these aspects are captured in the following formalization of the context meta-model and model for self-adaptive computing system.

3.2.2 Formalization

The proposed context meta-model is based on the notions of *dimension* and the *relations* between different dimension values.

The set of context dimensions $\mathcal{CD}$ is partitioned into two disjoint subsets: the set $\mathcal{DD}$ of *driving dimensions*, and the set $\mathcal{SD}$ of secondary dimensions, whose values, in each model, can depend on the values assumed by the driving dimensions. The two identified sets are:

$$\mathcal{DD} = \left\{ \texttt{goals}, \texttt{requirements} \right\}$$

$$\mathcal{SD} = \left\{ \begin{array}{l} \texttt{methods}, \texttt{control actions}, \\ \texttt{observations}, \texttt{metrics}, \\ \texttt{measures}, \texttt{raw data} \end{array} \right\}$$

Each dimension has an associated domain that depends on the model. A *context* is a set of pairs $C = \{(d_1, v_1), \ldots, (d_k, v_k)\}$ (with $k \geq 5$, i.e, at least one among `goals` and `requirements`, then `methods`, `control` actions, `metrics`, and `raw data`), where each $d_i \in \mathcal{CD} = \mathcal{DD} \cup \mathcal{SD}$ is a dimension and v_i its value, chosen from the domain $\mathcal{V}(d_i)$.

Each context C has to satisfy the following general constraints (leading to $k \geq 5$):

1. $\forall C((\texttt{goals}, v_1) \in C \vee (\texttt{requirements}, v_2) \in C)$;
2. $\forall d_i \in \mathcal{SD}$ s.t. $d_i \neq \texttt{observations} \wedge d_i \neq \texttt{measures}\ \exists (d_i, v_i) \in C$.

The first constraint imposes that at least one dimension between `goals` and `requirements` exist in each context, a fundamental aspect to define a self-adaptive computing system. The second constraint imposes that for each context, each dimension and an associated value must exist, except for `observations` and `measures`. In our graphical representation of the context meta-model, the `observations` and `measures` dimensions are represented with dashed lines, because their presence in the context is not mandatory.

Given a self-adaptive computing system, its **Context model** has to specify the domain for each dimension in $\mathcal{CD}$. For example, if we consider the METE research project, the domains for the `metrics` and `measures` dimensions are:

$$\mathcal{V}(\texttt{measures}) = \left\{ \begin{array}{l} \texttt{Instructions Per Cycle (IPC)}, \\ \texttt{Average Resource Usage} \end{array} \right\}$$

$$\mathcal{V}(\texttt{metrics}) = \left\{ \begin{array}{l} \texttt{Weighted IPC}, \texttt{Harmonic Speed-Up}, \\ \texttt{Resources Utilisation} \end{array} \right\}$$

In general, $\mathcal{V}(\texttt{metrics})$ contains also ad-hoc functions defined in each specific model. Moreover, for each model some model-dependent constraints must be defined, with the following forms:

$$\text{if } (d_i, v_i) \in C \text{ then } \bigwedge (d_j, v_j) \in C$$

when the constraint specifies a relation between `measures` and `raw data`, or between `metrics` and `measures`,

$$\text{if } (d_i, v_i) \in C \text{ then } \bigvee (d_j, v_j) \in C$$

otherwise. These relations allow the specification of what elements contribute to making the system self-adaptive and how, from the `raw data` to the `control actions` the objectives are achieved. In particular, relations among dimension values model what elements can be found in a self-adaptive computing system for a given context. To exemplify these concepts, we report for the METE project, in a given context C, three relationships:

- if (`requirements`, `Performance`) $\in C$
  then (`metrics`, `Weighted IPC`) $\in C$
  $\vee$ (`metrics`, `Harmonic Speed-Up`) $\in C$.

- if (`goals`, `Resource Exploitation`) $\in C$
  then (`metrics`, `Resource Utilization`) $\in C$.

- if (`metrics`, `Resource Utilization`) $\in C$
  then (`measures`, `Average Resource Usage`) $\in C$.

The first relationship states that requirement `Performance` uses either the `Weighted IPC` or the `Harmonic Speed-Up` metric (but not `Resource Utilization`). The second relationship states that when pursuing the `Resource Exploitation` goal, the `Resource Utilization` metric is adopted (and no other available metric). Finally, the last relationship defines the link between the `Resource Utilization` metric and the measures it uses.

3.2.3 Validation

The complete application of the context model to some self-adaptive systems taken from the literature is now presented as a proof of concept for showing its effectiveness.

System-wide Resource Management in Multi-Cores

Considering, as a first case study, the running example, the METE self-adaptive system can be modeled as shown in Figure 3.4, where the context dimensions and domains are presented. Performance is monitored through the weighted IPC and harmonic speed-up, both referring, with different semantics, to the IPC number, computed using the executed instructions count and the execution time. Resources exploitation is evaluated in terms of the resources usage, computed by accounting the number of used resources at each time. We can see how the available system knobs are the number of cores, the memory bandwidth and the shared cache space allocated for each application.

The graphical representation of the context model in terms of the appropriate dimensions, domains and relations/constraints is reported in Figure 3.4. The corresponding formal textual description is reported for the list of relations/constraints (Table 3.1), while the dimensions and domains definition can be straightforwardly inferred, following the previously given definitions.

if $(\texttt{requirements}, \texttt{Performance}) \in C$ then $(\texttt{metrics}, \texttt{Weighted } IPC) \in C$ $\vee(\texttt{metrics}, \texttt{Harmonic Speed-Up}) \in C$.
if $(\texttt{goals}, \texttt{Resource Exploitation}) \in C$ then $(\texttt{metrics}, \texttt{Resource Utilization}) \in C$.
if $(\texttt{metrics}, \texttt{Resource Utilization}) \in C$ then $(\texttt{measures}, \sum \texttt{Allocated Cores}) \in C$ $\wedge(\texttt{measures}, \sum \texttt{Off} - \texttt{Chip Memory Bandwidth}) \in C$ $\wedge(\texttt{measures}, \sum \texttt{Shared Cache Space}) \in C$.
if $(\texttt{metrics}, \texttt{Weighted IPC}) \in C$ then $(\texttt{measures}, \texttt{IPC}) \in C$.
if $(\texttt{metrics}, \texttt{Harmonic Speed-Up}) \in C$ then $(\texttt{measures}, \texttt{IPC}) \in C$.
if $(\texttt{measures}, \texttt{IPC}) \in C$ then $(\texttt{raw data}, \#\texttt{Exec. Instructions}) \in C$ $\wedge(\texttt{raw data}, \texttt{Execution Time}) \in C$.
if $(\texttt{measures}, \sum \texttt{Allocated Cores}) \in C$ then $(\texttt{raw data}, \#\texttt{Allocated Cores/App}) \in C$.
if $(\texttt{measures}, \sum \texttt{Off} - \texttt{Chip Memory Bandwidths}) \in C$ then $(\texttt{raw data}, \texttt{Off} - \texttt{Chip Memory Bandwidth/App}) \in C$.
if $(\texttt{measures}, \sum \texttt{Shared Cache Space}) \in C$ then $(\texttt{raw data}, \texttt{Shared Cache Space/App}) \in C$.
if $(\texttt{methods}, \texttt{SIMO Ctrl.\&ARMA model}) \in C$ then $(\texttt{control actions}, \#\texttt{Allocated Cores})$ $\vee(\texttt{control actions}, \texttt{Off-Chip Memory Bandwidth})$ $\vee(\texttt{control actions}, \texttt{Shared Cache Space}) \in C$.
if $(\texttt{goals}, \texttt{Resource Exploitation}) \in C$ then $(\texttt{methods}, \texttt{SIMO Ctrl.\&ARMA model}) \in C$.
if $(\texttt{requirements}, \texttt{Performance}) \in C$ then $(\texttt{methods}, \texttt{SIMO Ctrl.\&ARMA model}) \in C$.

Table 3.1: Model constraints for the METE self-adaptive system.

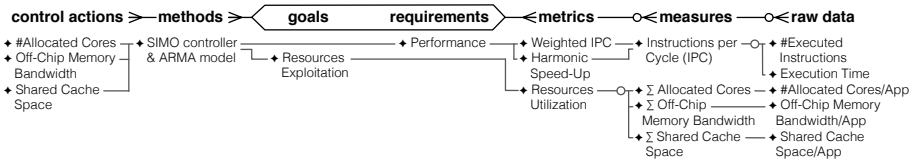


Figure 3.4: Context dimensions and domains for METE research project.

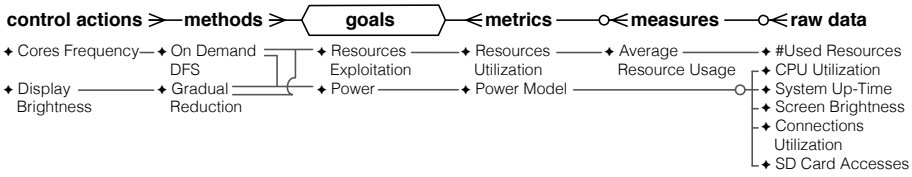


Figure 3.5: Context dimensions and domains for Into The Wild project.

Power Optimization for Mobile Architectures

The Into the Wild [85] project aims at optimizing power consumption in mobile architectures, by acting on the architecture parameters. A regression-based power estimation model and a metrics built directly on raw data are used to guide towards power optimization. A preliminary analysis shows how the screen and the CPU are the two largest power consuming components: the brightness of the former and the frequency of the latter are the knobs on which the self-adaptive computing system can act.

The graphical representation that summarizes the context model dimensions, domains and relations/constraints is shown in Figure 3.5.

Performance Management via Self-Adaptivity

The Metronome [86] project is another adaptive system having a performance requirement and aiming at optimizing the resource usage. In particular, the system keeps track of the running applications' progress by monitoring the heartbeat signals they emit and by aggregating them in a heartrate measure. For each application, a performance goal is specified in terms of a maximum and a minimum heartrate, i.e., a range within which this measure must stay, to maintain a uniform behavior. Moreover, to achieve this goal the system acts with a scheduling policy that adjusts the amount of processor time assigned to a given task (called virtual runtime) to fulfill the performance requirement and, at the same time, to preserve the scheduler fairness, also avoiding useless computation.

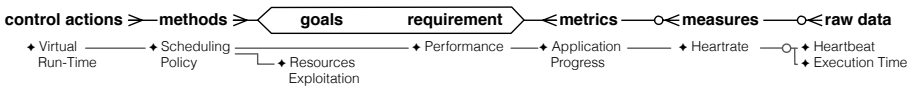


Figure 3.6: Context dimensions and domains for Metronome project.

The graphical representation of the system, according to the proposed model, is shown in Figure 3.6, reporting dimension values and relations among them.

3.3 Final Remarks

The context model defined in this chapter supports the definition, design, and implementation of a self-adaptive computing system. At present, the model supports the designer in specifying what are the characterizing elements with respect to which the system will exhibit self-adaptiveness properties, also relating the various elements among themselves. As such the context model can be used for organizing the specific self-adaptive aspects in a structured and systematic way, and for providing a shared dictionary for their documentation, as well as for preliminary validation of the architectural/implementation choices to make sure all necessary objects (sensors and actuators) are available.

Among all the dimensions considered in the model, reliability is the one on which the following chapters will be focused. The design of a self-adaptive system for dealing with transient faults is described in the next chapter. Such a system implements the proposed framework in the design phase and exploits different redundancy scheduling techniques to guarantee a level of system's reliability which changes at run-time according to the current needs.

4 Runtime Transient Fault Management

This chapter presents the novel approach we propose to support runtime adaptive management of transient faults that may occur. We first introduce an overview of solutions available in literature, highlighting the limitations our solution aims at tackling. Then, the approach is presented together with the experimental campains to validate our proposal.

4.1 Background

The approach defines a layer at the operating system level that achieves fault detection/tolerance/diagnosis properties by means of thread replication and re-execution mechanisms. The layer applies the most convenient hardening mechanism to achieve the desired trade-off between reliability and performance by adapting at run-time to the changes of the working scenario. Figure 4.1 shows the context dimensions and domains of the proposed self-adaptive system according to the model introduced in the previous chapter. The proposed strategy has been applied to a set of experimental sessions considering part of the case studies already introduced, to evaluate its benefits on the final system with respect to various strategies selected at design time. A comprehensive description of different aspects of the self-adaptive system here proposed can be found in [18, 19, 16].

The adoption of fault management strategies at such abstraction level has been preliminary investigated in [25]. In the scenario of a multicore architecture, the authors proposed a reliable dynamic scheduler,

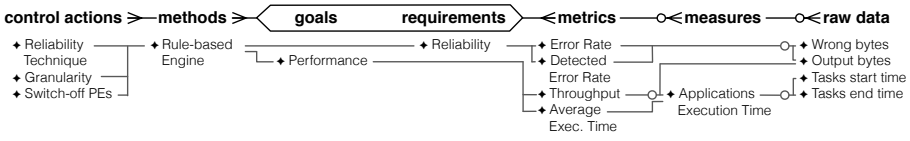


Figure 4.1: Context dimensions and domains for the fault tolerance management self-adaptive system.

describing a unique hardening technique, but focusing their attention on a broader spectrum of issues, including diagnosis, components aging and more. We propose a fault management layer at the operating system level, implementing a strategy for dynamically adapting the reliability level. The layer lies on top a revised version of a previously presented scheduler [25], and introduces new elements to define the overall *adaptable* reliable system. The performance/reliability trade-off is evaluated at runtime and the most convenient fault management technique, according to globally defined metrics, is chosen, while not considering, at the moment, other issues.

The methodology we envision is suited for application scenarios where the dependability requirements need to be enforced only in specific working conditions (e.g., when an emergency situation arises) and/or for limited time-windows. In these cases, the system can adapt its behavior and expose robust properties with possibly limited performance, or the other way around. This is mainly due to the fact that performance and reliability are usually conflicting goals. To balance this trade-off, the methodology can act both on the hardware-level (e.g., FPGA-based systems, on which dynamic partial reconfiguration can be exploited to vary the required reliability of a hardware component) and on the software-level (as the multicore scenario here proposed), based on i) how hardening techniques can be applied and ii) how the tuning of the hardening is managed.

4.2 State of the Art

The proposed self-adaptive system is based on the autonomic control loop paradigm. This paradigm has been implemented in many research fields with promising results; in particular, various approaches, as Angstrom Project [98], Barrelfish [7], and K42 by IBM [4], focus their attention on enhancing operating systems with self-adaptive capabilities through the introduction of an autonomic control loop. Their goal is to improve the system performance in terms of throughput or execution time and, sometimes, balance performance with power consumption, by applying self-adaptation to different aspects of the system (e.g., process scheduling or memory access management).

When considering reliability issues, several approaches for multicore systems have been proposed in literature. Some of them consider permanent faults only, but they worth being mentioned for some of the features they propose. In particular, the approach proposed in [49] features a thread remapping mechanism for recovering from permanent faults;

however the recovery schemes are defined at design time, thus offering a limited number of recoveries. An on-line approach for task remapping has been proposed in [73], to deal with the occurrence of permanent processor failures. The authors propose a runtime manager in charge of making decisions concerning the adaptation of the system to changing resource availability. In particular, it identifies the most convenient resource on which to remap the task, by means of online task remapping heuristics. Although the proposal exposes dynamic adaptability characteristics, they are exploited for the decision of the best remapping candidate, with respect to a static precomputed solution, whereas there is no dynamic tuning of the achieved reliability.

Approaches that deal with transient faults as well are closer to the topics covered in this chapter; a comprehensive survey discussing, among the others, solutions based on redundant execution to achieve fault detection/tolerance, is presented in [41]. Most of the works surveyed (e.g. [75, 65]) are based on application-level replication, re-execution or checkpointing. However, they adopt a single hardening technique, thus representing a rigid solution which is not able to adapt to the possible changes in the environment or in the system. Most of them limit their considerations to single threaded application, an assumption that does not fit today data-intensive applications. Some others (e.g. [2]) require a custom architectural support to manage the replicated threads and the checking/voting activity. Few approaches present a limited adaptability to the evolving scenario. For instance, in [65, 2], two different strategies for dynamically coupling processors are presented, aimed at re-configuring the system architecture after the identification of a permanently faulty core.

The approach presented in [99] offers the possibility to switch on and off the application duplication strategy depending on the specified requirement, that may be performance or reliability. An interesting adaptive approach is proposed in [30]: it is based on a set of simple rules for changing the checkpoint schema at run-time according to the architecture's health and current configuration aiming at optimizing the performance. Finally, in [25], an adaptive approach for multi/manycore systems executing parallel applications is proposed. The approach applies a single fault tolerance strategy and tries to perform a load balancing of the threads on the available processing units to optimize the performance while fulfilling reliability requirements. Moreover, the approach is able to adapt in case a unit is switched off after the detection of a permanent failure.

In most of the existing runtime adaptable approaches for achieving a dependable and high-performance system, the overall goal is to achieve

dependability while minimizing performance degradation, by adapting the execution of the application or the hardening to a changing platforms, affected by faults. Therefore, most efforts are devoted to determine, at runtime, which new *system configuration* can provide the best performance, with a modified (partially faulty) architecture. No solution has yet tried to address the issue of dynamically adapting and leveraging the achieved level of dependability, according to the evolving modified architecture, as faults occur, in order to still balance the dependability/performance trade-off.

4.3 Fault Tolerance through Self-Adaptiveness

The adaptive mechanisms that we envision are implemented by a fault management layer introduced on top of the architecture and of the operating system, as shown in Figure 4.2. The main goal of this layer is to harden the execution of the issued applications and, at the same time, to balance reliability and performance according to a set of conditions that may evolve in time. This new layer borrows some concepts from the one presented in [25], as far as hardening strategies are concerned, although we here generalize the approach in this respect, while focusing the attention of the new self-adaptive features.

The architecture of the fault management layer is now introduced, while the design of the orchestrator will be discussed later in this section. Do note that the work here presented mainly focuses on the proposal of the adaptive fault tolerant strategy from a functional point of view. Therefore, we present some details on the internals of the fault tolerant layer, but we will not delve into its implementation-specific aspect, since these aspects are strictly related to the specific target architecture and operating system.

4.3.1 Fault Management Mechanisms

The fault management layer introduces reliability properties (fault detection or fault tolerance) in the application execution, by means of mechanisms based on replications and re-executions of the overall applications or some of its threads. These mechanisms are also used for performing diagnosis activities in order to identify suspect cases of permanently damaged processing cores. In particular, the following techniques are supported at present.

- **Duplication with Comparison (DWC).** As shown in Figure 4.3, the technique guarantees the fault detection property by creating

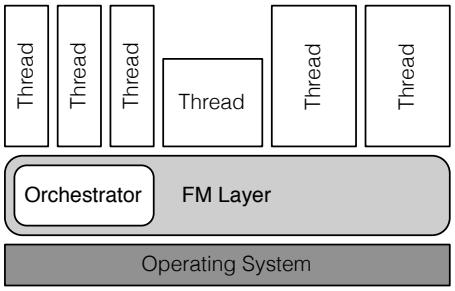


Figure 4.2: System structure overview: the FM layer acts between the applications and the OS.

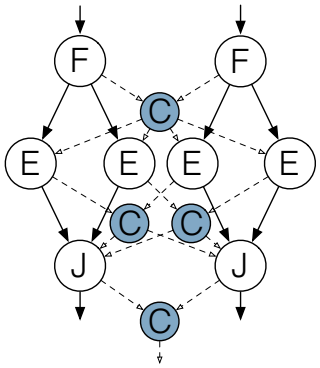


Figure 4.3: Duplication with Comparison (DWC) technique applied to all the tasks of the sample application of Figure 2.2.

a replica of the application and by comparing the outputs. In particular, a checker task is issued at the end of each node of the application's task-graph to identify discrepancies in the (intermediate) results.

- **Triplication (TMR).** This technique creates two replicas of the original application, so to have three results to be voted by specific 2-of-3 majority voter tasks that mitigate the possible occurred faults (Figure 4.4). Besides the fault tolerance property, the technique is also able to achieve fault diagnosis features, by identifying the core producing the erroneous mismatching value.

- **Duplication with Comparison and Re-execution (DWCR).** Similarly to the DWC technique, the original application is dupli-

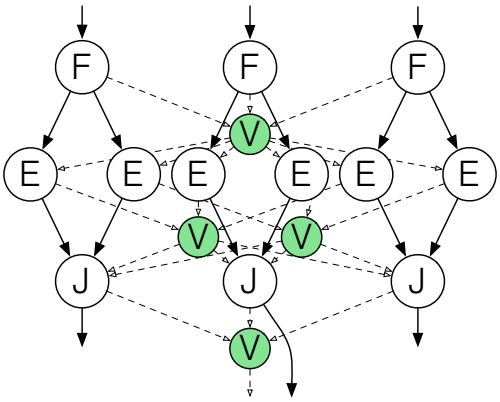


Figure 4.4: Triplication (TMR) technique applied to all the tasks of the sample application of Figure 2.2.

cated, to have the possibility to detect possible errors by comparing two results by means of a checker task. If an error is detected, a third replica of the task is created and executed: in this way a voter task is able to identify the correct result. Figure 4.5 shows an example. This technique provides the fault tolerance property, and may provide fault diagnosis features as well. It is characterized by a limited overhead for achieving the fault tolerance property, because the third replica is used and scheduled only after a problem has been detected. As a result the technique exhibits a time-redundancy style of mitigation causing a delay in computing the correct results, when compared against the previous techniques. In fact, the technique aims at limiting the overheads in the most common case of fault-free execution, while incurring in an additional timing overhead when a fault occurs.

Since we need to guarantee the correctness of the voters and checkers results, these tasks must be executed on the controller core, which is the only unit hardened by-design. Moreover, their duration is variable and, in particular, it depends on the amount of data they are testing. Finally, when a join node has only a synchronization functionality and does not generate any output data, the voter will check only the correct termination of its replicas.

Do note that the architecture uses a communication model based on a shared memory. Therefore, there is no actual transmission to the control core of the data to be compared/voted, that may cause a communication congestion. At the end of a task, the PE will signal the termination and,

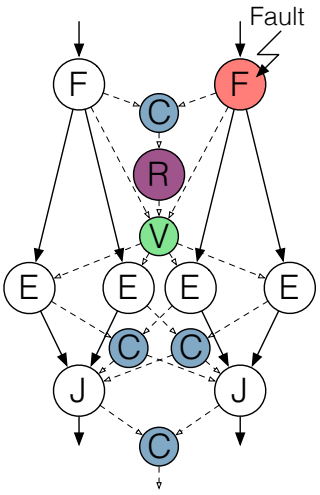


Figure 4.5: Duplication with Comparison and Re-execution (DWCR) technique applied to all the tasks of the sample application of Figure 2.2.

during the execution of the voter/checker task, the controller core will access the data in the memory region where the task wrote the results.

4.3.2 Layer Internals

In order to perform the additional hardening activities for enabling self-adaptive reliability, similarly to [62], the layer wraps a set of system-calls to the underlying operating system which are: `thread_create`, `thread_join` and `thread_ exit`, devoted to thread start, termination and synchronization, respectively, and `_start` and `_exit`, for the start and termination of the main application thread.

Let us explain the basic mechanism through a simple example. Consider a single-threaded application requiring fault tolerance properties, on which TMR is applied; when it is started, the layer intercepts the `_start` call and creates two additional thread replicas. In turn, the tile scheduler will execute the replicas, and, after their termination identified by trapping the `_exit` call, the layer creates and executes a task that votes the obtained results. To implement the exemplified behavior, the layer collects the information of the threads; in particular, it keeps track of the corresponding replicas of the same thread, to issue voting/checking tasks upon their termination. The FIFO scheduler uses a number of queues equal to the number of generated replicas and adopts

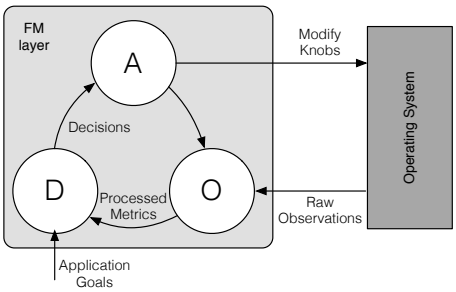


Figure 4.6: The Observe–Decide–Act control loop.

a round-robin policy for selecting the current queue from which pops the task to be executed; in this way it guarantees a balancing in the execution of the various replicas. Moreover, to avoid that the use of permanently failed PE for executing more than a replica could mine the effectiveness of the hardening techniques, the scheduler maps corresponding replicas on different cores.

The selection of the fault management mechanism to be applied is the most critical activity, because each strategy provides specific reliability properties and presents a different overhead on the performance due to the number of generated replicas. Thus, an ad-hoc version of the orchestrator has to be designed for performing this decision, and is presented in the following section.

4.3.3 Orchestrator Design

The innovative aspect of the proposed approach is the inclusion of the orchestrator in the fault management layer. It can act on the fault management mechanisms and the scheduler behavior. The dynamism of the environment consists in changes in the kind of issued applications, the application requirements (performance vs. reliability), health status of the processing core within the architecture, and the system workload.

The self-adaptive behavior has been implemented in the system by means of the, already introduced, Observe–Decide–Act (ODA) control loop, shown in Figure 4.6. The implementation of the three phases is discussed in details in the following subsections.

Observe

A crucial step in designing the observe phase is the definition of the quantities to be sensed and in the way they are aggregated into met-

rics. They provide a description of the system's current status as much complete and concise as possible to be useful to make decisions in the subsequent phase. The following metrics have been identified in order to observe the two main aspects of interest in the considered multicore scenario: the performance, since such architectures are used as intensive data processing applications, and the reliability, due to the increasing failure trend.

For estimating the performance, the classical metrics measuring the execution time are adopted: in particular, the average execution time, and the throughput. In an application modeled through a task-graph (as the example in Figure 2.2) the execution time of a single application cycle is measured by evaluating the elapsed time between the beginning of the execution of the first fork node and the end of the execution of the last join node:

$$exec\_time_{overall} = t_{end} - t_{start}. \tag{4.1}$$

Then, the average execution time is computed on a window of the last n executed cycles.

An alternative metric is the throughput. To evaluate this metric, the orchestrator considers a window of the last n executed cycles of the application and computes the ratio between the amount of produced data and the overall execution time for that window:

$$throughput = \frac{n \cdot d_{out}}{t_{end_n} - t_{start_1}} \tag{4.2}$$

where d_{out} is the amount of data produced during a single cycle and the cycles are numbered from 1 to n.

When referring to reliability, the most relevant issue of the considered data-intensive computing applications is related to the amount of errors they may experience. Thus, the defined reliability-oriented metric measured by the adaptive environment is the detected error rate. Moreover, it is computed on a window of the last n cycles of the application execution. When considering the application model, we can estimate the error rate by quantifying the portions of the results produced by faulty tasks. Do note that in quantifying the error rate, fault propagation, task crashes, and the detection method in use have to be taken into account too.

Not all the hardening strategies are able to mitigate the detected errors: in fact, DWC is not able to mitigate errors, and in some scenarios other techniques may potentially fail, as discussed in the following. When an error is not mitigated, it will be observed on the final output of

Detected Error Ratio	**#Resources** *low*	*high*
low	TMR < DWCR	TMR ∼ DWCR
high	TMR ∼ DWCR	TMR > DWCR

Table 4.1: Qualitative performance comparison between TMR and DCR techniques. The comparison refers to the same level of granularity.

the application. Therefore, we also defined the not-mitigated error rate which computes this percentage of errors.

Act

During the act phase, the system directly acts on the system knobs, according to the choices taken in the decision phase. In our scenario we can identify several knobs: the selection of the fault management mechanism to be applied, the selective application of the mechanism, the granularity at which it is applied, the possibility to activate or deactivate a processing unit or changing its frequency, and so on. We here focus on a subset of them presented in the following; all the other knobs are left for a future investigation.

The first considered knob is *mechanism selection*, that is the choice of the fault management mechanism to be applied on the running application. As discussed, each technique has specific properties (fault detection, fault tolerance and fault diagnosis) and performance overhead. In particular, the performance for the two mechanisms offering fault tolerance is listed in Table 4.1. TMR can achieve a greater throughput when there is a high number of available processing cores and the system is experiencing a high error rate: this is due to the fact that resources are better exploited when running the three replicas in parallel; DWCR would not be able to exploit these resources since the third replica would be executed strictly after the first two and a voter will be also executed in addition to the checker. On the other hand, DWCR proved to achieve a higher throughput in a scenario with a low error rate and a reduced number of resources since the third replica is optionally executed. In the other two cases the performance of the two techniques depends on the specific scenario and cannot be classified a-priori. Experimental evidences of these considerations will be shown later on.

The second knob is the *granularity* the mechanism is applied at. The

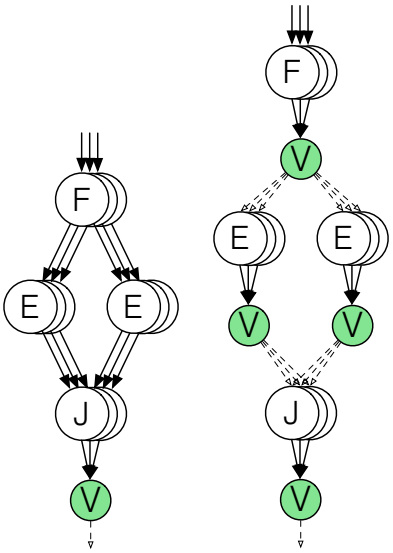
granularity represents the possibility to perform a varying placement strategy of the voter/checker tasks introduced by each fault management mechanism, thus offering another way to tune the performance/reliability trade-off. We consider only two different granularity levels, even if other intermediate ones could be added:

- *Coarser*: checker/voter tasks (according to the considered mechanism) are added only after the execution of an independent part of the program: after join and fork_join tasks.
- *Finer*: checker/voter tasks are placed after each task in the task-graph.

The concept of granularity can be, in theory, applied to all the hardening techniques. Figure 4.7 shows an example applying the TMR at the two different levels of granularity on the sample application of Figure 2.2. It is worth noting that in the coarser level only the final results are checked/voted, while the possible intermediate results are discarded. Moreover, when the join task has only a synchronization role and has no data as final output, the voter will test the output data of the child elaboration threads.

From a reliability point of view, some combinations of granularity level and fault management mechanism do not offer any advantage. This is the case of the DWC that offers the same features for both configurations; however, as discussed later, the performance of the two configurations depends on the specific working scenario and therefore it is not possible to choose the best configuration a-priori. At the opposite, DWCR and TMR offer different reliability properties according to the granularity.

When the coarser granularity-level is selected, it is not possible to guarantee diagnosis features since error propagation effects would be introduced and the source of the error would not be identifiable. Moreover, in case of permanent faults, mechanisms offering fault tolerant properties cannot guarantee a 100% coverage, i.e. not all the faults occurring in the system can be actually mitigated by the technique. In fact, the mapping constraint of the sibling replicas discussed in Section 4.3.2 allows a single permanent processor failure to invalidate tasks belonging to different group of replicas before they are voted, as shown in Figure 4.8. In order to guarantee a 100% coverage, the PEs should be partitioned in a number of groups equal to the number of generated replicas, i.e. three. However, since in general the number of processing unit is not multiple of three, this partitioning would imply lower performance. Nevertheless, even if not providing the full error coverage, these mechanisms still provide 100% fault detection coverage.



(a) Coarser Level (b) Finer Level

Figure 4.7: Application's task–graph hardened at the two different levels of granularity: the colored dashed tasks represent the voter nodes added to make fault mitigation possible.

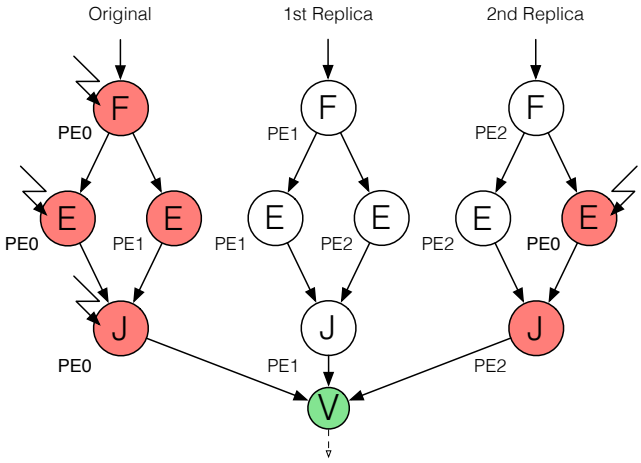


Figure 4.8: If PE0 is permanently faulty, the presented mapping of the tasks on three processing cores, PE0, PE1 and PE2, causes TMR to fail.

Granularity	*#Replicas*	*100% FD*	*100% FT*	*Diagnosis*
Duplication with Comparison (DWC)				
Coarser	2	Yes	No	No
Finer	2	Yes	No	No
Duplication with Comparison and re-Execution (DWCR)				
Coarser	2/3	Yes	No	No
Finer	2/3	Yes	Yes	Yes
Triplication (TMR)				
Coarser	3	Yes	No	No
Finer	3	Yes	Yes	Yes

Table 4.2: Qualitative evaluation of the described techniques with reference to reliability aspects (FD: Fault Detection – FT: Fault Tolerance).

The placement of voter and checker tasks after each node allows DWCR and TMR to achieve a 100% fault tolerance coverage; moreover, this placement offers also diagnosis capabilities. Table 4.2 summarizes the reliability aspects presented above.

The granularity level has also an impact on performance. Usually, the selection of the finer granularity level incurs in a higher overhead on the execution latency with respect to the coarser level, because a larger number of checker/voter tasks is introduced. However, in some specific situations the opposite behavior may occur, as we will show in the experimental results. More precisely, when the architecture is composed by a small number of processing units and the application generates a large number of parallel threads, the finer granularity level offers the possibility to better schedule on the controller core a larger number of checker/voter tasks comparing the results of the elaboration threads, since the other processing units are overloaded with the execution of the tasks' replicas. Due to this reason, DWC applied at the finer granularity level has not been discarded even if it does not offer any advantage from the reliability point of view. Finally, do note that the considerations drawn in Table 4.1 hold at each level of granularity and, more precisely, these relations are exacerbated as the level of granularity becomes coarser. In particular, if the system experiences a high detected error rate, DWCR is highly disadvantageous with respect to TMR since it would require a considerable number of tasks to be re-executed.

The last knob is the *resource activation/deactivation*. As discussed in the description of the fault management techniques, during the execution, the orchestrator may diagnose a suspected damaged PE. In this case, it can deactivate the PE in order to further analyze it by means of specific diagnosis tasks. A PE can be deactivated by simply removing it from the list of the available resources used by the scheduler. Later, if the result of the accurate analysis is negative, the PE can be reactivated and used again for executing the application.

As a final remark, do note that knobs can be adjusted only between two different iterations of the application execution for allowing the context switch. This is particularly true for the fault management mechanisms since it is quite complex, even not unfeasible, to adapt internal data and synchronizations related to replicas and checker/voter tasks from a mechanism to another one.

Decide

The decision phase is implemented by the orchestrator through a rule-based decision policy. From a high-level point of view, it evaluates the system's current status on the basis of the sensed metrics acquired in the observe step, and makes a decision on the actions to be taken through the knobs to reach the desired goal, specified as input when the application has been issued.

As discussed in the previous sections, the two conflicting aspects on which the proposed system focuses are performance and reliability; moreover, each one of the considered hardening mechanisms together with the granularity level of its application imply a specific impact on each one of the two aspects: some mechanisms obtain a reduced performance overhead but limited reliability properties, while other ones achieve high reliability but incurring in a large performance overhead. Thus, the aim of the system is to decide the most suitable mechanism to be adopted in each situation to achieve a good trade-off between the two aspects. In particular, the designed orchestrator allows one to specify the goal for each issued application, by specifying which of the two aspects has to be considered as constraints; consequently, the system will try to adapt its behavior to optimize the remaining dimension. The adaptation is even more necessary if we consider that the system conditions can evolve: in particular, different applications with different goals may be issued, and the architecture may change due to the deactivation of some faulty units.

The goal is specified as a threshold on the value of the metrics representing the aspect to be controlled (not-mitigated/detected error rate for the reliability, or average execution time/throughput for the perfor-

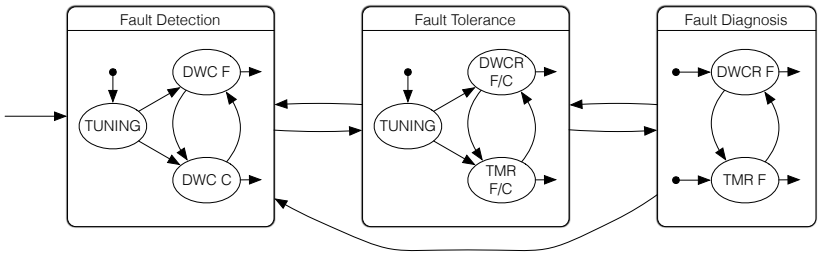


Figure 4.9: A FSM representation of the decision process in the case reliability is the constrained dimension.

mance). Do note that the orchestrator will try to fulfill the specified constraint, however, there is no guarantee that it will be satisfied *completely*; in fact, the various decisions are made at run-time, and therefore only at that time it will be possible to know whether the goal can be reached.

The implemented decision policy is a simple rule–based system. The specified goal is used by the system, first to identify the constrained variable and the free one, and then to set a threshold used to take a decision. In particular, the idea is to start the execution by choosing, among the hardening mechanisms, the one having, in theory, the best expected result for the unconstrained dimension and to analyze the effects on the constraint variable. The value specified as a constraint on the application is compared with the current value of the metrics: if the constraint is met, the mechanism currently in use does not need to be changed; otherwise, a more appropriate technique is selected. This first threshold is strict and a mechanism change occurs every time the threshold is reached. Other thresholds can be determined at run-time by estimating the effects of the available mechanisms on the current status of the system.

We discuss here the decision policy design when setting the reliability as the constrained dimension. The goal is specified together with the input task-graph as a not-mitigated error rate R_1 not to be reached. The decision policy can be modeled with a hierarchical finite state machine (FSM) is shown in Figure 4.9. The initial macro-state, labeled *fault detection*, forces the system to apply the DWC mechanism, that is the one that has the smallest impact on performance and allows to perfectly keep track of the value of the current detected error rate. As discussed DWC will only detect the errors without mitigating them. In particular,

the first step is a tuning phase that aims at evaluating the performance achieved by DWC when applied at finer and coarser granularity levels: each one of the two levels is tested for one cycle. Then, the system evolves in a second state where DWC is applied at the granularity level that obtained better performance.

If the not-mitigated error rate threshold R_1 is reached, then a mechanism enforcing the required fault coverage needs to be applied. This will have the effect of reducing the error rate, while penalizing performance. Therefore, the system evolves in the *fault tolerance* macro-state. In this macro-state, the decision policy first performs a tuning phase for profiling the two hardening mechanisms. In particular, for each mechanism both the granularity levels are evaluated, and the less convenient one is discarded; then, the decision policy selects the mechanism offering higher performance and consequently evolves in the related sub-state. Do note that the performance of DWCR depends on the number of required re-executions while the TMR one is almost constant. For this reason, the adaptation stage will switch among the two mechanisms (at the granularity level identified during the tuning phase) according to the performance monitored during the execution.

If the detected error rate does not decrease in a given executions' window, it means that some permanent failures may affect the architecture. Therefore, to perform fault diagnosis, the system evolves to the *fault diagnosis* macro-stage, which behaves similarly to the previous one but exploiting only the finer granularity level. Note that the tuning step is not necessary since the techniques have been already tested in the previous macro-state. During the diagnosis activity, based on the analysis of the fault detection on each PE, if a unit is determined to be faulty, it is switched off, and then the orchestrator evolves to the DWC state. Finally, if the detected error rate decreases, the system evolves from a macro-state to the previous one.

The FSM derived for the case of setting reliability as the constrained dimension can be straightforwardly adapted to the case where performance is the constrained dimension, or the case where a set of batch applications is issued. In particular, the same thresholding mechanism will be used, while the specific actions/techniques applied in each state will be different. Moreover, the described decision phase is clearly extensible with new mechanisms, automatically taken into account during the tuning state.

4.4 Experimental Results

In this section we present the experimental sessions carried out to demonstrate the effectiveness of the orchestrator. First, the experimental set-up is illustrated and, later, the results of the experiments are discussed in two different paragraphs.

4.4.1 Experimental Set-up

We adopted and enhanced the simulator previously defined for the experimental sessions presented in [25]. The original tool consists of a SystemC transaction level model [1] of the discussed architecture able to simulate the execution of applications modeled in terms of fork-join graphs. Thus, we implemented in the SystemC model all the considered fault management mechanisms, the support for the granularity levels and the discussed orchestrator.

In the performed experimental sessions, we considered a set of commonly used parallel applications taken from the case studies. In particular, we present here the results obtained on a specific application aligned with the application scenario we refer to, that it the edge detector. We characterized it with a static analysis on the source code for building the task-graphs annotated with the amount of data processed by each task. Moreover, we used an instruction set simulator running a sequential version of the application for estimating the execution times of the various tasks, as in [67]; such execution times contains also bus and memory access latencies. Indeed, as noted in [67], when working at this level of abstraction, interactions among tasks (e.g., resources conflicts) are not of interest since they are later considered in more accurate lower-level simulations. In particular we used the ReSP simulation environment [80] for simulating an ARM7 unit working at $333MHz$ connected to a memory through a bus, each one with a transmission latency equal to $10ns/word$. Finally, we also estimated the execution times of the voters and checkers for the different amount of data to be compared. The resulting task-graph is shown in Figure 4.10, while a subset of the execution times for the voters and checkers are presented in Table 4.3.

We generated fault lists according to an exponential probability distribution over the experiment duration, since it is well suited for describing transient faults. They were created by varying the failure rate of the distribution in order to stimulate the system in different ways, thus causing mutating environmental conditions. The generated lists obey the single failure model assumption. Note that, the considered fault probabilities have been created with a high frequency with respect to fault occurrence

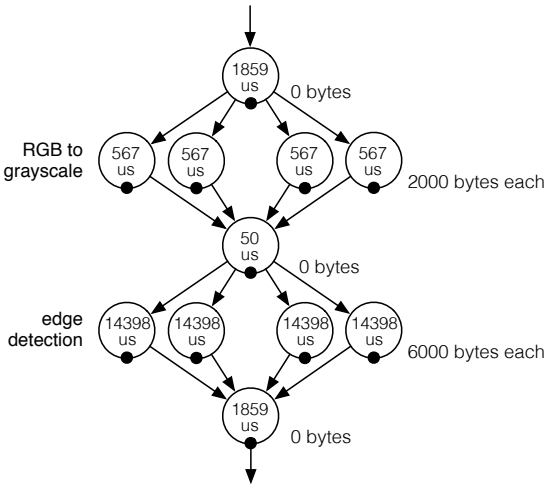


Figure 4.10: Task-graph for the edge detector application.

# **Data (byte)**	**Checker**	**Voter**
2000 byte	$423\mu s$	$920\mu s$
6000 byte	$1237\mu s$	$2550\mu s$

Table 4.3: Execution times for voting and checking various amount of data.

in the real world; the aim is to perform a sort of *accelerated* experiment to better highlight the capability of the orchestrator to adapt to the environment stimuli. Therefore, we expect the system to behave in the same way in the real world scenario, although on a longer time window. It is worth noting that the accelerated experiment requires the size of the window for the metric computation to be shrunk to make the orchestrator more reactive to the higher external stimuli.

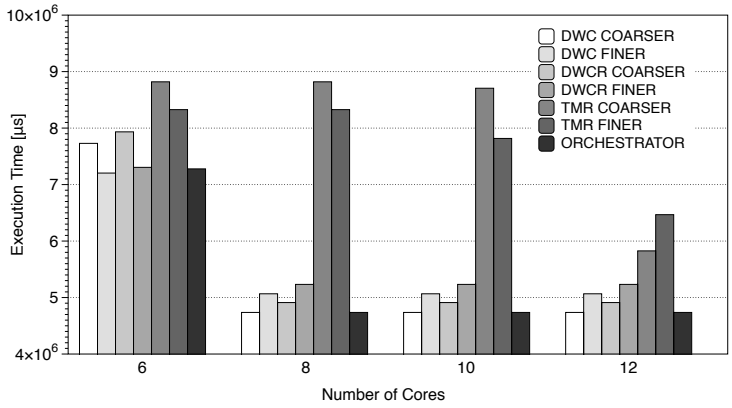
4.4.2 First Experimental Session

The goal of the first experimental session is to show how each single fault management mechanism behaves differently according to the specific scenario, offering a different performance/reliability trade-off. Moreover, the orchestrator is able to select, during the execution, the most suitable fault management mechanism at each time, outperforming the single statically-selected strategies. We considered four different architectures composed of 6, 8, 10 and 12 processing cores, respectively, and we generated three fault lists $F1$, $F2$ and $F3$ with a failure rate (i.e., probability of a single processor failure) equal to 0.0005, 0.001 and 0.003 every $200\mu s$.

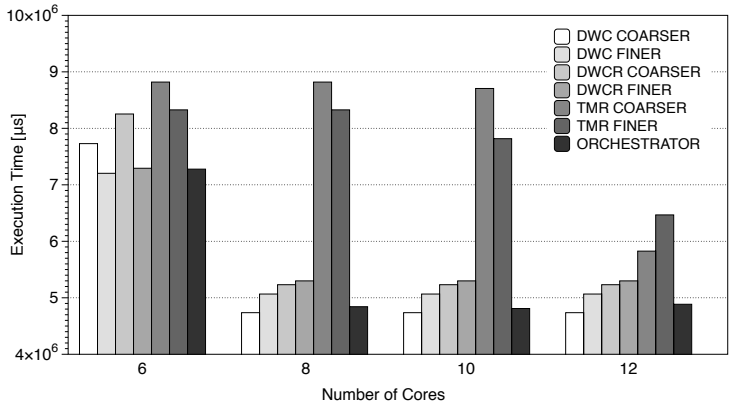
In this experimental session we ran the application hardened with all the available mechanisms, selected at design-time, and with the proposed decision policy. Moreover, we tested all the strategies on each pair <architecture, fault list>. The comparison of the various approaches was performed by executing the application for a specific number of cycles (equal to 200), thus mimicking the elaboration of a fixed amount of data. We considered a threshold for the not-mitigated error of the 5% and a window for the metrics computations of 10 cycles.

The results of the experimental session are presented in Figure 4.11, where the overall execution times are reported. Each chart presents the results obtained for each fault distribution when adopting various mechanisms on the considered architecture. The average of not-mitigated error rates confirmed the expected values: 0% for all fault tolerance strategies (since no permanent fault was injected), less than 5% for the proposed adaptation policy, while DWC is not able to mitigate any error. We did not report them in the figure.

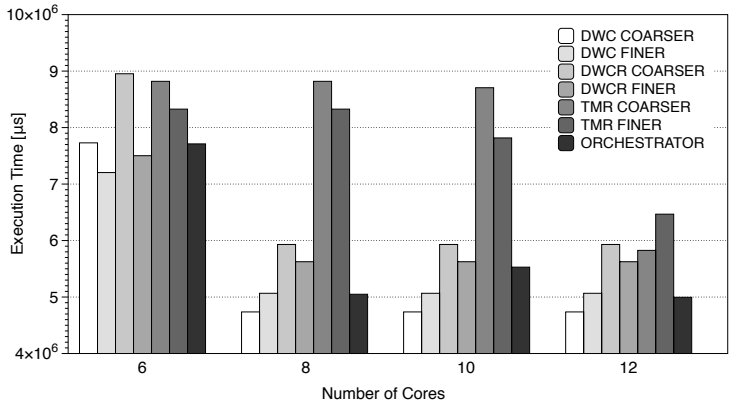
From an accurate analysis of the bar charts we can notice that, as expected, there is not a mechanism that is always prevailing on the other ones, and each of them achieves better performance in a specific scenario. Moreover, the following empirical rules can be inferred. In general, even if coarser granularity level introduces a smaller number of voter/checker tasks, the finer one is preferable when the number of available processing



(a) Fault list $F1$.



(b) Fault list $F2$.



(c) Fault list $F3$.

Figure 4.11: Overall execution times for the edge detector on the various architectures stimulated by each fault list.

cores is small, as already motivated in Section 4.3.3. In particular, this can be observed from the results reported in Figure 4.11 for the architectures with six cores. A detailed investigation of the scheduling Gantt charts allowed us to highlight the following situation: when the number of PE is limited, voter tasks analyzing intermediate results used in DWC/F do not overload the control core thus achieving an overall better performance than DWC/C, that postpones all results' checking at the end of the application's replicas execution. This phenomenon is exploited by the proposed technique that thus achieves better performance.

Moreover for DWC and TMR, the break-even point is represented by the value obtained by multiplying the maximum number of parallel threads generated by the application by the number of replicas required by the mechanism (for instance, for the edge detector, 4×3 for TMR and 4×2 for DWC); on the other hand, it is not possible to identify an exact break-even point for DWCR even if the trend is similar. The second consideration is that the performance of the two granularity levels for DWCR depends also on the fault frequency: in fact, when the frequency is high the finer level is more convenient since it allows to re-execute only a reduced portion of the application. A final consideration is that TMR seems not to be convenient in any scenario; however, as shown in the following experiment, it will outperform the other mechanisms when the error frequency is even higher, i.e., when a permanent fault affects the architecture.

When considering the proposed adaptation policy (last bar in each chart), we can notice that it outperforms all the statically selected approaches (DWC should not be considered since it is not able to mitigate the errors thus producing corrupted results). As expected, the orchestrator is able to select the best mechanism in each instant of time, thus maximizing the performance while meeting the specified threshold on the not-mitigated error ratio. The performance improvement on the best of static approaches in each scenario spans from 1% to 13%. In one scenario only the orchestrator exhibits worse performance than the DWCR applied at finer granularity (i.e. on the architecture with 6 cores stimulated by the fault list $F3$); indeed, the limited number of processing units in the considered architecture causes a high latency to execute a single cycle, and, consequently, the orchestrator evolves too slowly to select the appropriate mechanism to mitigate the high frequency of faults.

Even if the scenarios presented so far showed the necessity for a dynamic mechanism selection, they are not the situations in which the proposed orchestrator shows its effectiveness. In fact, the considered fault lists have been generated by means of a constant failure rate and no permanent fault has been injected. Therefore, in a second experi-

mental session, we aimed at showing the adaptation capability of the orchestrator by considering a fault list generated with a variable failure rate and a permanent fault corrupting a single processing core.

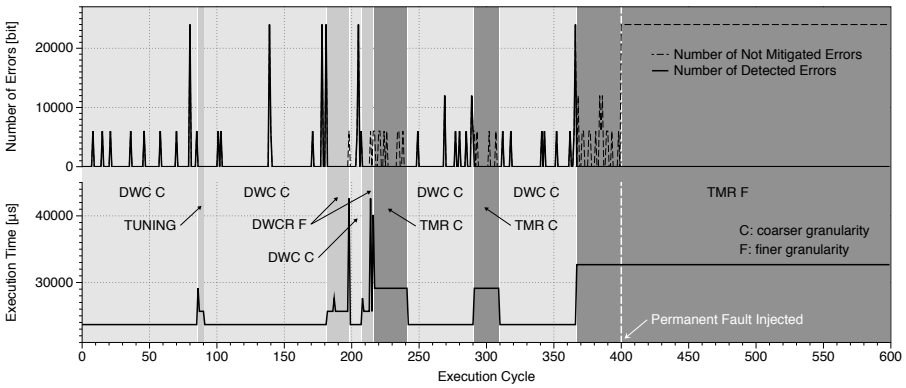
4.4.3 Second Experimental Session

In the second experimental session, we executed the edge detector in a scenario with evolving environmental conditions in terms of fault distribution. For about the first $4,000s$ the system experiences a failure rate equal to 0.001 every $200\mu s$; then, from time $4000s$ to $10,000s$ this probability is increased to 0.003 every $200\mu s$: in this way the behavior of the system when a varying failure rate occurs is tested. To make the scenario more complete and complex, at time $10s$ a permanent failure is injected in one of the architecture cores. The edge detector is run for 600 cycles on an architecture made up of 12 cores.

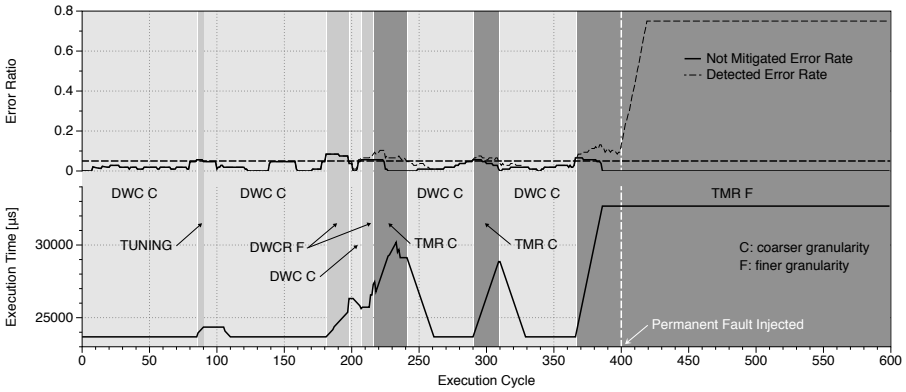
The first plot (Figure 4.12(a)), aims at showing which is the system's behavior perceived from the outside. The top part of the graph shows the number of detected and not-mitigated errors experienced at each cycle by the system. This allows to figure out the role played by the fault mitigation techniques in reducing the number of visible faults. Finally, the gray vertical areas describing the used hardening approach in each time interval.

The second plot (Figure 4.12(b)) shows the internal status of the orchestrator, and in particular presents the values of the metrics on which the orchestrator makes decisions on the mechanism to select at each execution cycle. On the top graph, the average window values for the detected and not-mitigated error rates are plot (for this experiment a window value of 20 cycles was chosen), while the dashed horizontal line at value 0.05 represents the reliability constraint specified by the user. In the bottom plot the average window value for the performance is shown. Therefore, the orchestrator selects a specific mechanism according to the current values of the metrics; the selected mechanism is reported by means of gray areas of different intensity.

By looking at all the graphs, it is possible to see how the proposed approach actually allows to adapt the behavior of the considered system to cope with the mutating fault distribution. Indeed, at the beginning of the execution, when the failure rate is relatively low, the orchestrator decides not to mitigate any fault, but only to detect them. When the number of detected errors becomes too high, the orchestrator performs a tuning phase, analyzing the performance of the techniques providing fault tolerance. Thus, in this first phase, DWRC at the finer level is selected to be executed alternatively to DWC at the coarser level. When



(a) Execution latency, and number of detected/not-mitigated errors at each cycle.



(b) Average execution latency and detected/not-mitigated error rates over the specified window.

Figure 4.12: Metrics computed over the overall experiment execution.

the failure probability increases, DWRC performance is no more satisfying and TMR at the coarser level becomes the chosen fault tolerant mechanism, to be executed in alternative to DWC. Finally, from about the 370^{th} execution cycle, the orchestrator identifies an even higher error ratio suspecting the presence of a permanent fault and definitively switches to TMR applied at the finer level to perform the fault diagnosis; if the response is positive, the orchestrator will switch-off the damaged PE and will apply again DWC.

As a final note, Figure 4.13 compares the throughput achieved by the presented approach with all the other statically selected mechanisms also with respect to the plain execution; on each bar the relative percentage value with respect to the plain execution is reported. As discussed above, DWC applied at both levels obtains best performance but it violates the reliability requirement on the error ratio. Then, when considering the other strategies, the orchestrator obtains an improvement of only the 5% with respect to the TMR at coarser level; however, this mechanism is not able, in the second part of the experiment, to perform fault diagnosis. Thus, a fair comparison can be performed only with TMR and DWCR applied at finer level since they offer all the discussed reliability properties; in such a scenario, the orchestrator obtains an improvement of 18% and 23% respectively, thus demonstrating its effectiveness. As a final note, the throughput of the plain execution is not considerably higher than hardened ones as it may expect. This is due to the fact that the number of threads generated by each fork is decided at design-time (equal to 4) and therefore it is not possible to fully exploit the architecture capabilities; a possible future development of the approach may consider the dynamic decision of the number of threads as also supported in OpenMP [95].

4.5 Final Remarks

In this chapter we introduced the design of an self-adaptive system and related orchestrator for fault management in multicore architectures, able to monitor and adapt itself in order to pursue the desired performance/reliability trade-off. The orchestrator is implemented in a fault management layer working on top of the operating system, in charge of providing reliability properties when desired. The proposed solution dynamically selects and applies fault detection/tolerance mechanisms to mitigate the effects of faults while optimizing performance. Experimental results performed in the image processing scenario show that the provided self-adaptability feature allows to achieve better performance

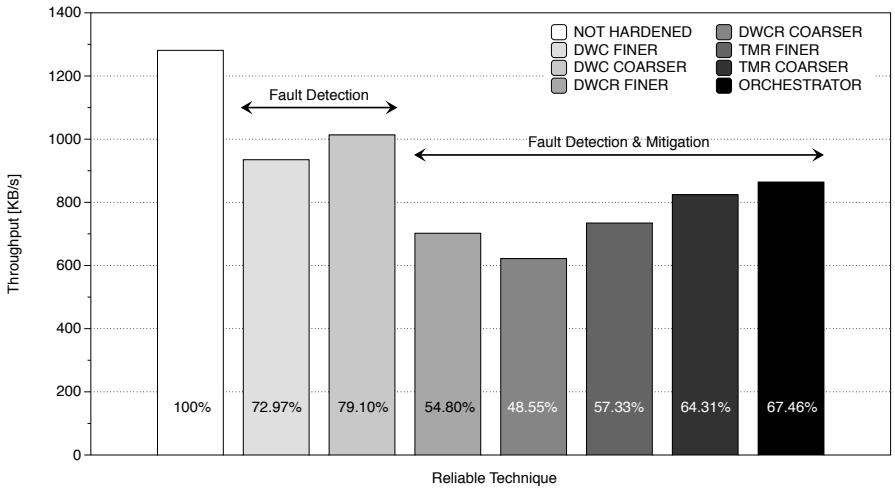


Figure 4.13: Throughput for the edge detector on the architecture with 12 processing cores, stimulated with a fault list presenting a variable failure rate and a permanent fault.

while fulfilling the reliability requirements with respect to traditional, static solutions.

In the next chapter a self-adaptive system for extending architecture lifetime through age mitigation (while minimizing energy consumption) will be presented. It is based on the same autonomic control loop paradigm, but exploits different monitors, decision policies and knobs to achieve such goal.

5 Runtime Aging Management

This chapter describes a novel self-adaptive system for mitigating aging in multi/manycore architectures. In the first part of the chapter the lifetime evaluation process will be presented and the design of a framework for lifetime estimation discussed. The remainder is devoted to the presentation of runtime resource management policies for extending architecture lifetime, both at the node and at the multi-node level, while minimizing energy consumption and meeting performance constraints.

5.1 Background

The first step in extending the system lifetime is to be able to measure it. This activity is particularly challenging when considering a complex multicore architecture with possibly multiple and subsequent PE failures, and when supporting runtime resource management strategies, whose effects on cores' aging and wear-out is a-priori unknown. Should sensors be available, it would be a matter of accessing their information. However, in a generic architecture such sensors are not available and it is necessary to estimate the aging of a core or a system by referring to the performed activity and the working conditions. When these aspects vary during the life of the system, the computation may be particularly complex. We here first discuss how we estimate the aging, then we introduce the tool to support such computation.

Reliability of a single system at time t, $R(t)$, is usually referred to as the probability that the system has been operational until t. As suggested by the JEDEC Solid State Technology Association [57], the lifetime reliability of a single digital component, such as a PE, can be modeled according to the Weibull distribution

$$R(t,T) = e^{-\left(\frac{t}{\alpha(T)}\right)^{\beta}}, \tag{5.1}$$

being t the current instant of time (generally measured in hours), T the (steady-state) PE temperature (Kelvin degrees), β the Weibull slope parameter (considered to be independent of the temperature), and $\alpha(T)$ the scale parameter [52]. The lifetime of the PE is estimated in terms of

its Mean Time To Failure (MTTF), defined as the area underlying the reliability function $R(t,T)$:

$$MTTF = \int_0^\infty R(t,T) \cdot dt. \tag{5.2}$$

The $\alpha(T)$ parameter formulation depends on the wear-out effects to be modeled, such as [57]: electromigration (EM), thermal cycling (TC), time-dependent gate oxide breakdown (TDDB), or negative-bias temperature instability (NBTI). As a proof of concept, EM related wear-out failures will be considered for the framework validation in the experimental results. In the EM model, Black's equation is used to compute $\alpha(T)$:

$$\alpha(T)_{EM} = \frac{A_0(J - J_{\text{crit}})^{-n} e^{\frac{E_a}{kT}}}{\Gamma\left(1 + \frac{1}{\beta}\right)} \tag{5.3}$$

where A_0 is a PE-dependent constant, J is the current density, J_{crit} is the critical current density for the EM effect to be triggered, E_a is the activation energy for EM, k is the Boltzmann's constant, n is a material-dependent constant, β the slope parameter of the Weibull distribution and $\Gamma()$ is to the gamma function. However, it is possible to integrate other effects either as standalone contributes or by using the sum-of-failure-rates (SOFR) approach for any combination of the above failure effects [90]. In fact, in the second part of the chapter (when DVFS comes into play), thermal cycling will also be considered. Coffin-Manson equation [57] can be used to model TC effects and $\alpha(T)$ can be written as:

$$\alpha(T)_{TC} = C_0(\Delta T - \Delta T_0)^{-q} f. \tag{5.4}$$

where ΔT represents the temperature cycling range and ΔT_0 the elastic portion of the thermal cycle (typically, $\Delta T_0 << \Delta T$, thus ΔT_0 can be dropped from Equation 5.4); C_0 is a material dependent constant, q is the Coffin-Manson exponent, and f is the frequency of thermal cycles. In conclusion, by exploiting the SOFR model, $\alpha(T)$ due to both TC and EM is given by the sum of $\alpha(T)_{EM}$ and $\alpha(T)_{TC}$.

To formulate $R(t)$ for the general scenario of a multicore system sharing a variable workload, the above formulas need to be revised to consider two specific aspects: i) temperature changes due to a variation in the workload, and ii) system composition in terms of its resources and its capability to survive beyond the first failure. These two aspects are here now briefly presented in relation to the scenario of the proposed approach; for a more precise formulation, it is possible to refer

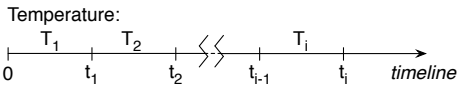


Figure 5.1: Model of a PE's temperature profile over time.

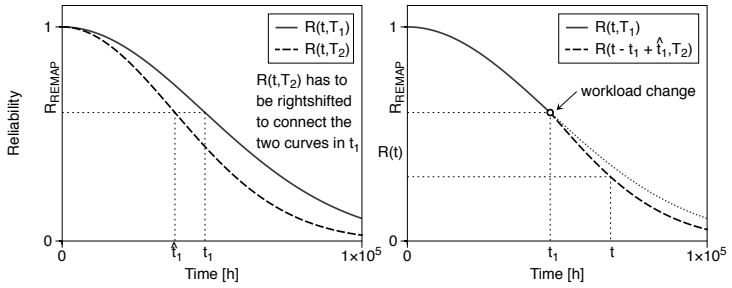


Figure 5.2: Reliability curve considering temperature changes.

to [69, 51]. When the temperature changes as shown in Figure 5.1 (to keep the computation manageable, steady-state temperatures are considered), the exact computation of the $R(t)$ function is computed using Equation 5.1 and its inverse function, $R^{-1}(r,T)$, defined as:

$$R^{-1}(r,T) = \alpha(T) \cdot (-\log(r))^{1/\beta} \tag{5.5}$$

being r a reliability value. The $R^{-1}(r,T)$ function returns the elapsed time t, starting from time 0, when the system reaches the reliability value of r, under the influence of a fixed temperature T. In particular, the $R(t,T_1)$ function can be directly applied only to a new PE (i.e., when $t < t_1$, being t_1 the time when a load redistribution takes place causing a change in the steady-state temperature). On the other hand, when $t \in (t_1, t_2]$, the PE reliability is equal to $R(t - t_1 + \hat{t}_1, T_2)$, where $\hat{t}_1 = R^{-1}(r_1, T_1)$ and r_1 is the reliability value in t_1 [69]. Similarly, the system's reliability for values of t in the next steps can be computed with a recursive approach by using the above-mentioned formulas. From a graphical point of view (shown in Figure 5.2), $R(t,T)$ is a curve starting from $t = 0$ (and $R(0,T) = 1$) with a varying shape that depends on the temperature T. Therefore, at each temperature change it is necessary to right-shift the new reliability curve in order for it to intersect the old reliability value. Finally, the $R(t)$ formula considering a variable PE's

temperature can also be rewritten in the following way [101]:

$$R(t) = e^{-\left(\sum_{j=1}^{i} \frac{\tau_j}{\alpha_j(T)}\right)^{\beta}}. \tag{5.6}$$

where τ_j represents the duration of each period of time with constant temperature T_j up to time t (i.e., $t = \sum_{j=1}^{i} \tau_i$).

All the equations discussed so far refer to the case of a system with a single PE. Let us now move to a system consisting of n PEs subject to a sequence of k failures, causing a redistribution of the workload after each failure occurrence. Such scenario is usually referred to as a load-sharing k-out-of-n:F system, made up of non-independent and non-identically distributed (non-i.i.d.) variables with arbitrary distribution. k ($\leq n$) is the minimum number of PEs that have to fail before the entire system becomes out-of-service. The reliability of such system is computed by using the total probability formula:

$$R(t) = P_{no\_f}(t) + P_{1\_f}(t) + \cdots + P_{k-1\_f}(t). \tag{5.7}$$

$P_{no\_f}(t)$ is the probability that no failure occurred and is computed as the product of the reliability of all PEs:

$$P_{no\_f}(t) = \prod_{i=1}^{n} R_i(t). \tag{5.8}$$

We call the MTTF to the first failure the integral between 0 and $+\infty$ of such formula; as discussed above, several works [50, 58, 29, 54] approximate the system MTTF to such value. $P_{1\_f}(t)$ is the sum of the probabilities of occurrence of n different cases where a single PE fails and its load is redistributed onto the other working units:

$$P_{1\_f}(t) = \sum_{i=1}^{n} \int_0^{\infty} f_i(t_1) \cdot \prod_{j=1, j\neq i}^{n} R_j(t|\mathbf{t_1^i}) dt_1 \tag{5.9}$$

where $f_i(t)$ is the probability density function of $R_i(t)$, i.e., the probability that the failure occurred on PE i at a specific time t_1, and $R_j(t|\mathbf{t_1^i})$ is the conditioned reliability function of PE j knowing that PE i failed at time t_1. It is worth noting that conditioned reliability functions can be computed by using the formulas for load remapping, as shown in [69]. Similarly, $P_{2\_f}(t)$ is computed as:

$$P_{2\_f}(t) = \sum_{i=1}^{n} \sum_{j=1, j\neq i}^{n} \int_0^{\infty} \int_0^{t_2} f_i(t_1) \cdot f_j(t_2|\mathbf{t_1^i}) \cdot \tag{5.10}$$

$$\cdot \prod_{m=1, m\neq i\neq j}^{n} R_m(t|\mathbf{t_1^i}, \mathbf{t_2^j}) \cdot dt_1 \cdot dt_2 \tag{5.11}$$

by using probability density functions and the reliability function conditioned by the previous sequence of failures, the first one occurred at t_1, the second one at $t_2 > t_1$. Finally, the general scenario considering the sequence of $k-1$ failures is a $(k-1)$-dimensional integral that can be written recursively on the basis of Equations from (5.7) to (5.10).

5.2 Aging Evaluation

The existing solutions for aging evaluation and lifetime estimation ([50, 58, 101, 43]) are either too simplistic (e.g., they consider the system to fail after the first PE fails) or are limited in the features (e.g., homogeneous workload distribution and static design-time mapping), not allowing the designer to fully estimate the system's lifetime in a dynamic working scenario. In the following we report a review of the literature highligthing the limitations our proposal aimed at overcoming.

5.2.1 State of the Art

Devices aging and wear-out mechanisms (such as EM, TC, TDDB, and NTBI) have been accurately studied and modeled [57]. These models usually consider a homogeneous device and assume a single steady-state operation mode at fixed (worst-case) temperature. These aspects constitute a significant limitation when taking into account modern systems composed of many processors, working in highly dynamic contexts, in terms of both the executed workload and its distribution on the PEs, elements that highly affect system's aging rate.

One of the first system-level models for lifetime reliability evaluation is presented in [90]; it uses a SOFR approach to take into account several aging mechanisms. The proposed approach, as well as subsequent ones, considers a single core architecture [33] or adopts exponential failure distributions [58] that are though unable to capture accumulated aging effects. As a result, they are not suitable for multicore architectures supporting load-sharing.

When using Weibull or lognormal distributions to consider the aging history, as in [50] and [101], an important issue related to the computational complexity of the lifetime reliability evaluation arises. In fact, in each instant of time, the reliability of each component is a function of the current working conditions and the aging effects accumulated in the previous periods of activity. Actually, the simulation of the overall life of the system capturing the workload evolutions is a prohibitive time-consuming activity. Therefore, many approaches [58, 50, 101] simulate and trace only a subset of representative workloads for a reduced period

	[90]	[33]	[58]	[50]/[54]	[101]/[51]	[29]	[43]
Architecture	S	S	**M**	**M**	**M**	**M**	**M**
Aging Model	**W**	**W**	E	**W**	**W**	**W**	sensors
Workload	static	Avg T	Avg T	Avg α	Avg α	Avg T	sensors
Toler. Faults	×	×	×	×	✓	×	✓

Table 5.1: Comparison of the state the art frameworks (*Architecture*: single core (S), multicore (M) – *Aging Model*s: exponential distribution (E), Weibull distribution (W) – *Workload*: average temperature (Avg T), average aging factor (Avg α) during the considered period).

of time with a fine granularity, then extrapolating the average temperature [58] or an average aging rate [50, 101] to be considered during the MTTF computation. However, while the former strategy is highly inaccurate (as shown in [50]), the latter is able to capture only workload changes in the short period of time considered for the simulation; they do not capture the evolution of the lifetime reliability curve due to changes that fall outside the simulation window.

Another issue in the computation of the lifetime reliability is the fact that the system may tolerate a given number of failures by remapping the workload from the failed PE to the healthy ones. As shown in some theoretical works [69, 51, 81], the system reliability formula is based on a multidimensional integral, whose dimension is given by the number of failures the system can tolerate. In practice, its analytical solution is unfeasible because the direct numerical computation is not affordable in terms of execution time. As a consequence, many approaches [50, 58, 29, 54] consider the entire system not to survive beyond the first failure, while others [42] compute the MTTF iteratively by selecting at each step the lowest MTTFs. Both solutions are inaccurate.

Finally, other approaches [51, 101, 43] adopt Monte Carlo simulation to quickly and accurately compute the multidimensional integral. However, they take into account only a "static" working scenario, with lack of generality and offering only a theoretical discussion of lifetime reliability estimation. In this work, we aim at overcoming such shortcomings. Table 5.1 summarizes the features of the cited state of the art frameworks.

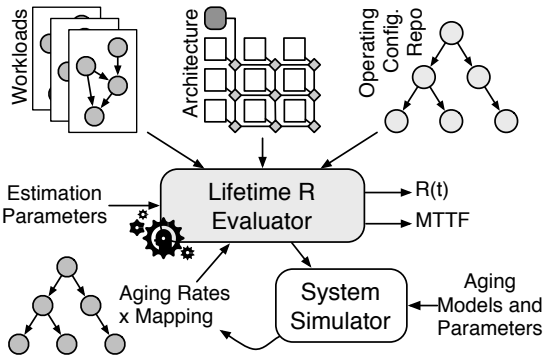


Figure 5.3: Workflow of the proposed framework.

5.2.2 The proposed framework

The preliminary discussion on the MTTF computation has shown that the direct numerical evaluation of the reliability of a k-out-of-n:F system subject to a variable workload is computationally feasible only for small values of k and, therefore, motivates once more the proposed approach. This is the reason why we developed CALIPER (monte CArlo LIfetime p reliability EstimatoR) [17]. It supports: i) common failure mechanisms that model component failures as non-independent and non-identically distributed (non-i.i.d.) variables, ii) the computation of the lifetime of a system composed of n processors sharing a workload and able to tolerate a given number of k failures by means of load redistribution, and iii) the reliability curve analysis for long-period evolving workload scenarios. A block diagram for the framework is shown in Figure 5.3. It takes as inputs the specifications of i) the system's architecture, ii) the workloads (made up of a set of applications), iii) the failure mechanisms of interest, and iv) the representative operating configurations. In particular, the architecture and application specifications are models that can be simulated, with the former one being characterized from a thermal point of view; moreover, the parameters characterization of the failure mechanisms is to be provided. The last input is the description of all possible working conditions of the architecture, corresponding to different healthy/failed PEs with different workload distributions. These operating configurations are organized in a tree-based structure, called *operating configuration repository*, where the root represents the initial architecture and each path starting from it is a sequence of PE failures to be considered; therefore, first-level nodes correspond to single PE failures, i-level nodes to i failed PEs. The depth of the tree is bounded by k failures, according to the

designer's specifications. Each node (called *operating configuration*) in the tree is annotated with a set of mappings, each one describing how the workload is distributed on the healthy PEs. There will be a single mapping in the scenario where changes in the workload distribution are only triggered by a PE's failure. Otherwise, when a dynamic workload scenario is envisioned, the time plan of the various workloads will be provided, by listing the set of considered mappings, each one with its own duration.

CALIPER is internally organized into two modules: the *lifetime reliability evaluator* and a supporting *system simulator*. The first module, the core of the proposed work, performs the lifetime reliability evaluation for the overall system by means of Monte Carlo simulations; more precisely, it estimates the system's reliability curve $R(t)$ and its expected lifetime, in terms of MTTF. To perform such computation, the evaluator module requires the aging characterization for the architecture's PEs. This characterization must be computed under the influence of the executed workload in all operating configurations (and related mappings) considered in each specific Monte Carlo simulation. Therefore, the *system simulator* is invoked once for each considered mapping to estimate the thermal profile of the PEs and to derive their aging rates $\alpha(T)$. The internals of the two modules are described in details in the following.

Aging rates estimation

The simulation engine is a service module used to characterize the lifetime reliability of the system at processor level; it receives as input the operating configuration and the mapping, and produces as output the reliability characterization of each PE within the architecture for that scenario, in terms of the average aging rate. To this end, the main activity of this module is to estimate the thermal profile of each PE with respect to the given workload distribution.

Within the same operating configuration mapping, PE's temperature varies significantly from application to application and, at a finer granularity, with different applications' tasks; moreover, it is also affected by thermal interferences with neighbor units on the architecture floorplan. As an example, Figure 5.4 reports the thermal profile associated with the execution of a given application, modeled by a periodic task graph, with respect to a given mapping (containing also the scheduling information): the temperature of a PE depends also on the activity of the neighbor units, running their part of the application. Thus, architecture and application models have to capture all these aspects; more details will be provided in Section 5.2.3 for the specific solution we adopted in

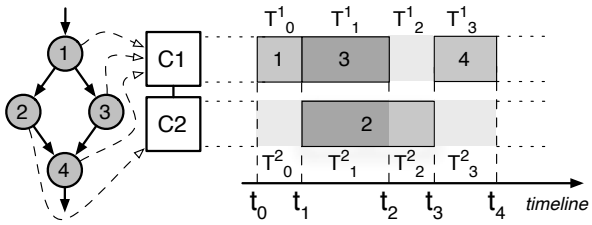


Figure 5.4: Example of application execution and thermal profile.

the proposed framework prototype.

Since for each mapping in the operating configuration repository the workload is almost stationary, as shown in [101], it is not necessary to simulate the system for its complete life and to compute the reliability curve by means of Equation 5.6; such activity would be actually computationally non affordable. Indeed, it is possible to compute the average aging rate by tracing the system execution for a representative time period and by using the following formula:

$$\alpha = \frac{\sum_{i=0}^{p} \tau_i}{\sum_{i=0}^{p} \frac{\tau_i}{\alpha_i(T)}} \tag{5.12}$$

where τ_i represents the duration of the p atomic steps (with constant temperature on each PE) within the simulated period. Then, the reliability evaluator uses such values as input parameters in Equations 5.1 and 5.5 to derive the lifetime reliability of the overall system. As a final note, even if simulated times may span from few microseconds to several hours, the result is the average aging per time unit, since Equation 5.12 is a ratio between the accumulated aging rates and the time window; considering the units of measure of the various parameters, the α value is generally referred per hour.

Reliability computation

The second module of the framework is devoted to the evaluation of the lifetime reliability curve $R(t)$ of the entire system and the related MTTF. Since numerical evaluation of the model presented in Section 5.1 is computationally unsustainable for more than 4 subsequent failures (as shown in Section 5.2.3), this activity is performed by means of Monte Carlo simulations to greatly reduce the computational time while obtaining accurate results. In the following paragraphs, the basic approach is discussed in details, and two improvements are then presented: i) the

automatic tuning of the number of simulations needed to obtain a certain confidence in the obtained results and ii) the handling of variable workloads.

The basic scenario we consider (non-i.i.d. load-sharing) is characterized by a change in the workload distribution on the healthy PEs only when a PE fails. Multiple subsequent failures are considered: the whole system fails when the number of healthy PEs is below a threshold specified by the designer. The pseudo-code for the lifetime computation is shown in Algorithm 1 and here briefly explained. Before getting into the details, for readability's sake, it is worth redefining the two functions $R()$ and $R^{-1}()$, describing the reliability of a single PE, to take as input the α parameter, computed by means of the system simulator, in place of the temperature T. The inputs of the algorithm are: i) the number `n` of PEs in the architecture, ii) the number of failures `k` that causes the entire system to become out-of-service, and iii) the number of simulations to be performed `#iter`.

A single Monte Carlo test consists of a simulation of a randomly-chosen sequence of PE failures' events; the goal is to compute the time to failure (TTF) of the overall system during that specific simulation. To compute the lifetime reliability, the same procedure must be iterated `#iter` times. After an initialization phase (Lines 2–5), the random walk step takes place and is repeated until the number of healthy PEs reaches `n-k` (Lines 7–27). Starting from a completely healthy architecture, for each iteration a specific subset of healthy PEs with a given workload distribution is selected and a failing event happens; the PE failing at time t is randomly chosen among the healthy ones. This failure triggers the update of the system reliability based on the ones of the remaining healthy PEs. To compute the α values for each step, the `getAlpha()` function is used; it invokes the simulation engine by specifying the current operating configuration `currConfig`. For all the healthy PEs `i`, a new reliability value `randR` is randomly chosen within $[0, currR[i]]$ (Line 11): it represents the reliability at which that PE will fail. The corresponding time instant `t` (all times are measured in hours) for such failure is computed (Line 12). Moreover, the time period between the current failure and the subsequent one needs to be computed, by evaluating the equivalent time `eqT` and subtracting it to `t` (Lines 13–14, please refer to Section 5.1 for the theoretical details). After having computed the failure times for all healthy PEs, the smallest one is selected to identify the next event, i.e., the next PE to fail; the simulation is thus advanced until such `t`, the PE is removed from the list of healthy ones (Lines 21–22), and the reliability values of all remaining PEs are updated (Line 24), to begin with a new iteration. At the end of each simulation, the value called

$\texttt{MTTF}_\texttt{m}$ is updated by computing the sample mean of the system's TTF values experienced in the last m iterations (Line 29).

When all the simulations have been performed, collected data are summarized by generating the system reliability curve (Lines 32–39); for each time t corresponding to a failure, the value of R is computed as the ratio between the number of simulations with time greater than the current time and the total number of performed simulations. Finally, the algorithm displays the reliability curve and returns the $\texttt{MTTF}_\texttt{m}$, representing the MTTF of the system updated to the last iteration.

Since Monte Carlo simulations are a stochastic approach, it is necessary to assess a certain level of accuracy in the obtained results. Thus, the basic algorithm has been enhanced to automatically tune the number of tests `#iter` to be performed to obtain a given confidence level. The program iterates the simulations until either a specified percentage $c\%$ of the width of the confidence interval for the $MTTF_m$ value is less than a given threshold th, or the maximum number of iterations `#maxiter` is reached (to avoid starvation). The first condition is expressed as:

$$2 \cdot \Psi^{-1}\left(\frac{1+c\%}{2}\right) \cdot \frac{c_v[TTF_m]}{\sqrt{m}} \leq th \tag{5.13}$$

where m is the number of performed tests, $c_v[TTF_m]$ is the coefficient of variation of the m TTF samples considered (i.e., the same ones used to estimate $MTTF_m$), computed as $\frac{\sqrt{Var[TTF_m]}}{MTTF_m}$, and $\Psi^{-1}(x)$ is the inverse of the cumulative normal distribution. $c\%$, th and `#maxiter` are received as input in place of `#iter`.

Frequent workload remappings can be approximated by using average aging rates as demonstrated in literature [50]; however, when considering long-term workload changes, such method may not be accurate enough. To this purpose, the framework supports the specification of a sequence of mappings for each operating configuration, each one provided with its duration. Algorithm 1 has been slightly revised to support such aspect; more precisely, the computation of the new reliability value (Line 12) is replaced with the code snippet listed in Algorithm 2: it performs a step-by-step update of the reliability value and related elapsed time by using $R()$ and $R^{-1}()$ functions (as already described in Section 5.1) based on the specified workload time plan. Do note that `getCurrMapping()` and `getTimeToNextChange()` functions return the mapping and the related period for the current operating scenario. Correspondingly, the reliability computation of the remaining healthy PEs upon a PE failure (Line 24 in Algorithm 1) needs to be adapted as well, with a similar strategy.

Algorithm 1 Monte Carlo-based reliability computation

1: **for** $m = 1$ **to** $\#iter$ **do**
2: $currConfig = getInitConfig()$
3: $healthy = \{0, 1, ..., n-1\}$
4: $currR = \{1, 1, ..., 1\}$
5: $totalTime = 0$
6: $TTFAccumulator = 0$
7: **while** size($healthy$) $> n - k$ **do**
8: $failingPE = -1$
9: $alpha = getAlpha(currConfig)$
10: **for all** i **in** $healthy$ **do**
11: $randR = \text{random}() \cdot currR[i]$
12: $t = R^{-1}(randR, alpha[i])$
13: $eqT = R^{-1}(currR[i], alpha[i])$
14: $t = t - eqT$
15: **if** $failingPE == -1$ **or** $failTime > t$ **then**
16: $failingPE = i$
17: $failTime = t$
18: $prevEqTime = eqT$
19: **end if**
20: **end for**
21: $totalTime = totalTime + failTime$
22: $healthy.remove(failingPE)$
23: **for all** i **in** $healthy$ **do**
24: $currR[i] = R(failTime + prevEqTime, alpha[i])$
25: **end for**
26: $currConfig = updateConfig(failingPE)$
27: **end while**
28: $TTFAccumulator = TTFAccumulator + totalTime$
29: $MTTF_m = TTFAccumulator/m$
30: $stats[totalTime]++$
31: **end for**
32: $currHealthyIter = \#iter$
33: $precT = 0$
34: **for all** t **in** $stats.keys()$ **do**
35: $currHealthyIter = currHealthyIter - stats[entry]$
36: $currRvalue = currHealthyIter/\#iter$
37: $precT = t$
38: **print** t, $currRvalue$
39: **end for**
40: **return** $MTTF_m$

Algorithm 2 Computation of the next PE to fail for dynamic workload changes

Input: $currConfig$, $randR$, $currR[i]$, $alpha[i]$, $totalTime$
Output: t

1: $currMapping = getCurrMapping(currConfig, t)$
2: $period = getTimeToNextChange(currConfig, t)$
3: $t = totalTime$
4: $alpha = getAlpha(currConfig, currMapping)$
5: $eqT = R^{-1}(currR[i], alpha[i])$
6: $testR = R(eqT + period, alpha[i])$
7: **while** $testR > randR$ **do**
8: $t = t + period$
9: $currR[i] = testR$
10: $currMapping = getCurrMapping(currConfig, t)$
11: $period = getTimeToNextChange(currConfig, t)$
12: $alpha = getAlpha(currConfig, currMapping)$
13: $eqT = R^{-1}(currR[i], alpha[i])$
14: $testR = R(eqT + period, alpha[i])$
15: **end while**
16: **if** $currR[i] > randR$ **then**
17: $eqT = R^{-1}(currR[i], alpha[i])$
18: $lastT = R^{-1}(randR, alpha[i])$
19: $t = t + lastT - eqT$
20: **end if**
21: **return** t

5.2.3 Experimental evaluation

A SystemC discrete event simulator (with a structure similar to [54, 36, 21, 50]) has been developed, while HotSpot [87] has been used to characterize the architecture's thermal profile. Do note that CALIPER can be integrated in a state-of-the-art framework, such as the one in [32].

The system considered in the experimental sessions is a mesh-based System-on-Chip architecture with $N \times M$ PEs, spanning from 2×1 to 3×4. The system executes a workload that is uniformly distributed among the healthy PEs, such that all PEs have a 50% occupation in the initial healthy architecture. The workload is then redistributed after the occurrence of each failure and/or a given period. In the experiments, we consider the system to be out-of-service when, after a remapping, the workload per PE is higher than 100%. Finally, the EM-related parameters have been borrowed from [54].

Approach Validation

The approach has been validated against the exact formulation presented in Section 5.1 based on multidimensional integrals. The exact model has been solved by means of numerical methods (step discretization and rectangles). The discretization step has been set to 500 hours, causing an overall negligible error, since the $R()$ curve is almost equal to zero at $400,000$ hours, in a 1-out-of-2:F scenario. Due to the high computational complexity of exact approaches, validation against them has been feasible for architectures with at most 2×3 PEs, tolerating up to 3 failures. In all the performed tests, the observed difference between the MTTF computed by means of exact approaches and the one computed by means of CALIPER is lower than 1%; do consider that such error may also be related to the discretization step. All the scalability experiments discussed in this section have been run with a constant workload throughout the whole system lifetime.

The main scalability problem of the exact method is related to the $(k-1)$-dimensional integral to be solved for a system failing after k failures (Equation 5.7). Such an integral has an asymptotic complexity $O(t^k)$, being t the overall simulation duration. In fact, when considering a 2×3 architecture, the exact method took less than 3 seconds for computing the MTTF for $k = 1$ or $k = 2$ failures; the computation time increased to 1 minute and 10 seconds for 3 failures, while requiring more than 2 hours for 4 failures. On the other hand, the execution complexity of CALIPER scales almost linearly w.r.t. the number of

performed iterations and the number of simulated failures, as shown in Figure 5.5, which considers a 3×4 architecture. We measured the same error ($< 1\%$) in the experiments for which we were able to perform the validation. Do consider that aging rates have been precomputed and tabulated in an input file to perform an analysis of the performance of the Monte Carlo simulations approach only.

It is also interesting to analyze how the number of iterations for a fixed width of the confidence interval ($c\% = 95\%$) and threshold parameter ($th = 0.1$) changes with reference to the considered architecture and the number of failures k. Figure 5.6 shows that the number of iterations is almost constant if k is fixed, regardless of the considered architecture's dimension and topology, and more important, it decreases as k grows. This could seem to be a surprising result, but the explanation is actually intuitive: as it can be seen in Equation 5.13 the width of the confidence interval (i.e., the left side of the inequality) is proportional to the coefficient of variation $c_v[TTF_m]$. By increasing the number of failures, the coefficient of variation decreases and the number of required iterations for the algorithm to converge decreases as well. Figure 5.7 empirically proves this trend. A theoretical proof of this behavior can be provided for the exponential case, since the corresponding distribution becomes hypoexponential; the same idea holds for the Weibull case.

Constant workload scenarios: comparison

CALIPER has been compared against some state-of-the-art approaches assuming a constant workload scenario:

- the Mean Time To First Failure (FF), adopted in [50, 58, 29, 54], where the whole architecture is considered to be out-of-service when the first PE fails. In this case the reliability is computed by exploiting Equation 5.7;
- the sum of MTTF (Sum), proposed in [42]: in this case, at each step, the PE selected to fail is the one with the lowest MTTF, the reliability for the healthy PEs is updated and a new step is computed, until the architecture cannot reach the required performance;
- the worst case temperature (wc) [57]: at each step, the worst case temperature in the architecture is considered for computing the system aging.

Figure 5.8 reports the obtained results for such a comparison. While all the considered approaches have comparable execution times (same trend

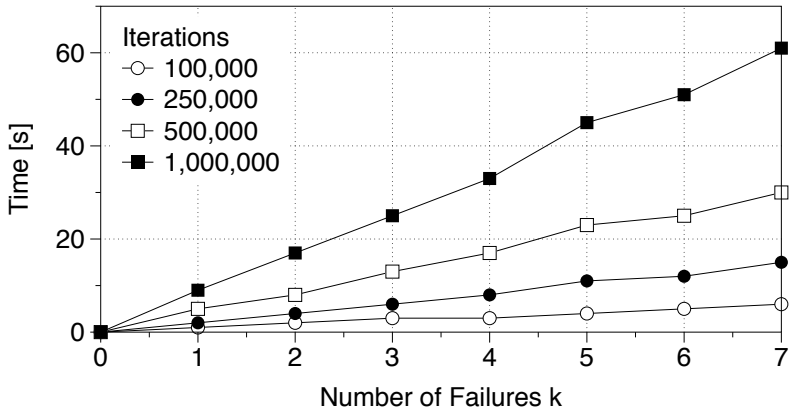


Figure 5.5: Simulations execution times w.r.t. number of failures k.

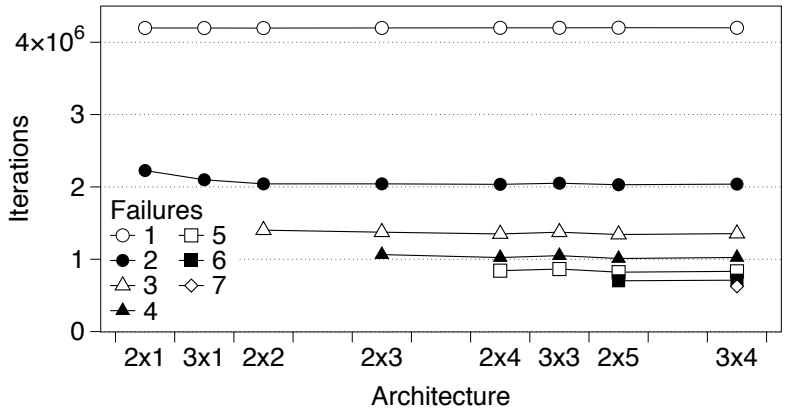


Figure 5.6: Iterations number w.r.t. different architectures topologies.

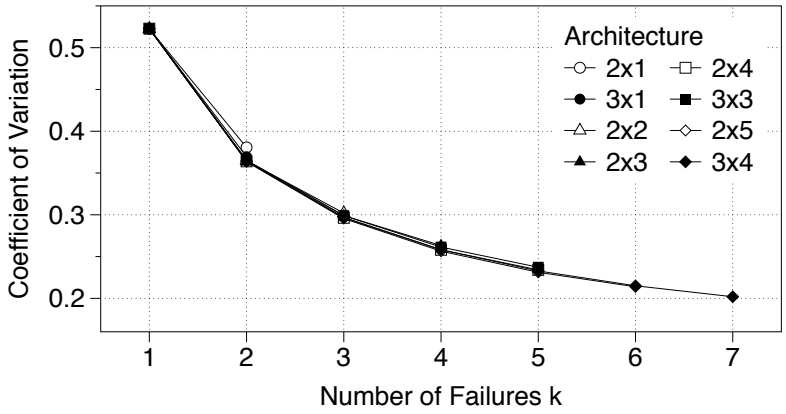


Figure 5.7: Coefficient of variation w.r.t. number of failures k.

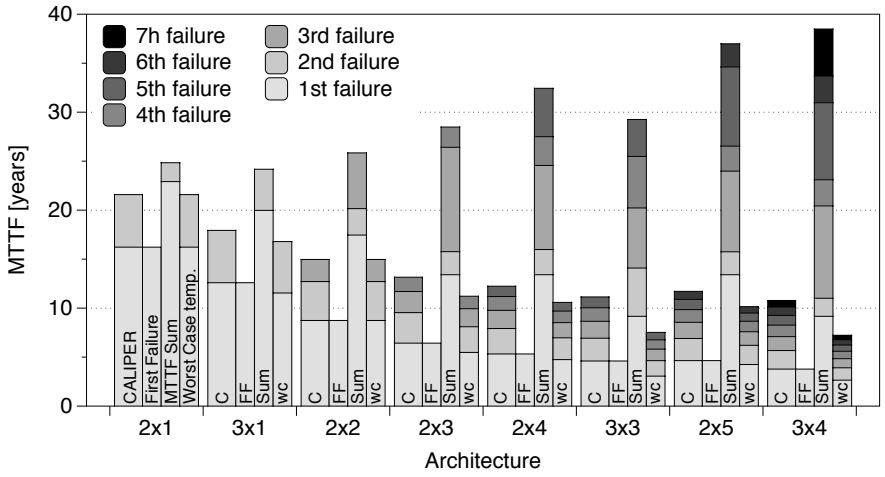


Figure 5.8: Comparison vs. past works for constant workloads.

for *FF* and *wc* and around 1 second for *sum*), estimated lifetimes are very different. The simplification considering only the first failure clearly underestimates the real estimated lifetime of the architecture. As expected, the error increases as the number of PE failures k increases: from 33.1% (1-out-of-n:F) to 185.4% (6-out-of-n:F). The *sum* approach proves to be even less precise, by highly overestimating the evaluated system lifetime. Again, the error range depends on k, reaching 256.9% for the 6-out-of-n:F case. These two approaches show trends in the successive failures that are not proportional at all; this means that the results produced by the *sum* approach are affected by a considerable non-systematic error. *wc* is the approach that provides the closest results to the CALIPER ones. The lifetime is here underestimated as well, but the error is significantly smaller, at an average of 14.2%. It is worth noting that this approach obtains exact results when the worst case temperature corresponds to the actual one: this is usually true for very small architectures (namely, 2×1 and 2×2 in our experiments), since, when the workload is uniformly distributed, temperatures of all PEs are almost the same.

In conclusion, as a side note, it is interesting to highlight how, given the same starting workload for each PE in each architecture, smaller architectures have longer lifetimes. This is due both to the higher thermal interferences that take place in bigger architectures and to the fact that the actual overall workload is bigger; in fact, if the same workload had been used for all the experiments, bigger architectures would have had longer lifetime than smaller ones, however they would be highly re-

dundant with reference to the workload and the cost would have been unjustified.

Varying workload scenarios: comparison

The aim of the last experimental session is to show how CALIPER is able to provide more accurate estimations of lifetime reliability than state-of-the-art techniques in scenarios considering varying application workloads. Here a 3×4 architecture is considered, with 5 as a maximum number of tolerated failures and a uniformly distributed workload periodically changing among 60%, 40%, and 15% single PE utilization of the initial architecture. Figure 5.9 shows the reliability curves for various workload-change period spanning from 1 month to 5 years and compares them against average aging rates computed based on Equation 5.12 [101, 50]. From the figure it is possible to note how the approach in [101, 50] is able to accurately approximate workload changes with periods up to some months (curve oscillations are actually imperceptible), but it fails with longer periods. Obtained MTTFs slightly differ w.r.t. the average approximation up to 9% with the 5-year period; however, the maximum difference in the reliability curve with the average approximation spans from 0.07 of the 1-year period, to 0.15 of the 3-year period, to 0.19 of the 5-year period (over the $(0, 1]$ reliability range). Do note that having a precise estimation of the shape of the reliability curve is fundamental, for example, in computing which are the most probable failure periods, beside the estimated MTTF. In practical scenarios, the higher accuracy of the proposed tool has a considerable relevance in the overall evaluation of systems expected to experience highly different working phases during their operational life, with long-term periods with intensive workloads and periods of resources' underutilization.

Once it is clear how to evaluate the lifetime of an architecture consisting of several processing elements having independent characteristics, it is possible to move the research attention to the optimization side. The rest of this chapter will be devoted to the design of algorithms for the extension of architectures lifetime. In particular, the focus will be first on mitigating the aging at the node level, then moving to multi-node architectures.

5.3 Aging Mitigation through Self-Adaptiveness

When dealing with components wear-out mechanisms (which are mostly related with temperature) another dimension is to be carefully taken into

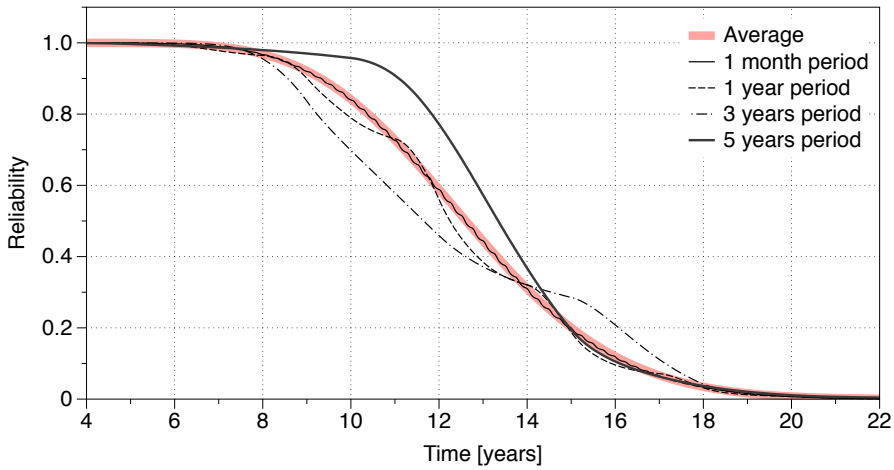


Figure 5.9: Reliability curve for different workload change's periods.

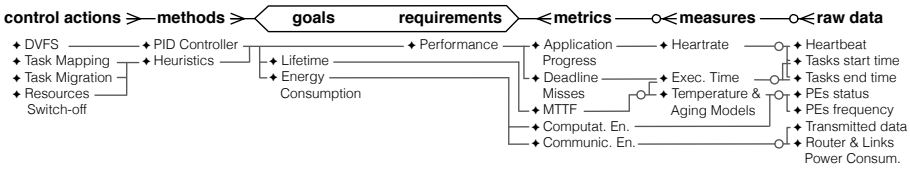


Figure 5.10: Context dimensions and domains for the aging mitigation self-adaptive system.

consideration together with performance: energy consumption. In fact, all the knobs exploited for aging mitigation impact energy consumption as well: the selection of the kind of PE on which to run the application, the frequency at which the selected PE is set, and even the position of the select PE influence the architecture energy consumption, both in terms of computation and communication energy. Figure 5.10 gives a representation of the self-adaptive system that will be presented in the remainder of this chapter by means of the model proposed in Chapter 3.

Shrinking transistor geometries and aggressive voltage scaling are negatively impacting the dependability of the processing elements and the communication backbone of multi/manycore systems [26]. Permanent device defects have gained a lot of research focus over the past decades due to their adverse effects in the deep sub-micron technologies. Quite a few research works were directed towards application mapping on many-

core platforms with the objective of balancing the temperature of the cores [27, 28, 96]. Although lifetime reliability of a PE is closely related to temperature, other aging factors, such as operating frequency, voltage and current-density, are not captured. If resource management decisions are taken without keeping components aging explicitly into consideration, some PEs can age faster than others, thereby reducing the operational life of a system. As already mentioned, energy consumption represents another key point to be considered in the lifetime optimization of multi/manycore architectures. Architecture energy consumption is usually considered to be devided into two main contributions: computation and communication energy. The former one refers to the energy spent by the PEs for running the applications on the available PEs; the latter one considers the energy needed to power up routers and links to make the data trasmission among different PEs possible.

Asymmetric architectures have already been proven to be beneficial in improving performance and computation energy consumption [89]. However, if the adopted policies do not take into account how they affect the system lifetime, the architecture reliability might be significantly affected. On the other hand, those approaches that focus only on lifetime improvement ([54], [74]) may lead to non-optimal solutions in terms of energy consumption. This is due to the fact that the selection of the execution resource is driven only by the health state of the architecture and aggressive frequency scaling tend to be avoided. This is the reason why a combined on-line lifetime/energy optimization self-adaptive system is to be built. The idea of relying on an existing energy optimization framework is motivated by the possibility of easily merging its decisions with reliability-driven ones, as better explained in the following.

It may seem that the optimization of the computation energy consumption would naturally lead to the improvement of the system lifetime, being both aspects mainly related to temperature. As shown in Figure 5.11, the side-effects of the strategies purely devoted to the energy minimization have a negative impact on lifetime, and viceversa. The shown results are referred to a set of five benchmarking applications (characterized by different shapes in the graph and presented in the experimental evaluation section) running on the reference asymmetric architecture (refer to Section 2.2.1 for further details). All the applications have been executed with an approach optimized for energy-consumption only [89] (black shapes) or for lifetime improvement only [20] (white shapes). The dashed lines in the plot connect the points sharing the same workload, where the leftmost point represents the best result for energy consumption and the rightmost the best one for lifetime. In between, an infinite set of points exploiting the trade-off among the two quantities

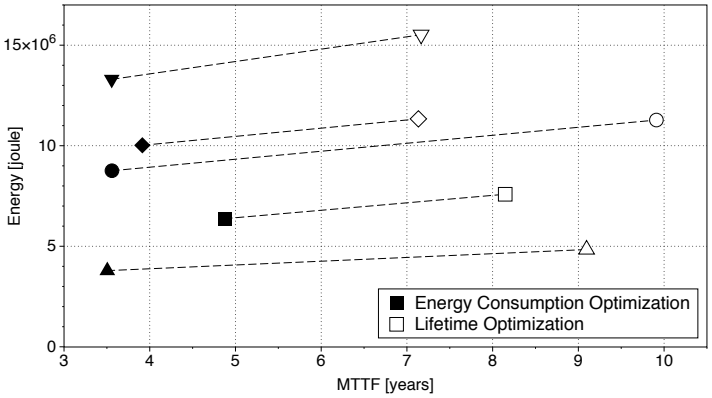


Figure 5.11: Energy consumption vs. lifetime optimization.

can be found. Therefore, a new solution taking into account both goals is necessary and for this purpose we have defined a *resource management self-adaptive system*, designed as an evolution of the solution presented in [89], which is focused on a power management technique for asymmetric multicores that can provide satisfactory user experience while minimizing energy consumption. We aim at building our lifetime optimization framework on the top of it, to make it aware of the components aging and exploit resource management so to extend the overall system lifetime without negatively impacting energy consumption. In particular, among all the wear-out phenomena affecting electronical components, we explicitly target electromigration and thermal cycling, even if the adopted model can straightforwardly integrate other effects. To mitigate their effects, the proposed framework acts on applications mapping and on the voltage/frequency points at which the different resources work (DVFS). Details on the work here described can also be found in [22].

5.3.1 Single Node

Before getting into the details of the proposed system, it is worth clarifying a few points. Serial applications only are considered in this work, possibly characterised by various *phases*. Each application i is characterized by an expected performance constraint qos_i, expressed by means of a Quality of Service (QoS) measure, for which we exploited the concept of *heartrate* [45]. It is defined as the throughput of the critical kernel in an application, such as the number of frames per second for a video encoder application. This value is provided together with the application

at its arrival time T_i, assumed not to be known a-priori. The architecture must try to meet the given performance constraint, according to the soft real-time paradigm. Multiple instances of the same application and different applications are allowed to be executed simultaneously, partially or completely overlapping. However, we assume that only one application is allowed to be executed on a core at each instant of time.

We adopt a hierarchical approach in estimating the MTTF of the entire system. The MTTF of the node is estimated by using the framework described in Section 5.2.3; the MTTF of the entire system is estimated by using the first failure model, which offers a pessimistic estimation, since the system could still survive although with a lower QoS. The extension of the lifetime estimation framework to support two-level architectures will be considered as a future work.

Within a node, for each PE, the energy consumption depends on the specific core characteristics and on the voltage levels related to the operating points it has been working at (particularly relevant when DVFS is enabled), in addition to the execution time. The energy consumption of a node is given by summing each PE's contribution, and the overall energy consumption is computed by adding the contributions for all the nodes.

State of Art

There are a few recent approaches aimed at resource management for either power consumption and/or lifetime optimization in symmetric [34, 21, 54, 37] or asymmetric [43, 29] architectures. The authors in [37] propose an off-line technique to improve the lifetime reliability of MPSoCs, while [54] describes a combination of design-time and runtime techniques to optimize it. It is commonly agreed upon that online adaptation is essential when dealing with aging, temperature, or energy related optimization, due to the lack of significant information that could drive a design-time solution space exploration. Thus, when moving to on-line optimization, [34] is one of the first approaches considering both lifetime reliability and energy consumption in MPSoCs; however, the two dimensions are optimized separately. Energy and reliability optimization is considered in [21] as well: the proposed hybrid approach does not consider computational energy optimization and DVFS-enabled architectures.

All the above-mentioned works deal with symmetric architectures. On the asymmetric side, in [29], the authors introduce dynamic reliability aware task scheduling without taking into account energy consumption optimization. Finally, among all the approaches proposed in literature,

[43] is the one that offers a solution similar to ours; however, it does not consider combined energy and lifetime reliability optimization and the reference architecture does not support DVFS, an important asset to improve energy consumption.

We here also briefly introduce the architecture of the framework presented in [89], on top of which our proposal is built. The original solution manages energy consumption at the node level, where the node has an ARM big.LITTLE architecture, integrating two high performing, complex, out-of-order ARM Cortex-A15 and three energy-efficient, simple, in-order ARM Cortex-A7 cores on the same chip.

Three are the relevant components for the discussion. At the node level there are a *Balancer* and a *Migrator*. The former guarantees that the cores within the clusters are evenly balanced with respect to the load, by mapping incoming tasks based on the resources past utilization. The latter is in charge of migrating the applications from one cluster to the other one, when the present mapping is not the best choice in terms of energy/performance trade-off. More precisely, it moves applications that do not achieve their target QoS at the maximum frequency on the A7 cluster to the A15 cluster. Dually, it moves applications from the A15 cluster to the A7 one when the measured QoS is above the maximum target QoS at the minimum frequency in the A15 cluster.

Finally, a per-cluster *DVFS Controller* is employed. This controller manipulates the voltage-frequency levels to meet the target performance goals of all the applications in the cluster while minimizing the energy consumption. This component is based on a PID controller using a control-theoretic approach. Since the cores working points can be manipulated only at the cluster level, the target QoS of the PID controller is determined by the application with the highest computational demand. By meeting the performance target at the lowest possible voltage-frequency level, the energy consumption is minimized.

Orchestrator Design

The decision taken by the components of the base framework are completely aging unaware. We aim at introducing an orchestrator for supporting the aging-aware resource management and to improve the lifetime of the system with a limited impact from the computation energy point of view. The approach we propose is a two-level one: single node and multi-node scenario. While the first one is an extension of the existing architecture for incorporating the combined lifetime-energy optimization, the latter is a completely innovative solution.

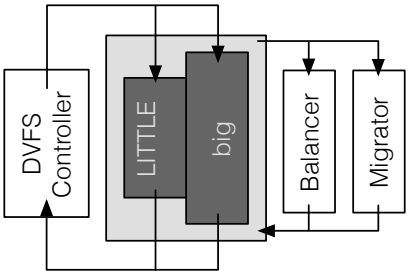


Figure 5.12: Block diagram for the reference energy optimization framework.

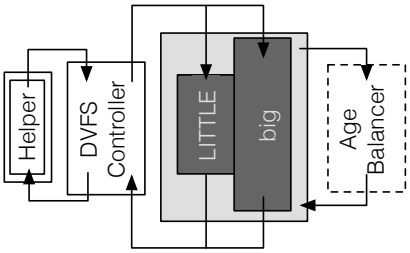


Figure 5.13: Block diagram for the proposed lifetime and energy optimization framework.

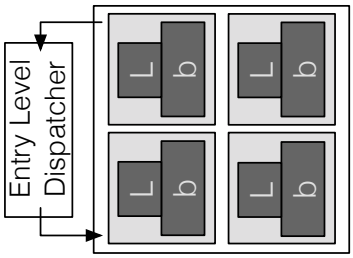


Figure 5.14: Extension of the proposed framework to a mult-node architecture.

Algorithm 3 Age Balancer algorithm

```
1: mappings = {0}
2: clusters = getClusters()
3: for all c in clusters do
4:    PE = getCores(clusters[c])
5:    apps = getApplications(clusters[c])
6:    for all a in apps do
7:       bestCluster = getBestCluster(apps[a])
8:       if bestCluster! = clusters[c]
         and isFree(bestCluster) then
9:          bestCluster.addApp(apps[a])
10:         apps.remove(apps[a])
11:      end if
12:   end for
13:   sortedPE = coreAgeSort(PE)
14:   sortedA = appsInverseAgeSort(apps)
15:   for all i in sortedA do
16:      mappings[clusters[c]].add(sortedA[i], sortedPE[i])
17:   end for
18: end for
19: return mappings
```

For the single node scenario, as shown in Figure 5.13, the Balancer and the Migrator have been modified and merged into a single component, the *Age Balancer*. Its aim is to re-map running applications, by selecting the best cluster for exploiting the energy/performance trade-off and the best PE for balancing the cluster aging. A new component, the *Helper Controller*, has been introduced to support the DVFS controller decisions. This component explicitly aims at reducing wear-out phenomena related to thermal cycling due to aggressive voltage and frequency scaling.

Age Balancer. The first mapping of each application is computed according to its worst-case QoS; the less energy-hungry resource able to satisfy the worst-case performance requirement is selected. However, the average QoS of an application is typically much lower than the worst-case QoS, which can be gathered by looking at the applications,Äô execution traces. Thus, the initial mapping usually represents a over-provisioned design decision; for example, during phases of low computational requirements, the application could be migrated to a different cluster type and still meet the performance requirements with a much lower energy consumption. The *Age Balancer* merges the Balancer and the Migrator

tasks into a single component, in charge of periodically adjusting the applications mapping, considering both their current QoS (with reference to the desired one) and the aging of the PEs.

Algorithm 3 shows the pseudo-code of the *Age Balancer* heuristics. The algorithm iterates through all the clusters in the node and within each cluster, through all the applications, to find a different cluster where to move the application so that the performance requirement is satisfied and energy consumption is minimized. At the same time, within the cluster, the application that contributed the most to the aging in the previous interval is mapped onto the PE aged the least. By doing this, the algorithm achieves a balanced aging across all the PEs.

Helper Controller. The aim of this controller is to assist the DVFS controller in manipulating the voltage-frequency levels of the cores; one controller for each cluster is needed, since the DVFS is available per-cluster. As mentioned, the PID controller is in charge of guaranteeing the performance target to be met at the lowest voltage-frequency level. However, the dynamic phases of an application can make the PID controller switch aggressively between these levels. Although aggressive scaling improves energy consumption, it also has significant negative impact on the lifetime reliability through thermal cycling. To avoid this phenomenon, we propose a *Helper Controller* that prevents the drastic modification in voltage-frequency level as follows. By reading the power sensors, this controller can measure the actual power consumption of the cluster. It estimates the power consumption at different voltage-frequency levels by using equation $P = A \times C \times V^2 \times F$, where A is the activity factor, C is the load capacitance, V is the voltage and F is the frequency. The target power consumption can be estimated given the target voltage and frequency. Using the estimated power consumption, the helper controller employs the widely used thermal RC model [87] to estimate the temperature at various voltage-frequency levels. From the current temperature measurement and Equation 5.4, the helper controller can prune the set of voltage-frequency levels that can potentially affect the lifetime reliability of the system by a certain threshold. In our work, we assume the threshold to be 10% of the actual MTTF of the system. More precisely, to prevent TC, when the PID controller is transitioning from higher to lower frequency, the DVFS controller selects the highest values between the PID controller and the minimum of the frequencies computed by helper controller. When the transition is from lower to higher frequency, the DVFS controller selects the lowest value between the PID controller and the maximum frequency computed by the helper controller.

The orchestrator is in charge of coordinating the different components

and invoking them with a proper frequency. The per-cluster DVFS controller is invoked at a higher frequency compared to the Age balancer. There are three major reasons behind the aforementioned choice. First, the age balancer focuses primarily on improving the lifetime reliability of the system. It has been a well-established phenomenon that the lifetime reliability of a microprocessor is significantly related to the temperature. As the temperature changes occur slowly [87], the age balancer can be invoked at a much lower frequency. Second, the overhead of invoking the age balancer is high if compared to the one of DVFS controller. In ARM big.LITTLE, the overhead of migrating application across clusters is in the order of milliseconds (from 2 to 4 ms) [89]. Therefore, the age balancer has to be invoked very infrequently. Lastly, the overhead of changing voltage-frequency levels is quite minimal [89]. Changing voltage-frequency levels can positive impact the energy consumption of the system; therefore, the DVFS controller is invoked at a higher frequency.

Experimental Evaluation

The experimental sessions here described aim at demonstrating how the proposed framework improves the results obtained by the base framework in terms of architecture lifetime without heavily impacting on energy consumption and achieves almost the same lifetime improvements as a dedicated lifetime optimization framework, also improving energy consumption. More precisely, we take as reference for our comparison two single objective frameworks optimizing energy (i.e, *Energy-only* [89]) or lifetime (a heuristic for MTTF maximization inspired by [20], *MTTF-only*).

The experimental platform is a four-node architecture, where each node is an ARM big.LITTLE. A single node is considered for the first experimental session. We collected applications power and performance traces by running applications on a real ARM big.LITTLE core using a Versatile Express development platform [3], built at 45nm GP technology. The test chip consists of two core Cortex-A15 (big) cluster and three core Cortex-A7 (LITTLE) cluster. Since no per-core thermal sensors are available on-board, we developed a simulation environment, based on SystemC/TLM, integrating the models of collected real data characterizing both the applications and the board. The simulator is fundamental in estimating the architecture thermal profile: it implements a steady-state temperature model, validated by means of HotSpot [87], that takes into account the self-activity of each core, the operating frequencies and

the neighbor cores' temperature. The obtained temperatures are processed and used to compute the cores' reliability according to Equations 5.1–5.4. We referred to [54] for choosing the values of the parameters related to EM, and to [76] for the ones related to TC.

Real-life applications selected from the one listed as case studies in the introductory chapter have been used to run the experiments. These applications have been selected to represent a good mix of heterogeneous behaviors. All the applications have been executed on both the A15 and on the A7 core for each of the available working frequencies; from $500MHz$ to $1.2GHz$, with a $100MHz$ step, for A15; from $600MHz$ to $1.0GHz$, with a $100MHz$ step, for A7. Most of the applications executing under $600MHz$ on A7 experienced very poor performance, therefore we discarded the related traces. For each <app, freq> pair a trace file is computed and stored, containing the application's power consumption and its performance at regular time intervals. The power sensors in ARM big.LITTLE have been used to measure power consumption. The QoS of the applications are expressed in terms of heartrate; we inserted the heartbeat register points as mentioned in [89].

Ideally, each single application has to be profiled individually to estimate the speedup for various voltage-frequency levels, resulting in complex off-line analyses. To avoid extensive profiling, we assume a linear relationship between frequency and heartrate for all the applications. The invocation frequency of age balancer is $200ms$; the DVFS controller one is $50ms$, which is much higher than the Linux scheduling epoch ($10ms$). Both frequencies have been empirically selected to obtain the best accuracy/overhead trade-off. The overhead of changing voltage-frequency levels is assumed to be $50\mu s$; application re-mapping within the same cluster takes $200\mu s$, $4ms$ across clusters [89].

For the first execution scenario, we defined five different use cases (UC_{sx}) consisting of 50 applications, randomly selected among the applications previously presented. For each application: i) the arrival time is computed according to a Poisson distribution (described by its parameter λ) and ii) the expected heartrate is computed as the average heartrate of a randomly selected trace among the available ones. The λ value is constant and chosen such that the architecture is fully loaded with no waiting queues.

Figure 5.15 shows the results for this first campaign. In the top graph, we report the ratio between the computed MTTF and the MTTF achieved by the lifetime-only optimization approach (dashed horizontal line, [20]). As expected, both the energy-only and the propose approach are not as effective, however on average they reach 80.6% and 92.3% of

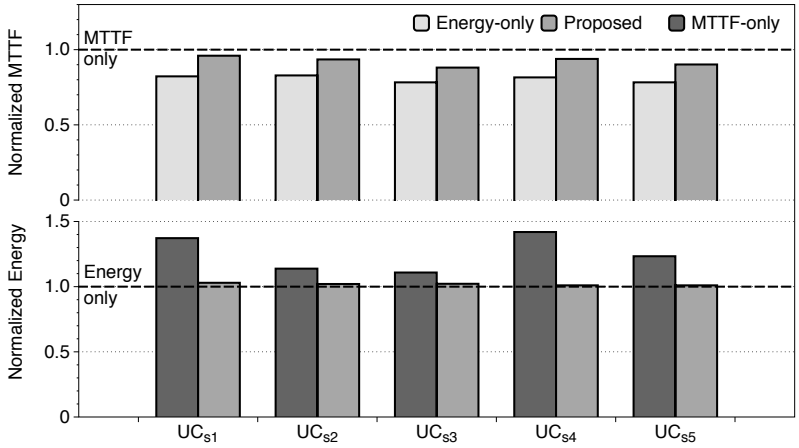


Figure 5.15: The proposed approach compared with the baseline framework in a *single node* architecture.

the optimal lifetime, respectively. The bottom part of the graph reports the corresponding energy ratio with respect to the energy consumption of the energy-only optimization approach (dashed horizontal line, [89]). In this case, the energy overhead for *Proposed* is, on average, 1.8% higher, while the lifetime optimization framework deteriorates the best results by 25.4%.

5.3.2 Multi Node and Computation Energy

In the multi-node scenario the architecture is considered to be made of multiple nodes, where each node has a ARM big.LITTLE configuration. As far as computation energy is concerned, the communication infrastructure is not relevant. However, since we will also consider communication energy, we adopt a NoC infrastructure.

Orchestrator Design

When considering a multiple node architecture, a new component for managing the overall architecture is introduced. The aim of this entity, dubbed *Entry Level Dispatcher* (as shown in Figure 5.14), is to select the best initial mapping for the incoming applications; it must take nodes aging into consideration and, if possible, select the best energy-aware solution as well. The dispatcher takes decisions based on the:

- application profiles, consisting of its worst-case required QoS and its average power consumption, for each different core type;

- system status, i.e. the number of *active* and *free* clusters/nodes.

These profiles are obtained using off-line analysis. The worst-case required QoS and average power consumption are measured at the maximum voltage-frequency level for each core type. Assuming a linear relationship between QoS and voltage-frequency level, we can estimate the performance at different frequency levels; similarly, the power consumption can be estimated as well, as already explained. We assume that the asymmetric multicore is equipped with per-core sensors measuring power, energy, voltage, frequency and temperature. We also assume that each core is equipped with wear sensors similar to the ones exploited in [43]. The dispatcher can access the sensors information to determine the system status. The dispatcher schedules the incoming applications to the appropriate node and cluster, aiming at improving the lifetime reliability while minimizing the energy, with performance as a constraint. The flow chart is shown in Figure 5.16.

The arriving applications are enqueued in a FIFO queue. Using the application profile, the dispatcher can estimate the best cluster type and its voltage-frequency level that can satisfy the applications performance constraint at a minimal energy consumption. First, the dispatcher identifies already powered-on clusters of the same type having free PEs: these are the ideal candidates. For a lifetime reliability-aware mapping, the dispatcher selects the cluster that is least aged among the clusters with similar voltage-frequency levels. However, if the dispatcher does not find any powered-on cluster with free PEs, it powers up a free cluster similar to the type previously estimated. If no such cluster is available, the dispatcher selects another cluster type that can meet the performance at the lowest possible energy cost. Last, if no free clusters are available, it waits until a PE becomes free.

Experimental Evaluation

We have extended the policies of the reference framework to the multi-node scenario in a straightforward manner, to have a baseline reference. The same application scenario has been adopted, while the value of λ has been tuned again to have the architecture constantly busy, but to avoid waiting queues.

The results, plotted in Figure 5.17 for the 4-node architecture, show almost the same trend of the previous experimental session. The proposed technique proves to still be able to mimic the optimal results in terms of lifetime, achieving, on average, 90.6% of the optimal lifetime. On the other hand, the lifetime obtained by the aging unaware solution drops to 47.8%. Results for energy consumption are more similar to

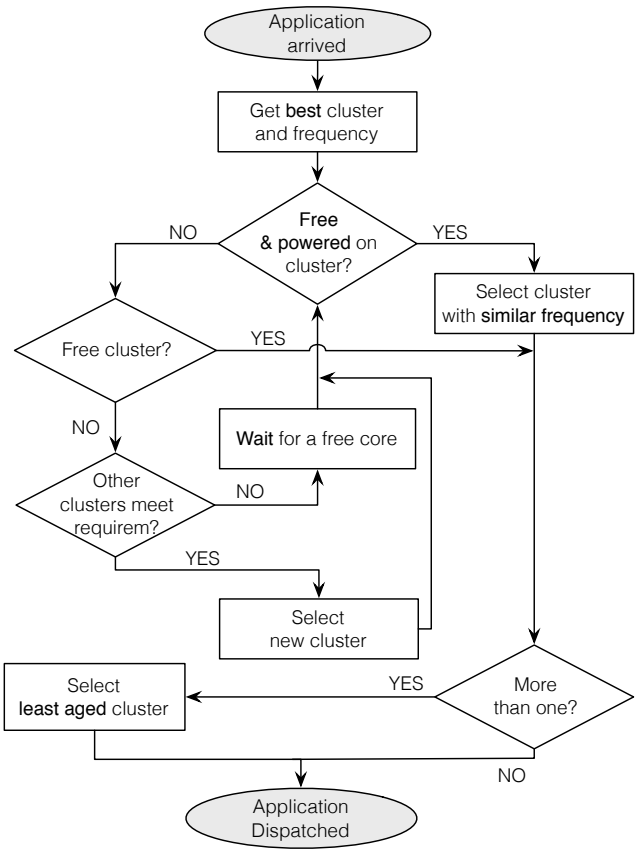


Figure 5.16: Control flow chart for the dispatching algorithm.

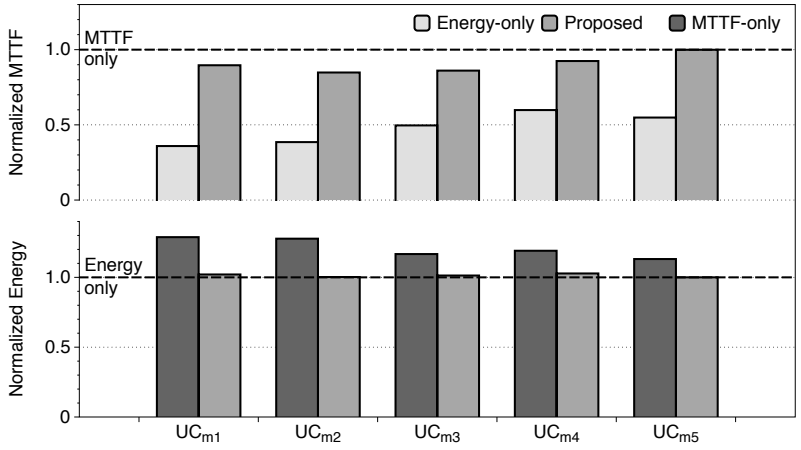


Figure 5.17: The proposed approach compared with the baseline framework in a *multi-node* architecture.

the previous scenario. The *Proposed* solution maintains an average 1.3% overhead with respect to the best energy consumption values, while the overhead of the energy unaware solution is 21.1%.

As expected, the frameworks optimizing a single objective (energy or lifetime) excel with respect to the metric they adopt, being characterized by high impact on the other aspect; the energy-optimization framework also has some limitations in terms of scalability. On the other hand, the combined solution we propose achieves results that are more similar to the optimal ones, for both lifetime and energy consumption; it shows a negligible overhead in energy consumption and a significant lifetime extension. More precisely, we can also note that although lifetime is related to temperature as well as energy-optimization policies, it is necessary to consider lifetime reliability as a first-class citizen to actually obtain optimal results with respect to such aspects. Furthermore, experimental results show that the combined optimization introduces only marginal overheads on the independent metrics, thus only reducing the potential optimal condition of a single goal optimization. Altogether, the introduction of a second important optimization objective has an overhead that is very limited with respect to the benefits it introduces when compared to the single objective solution.

It is also worth noting how, when moving from the single to the multi-node scenario, the combined approach shows good scalability: energy overhead from 1.8% to 1.3% and lifetime from 92.3% to 90.6% of the optimal value. Conversely, the expected lifetime of the base framework heav-

ily drops when considering multi node architectures, leading to greater relative improvements when compared with the other approaches.

5.3.3 Multi Node and Communication Energy

Most recent multi-node systems consist of processing nodes interconnected via networks-on-chip (NoCs) in a mesh-based architecture. Data processing applications mapped onto these platforms are typically characterized by large data exchange among the tasks. Therefore, task allocation is pivotal in determining energy consumption associated with communication among dependent tasks of the application. Data communication agnostic mapping of these applications can lead to a significant energy consumption on the NoC communication infrastructure, contributing to as much as $\approx 40\%$ of the overall application energy consumption [47, 12].

We here propose the design of an orchestrator and a self-adative system to dynamically adapts a set of pre-computed, design-time decisions, based on run-time application dynamism. The objective of the mixed design-time/runtime approach is to mitigate aging in a manycore system and minimize application communication energy while satisfying throughput requirements to provide the desired quality-of-service to end users. The solutions here presented have been published in [20].

State of the Art

The problem of scheduling dependent tasks with precedence constraints on a finite set of processing elements, with the aim of maximizing or minimizing an objective function, is NP-complete. Energy/reliability-aware task mapping and scheduling fall within this set of problems; it can be performed at design-time [63] or at run-time [27]. Design-time approaches can devote much more time in finding the best solution, since the computation is performed statically and off-line once in the entire system lifetime [53, 54, 91, 35, 36]. In [53] and [54], a simulated annealing-based technique is proposed to address the lifetime reliability-aware task mapping problem with the objective of maximizing system lifetime measured as mean time to failure, MTTF. Another reliability-driven task mapping approach is proposed in [91]: a cluster-based allocation technique to cope with fault-tolerant issues by smartly allocating the extra tasks needed for this purpose. In [35], two static energy-aware heuristics for task mapping are presented to optimize performance with respect to energy consumption. Finally, a convex optimization-based mapping generation technique to maximize system MTTF is proposed in [36]. Many

mapping solutions for several use cases are computed at design-time and stored; at run-time, the best pre-computed solution is applied according to the current fault-scenario. This approach assumes to know all the possible run-time scenarios and requires substantial amount of memory to store a mapping database; moreover, it does not consider energy consumption.

The main limitation of all these static design-time policies is the fact that they assume a-priori knowledge of the workload (e.g., task-graph composition, execution times, and applications' schedules) and of the system (e.g., faulty nodes, and architecture aging trend). These assumptions are admissible when considering a static application scenario, but they do not hold in scenarios where the applications number and their characteristics are unknown in advance and can change in an unpredictable manner. The aim of the approach here presented is to simultaneously optimize the three dimensions: components aging, reliability and communication energy consumption while adapting the system to runtime dynamism.

Dynamic approaches are more suitable to adapt task mapping at unknown runtime scenarios. An interesting work is the one presented in [43]: simple mapping configurations are statically computed and then enhanced through run-time heuristics, allowing to track actual components aging. In [29], a run-time task mapping technique is proposed to explicitly optimize system lifetime; application mapping is computed at run-time together with the frequency at which the re-mapping algorithm needs to be invoked. While these approaches share some common points with the one this paper introduces, they do not take into account energy consumption which is critical for modern embedded systems. The work that is more closely related to the one here proposed is [31]; although data-communication is optimized together with reliability, it does not guarantee maximization of the system lifetime.

Background

In this context, applications are modelled as fork-join task-graphs and are supposed to be executed in a mutually exclusive fashion on the reference architecture. This means that all applications are known at design-time, but not the workload in terms of the order of execution and the starting times, thus leading to a highly-variable workload scenario and the need for a run-time mapping policy.

Each processing node has a specific position in the NoC architecture described by its (x, y) integer coordinates; a node can communicate with any other node in the architecture by means of messages sent through

the NoC routing elements. The location of the nodes is fundamental to properly model the communication among them, in terms of message latency and energy consumption, as explained in Section 5.3.3. The *fabric controller*is in charge of dispatching the applications to the other processing nodes, by computing the best mapping and scheduling configuration, according to specific ad-hoc designed metrics. This special node is connected to the NoC infrastructure through a special link, it is hardened by design so to achieve fault tolerant properties, and it is assumed not to influence the thermal profile of the system.

Computation energy consumption is considered almost constant, regardless of the mapping, when considering homogeneous architectures [48]. For this reason it is neglected in the following analysis. Communication energy consumption, on the other side, is mapping-dependent and represents $\approx 60\%$ of the overall application energy consumption. In [102], the authors defined bit energy (E_{bit}) as the energy consumed when one bit of data is communicated through the routers and links of a NoC.

$$E_{bit} = E_{S_{bit}} + E_{L_{bit}} \tag{5.14}$$

where $E_{S_{bit}}$ and $E_{L_{bit}}$ are the energy consumed by the switch and the link, respectively. The energy per bit consumed in transferring data between task v_i and v_j mapped on processor p and processor q, respectively, and positioned $n_{hops}(p,q)$ away is given by Equation 5.15 according to [48].

$$E_{bit}(p,q) = \begin{cases} n_{hops}(p,q)E_{S_{bit}} + (n_{hops}(p,q)-1)E_{L_{bit}} & \text{if } p \neq q \\ 0 & \text{otherwise} \end{cases} \tag{5.15}$$

where $n_{hops}(p,q)$ is the number of routers between processors p and q. The total communication energy is thus given by

$$CE = \sum_{(i,j)\in E} d_{ij} \times E_{bit}(p,q). \tag{5.16}$$

Orchestrator Design

The problem that the proposed self-adaptive system aims at solving can now be better formalized. Given i) a set of n applications $A = \{a_1, a_2, \ldots, a_n\}$, modeled as directed graphs $G = (V, E)$ and executed in a mutually exclusive fashion, ii) a computing architecture composed of m processing elements, each one described by its coordinates (x, y) and connected through a NoC infrastructure to the others, and iii) a user-defined throughput constraint, which may vary during the execution, the

given applications must be mapped on the available processing nodes so as to meet the given constraint. Moreover, the mapping policies must be chosen so as to minimize the energy consumption and to maximize the nodes lifetime, by balancing the components aging to extend the system lifetime.

The designed solution combines a static approach together with a dynamic one to solve the mapping problem in the presence of conflicting optimization goals (as performance, reliability and energy consumption) while retaining the best from both worlds.

An overall schema of the proposed approach is presented in Figure 5.18: it exploits design-time strategies to compute good mapping solutions to start from; these solutions are then dynamically optimized. In this proposal, we considered a state-of-the-art static strategy [36] aiming at optimizing reliability under a performance constraint; this approach is able to exploit all the needed information (the applications task-graphs, the architecture topology, the performance constraint and reliability/performance metrics – grouped by a dashed line in Figure 5.18) available at design-time and to perform an accurate design space exploration devoted to the identification of the best solution. The output of this first step is a database of solutions (*Mapping Database*), which contains the best mapping for each input application. Note that, although the static mapping of [36] is used as the starting mapping in the proposed dynamic approach, the strategy is orthogonal with respect to the static mapping generation and can be used in conjunction with other existing static techniques (e.g., [53]).

The dynamic approach selects the best mapping for an application from the database and uses it at run-time to optimize energy and reliability (measured as mean time to failure, MTTF) while fulfilling the throughput constraint.

In particular, the dynamic part is based on an orchestrator running on the architecture's fabric controller, able to monitor the system behavior and status (both architectural parameters and application ones), and consequently take decisions and act to modify the working conditions and parameters with the aim to improve the pursued goals.

In the *observe* phase the throughput is computed according to the start and end time of each application iteration and the energy consumption is estimated according to the current mapping. However, the orchestrator cannot compute the actual MTTF because it would require to be able to predict changes in the architecture. Indeed, a good parameter representing the current aging status of each of the core is $\alpha(T)$ (refer

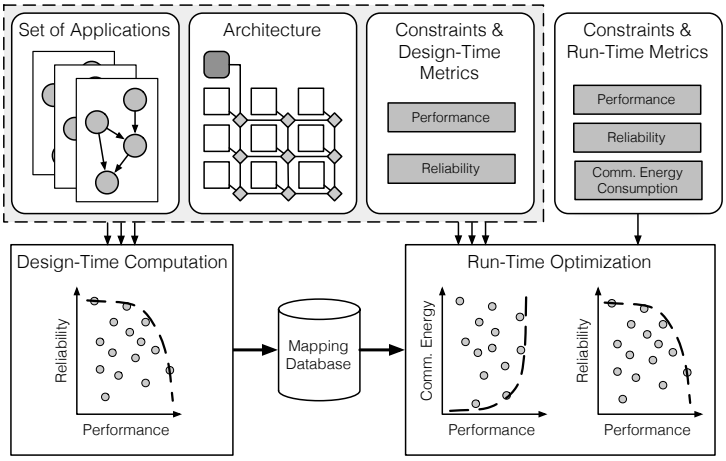


Figure 5.18: Overview of the proposed methodology.

to Equation 5.3), and it can be computed by monitoring temperature variations in time by means of hardware sensors.

The *decide* phase is entered when a specified activation condition on the monitored metrics is triggered. It is possible to set an activation condition when a core is aging faster than the others (how much faster is defined by the designer) or to make the orchestrator adapt periodically. This phase is devoted to the identification of the most convenient task, or set of tasks, to be remapped and the selection of the node where to move them, in order to improve the current values of the optimization metrics while fulfilling the given performance constraint on the throughput. We considered only the relocation of a single task per move, however, multiple relocations are under investigation to speed up the adaptation process.

The algorithm for the definition of the remapping moves is shown in Algorithm 4. The basic idea is to improve the architecture lifetime by periodically unloading the eldest core and distributing a part of the tasks mapped on it to other units, while trying at the same time to limit the energy consumption. Thus, the orchestrator selects the eldest node C_o and a set of the k youngest ones $\mathbb{C}_{\mathbb{N}} = \{C_{n_1}, ..., C_{n_k}\}$ (Lines 1-6). Then, it defines all the possible moves as a single relocation of a task t_j from C_o to C_{n_i} (Lines 7-11), and sorts them according to the following priority order: considering the age of the node C_{n_k} (youngest first, since it is the most unstressed one) and, in case of same value, considering task t_j duration (largest first, since it is the one mainly contributing to core aging – Line 12). Moreover, to pursue energy saving, in the sorting

process, the age of the moves causing an increase in the communication energy will be weighted by the energy variation $\Delta CE_{t_j,C_{n_i}}$. It is worth noting that the energy variation contribution can be turned-off while optimizing reliability only.

The orchestrator evaluates each move's costs/benefits ratio by applying it for one cycle to analyse the achievable make-span and, consequently, whether the throughput constraint is met or not. Indeed, differently from the energy variation that is estimated according to a defined model, the current make-span cannot be measured off-line, since it depends on the actual execution of the application on the architecture; in fact, an estimation of such a value would require the scheduling to be known in advance, and this computation would be too time-consuming to be performed on-the-fly on the fabric controller. Therefore, in the *act* phase the engine will attempt a move per cycle in the given order; if the move is accepted, the engine will sleep until the activation condition will be triggered another time, otherwise it will try with the subsequent move in the list (Lines 13-21).

Algorithm 4 Task remapping strategy

1: $T_C \leftarrow$ *specified throughput constraint*
2: $C_o \leftarrow$ *eldest node in the architecture*
3: $\mathbb{C}_n \leftarrow k$ *youngest nodes in the architecture*
4: $\mathbb{T}_o \leftarrow$ *set of tasks mapped on* C_o
5: $CE_{ref} \leftarrow$ *Energy consumption of the current mapping*
6: $\mathbb{M} \leftarrow \oslash$ – *Set of candidate moves*
7: **for each** $t_j \in \mathbb{T}_o \wedge C_{n_i} \in \mathbb{C}_n$ **do**
8: *define move* $t_j : C_o \rightarrow C_{n_i}$
9: $\Delta CE_{t_j,C_{n_i}} \leftarrow CE_{ref} - CE_{move}$
10: $\mathbb{M} \leftarrow \mathbb{M} \cup \{t_j : C_o \rightarrow C_{n_i}\}$
11: **end for**
12: *Sort* $\mathbb{M}$ *according to priority function* $f(\alpha_{C_{n_i}}, \tau_j, \Delta CE_{t_j,C_{n_i}})$
13: *apply first* $t_j : C_o \rightarrow C_{n_i} \in \mathbb{M}$
14: *run the application per 1 cycle*
15: $T \leftarrow$ *current throughput*
16: **while** $T < T_C$ **do**
17: *undo previous move*
18: *apply next* $t_j : C_o \rightarrow C_{n_i} \in \mathbb{M}$
19: *run the application per 1 cycle*
20: $T \leftarrow$ *current throughput*
21: **end while**
22: **return**

Experimental Evaluation

We compare the effectiveness of the proposed dynamic approach with the *Static MaxMTTF* one, proposed in [36]. A functional simulator has been implemented using SystemC/TLM [1] to model the dynamic engine together with the described NoC-based manycore architecture running applications modeled as task-graphs. A set of six real-life applications taken from the case studies is considered, namely FFT, MPEG Decoder, MWD, Picture-in-Picture (PiP), VOPD, and Romberg Integration, from [11]; two different architectural platforms are selected, composed of a 3×3 and 3×4 mesh NoC, respectively. The bit energy (Ebit) for modeling communication energy of an application is calculated using expressions provided in [102] for packet-based NoC using 65nm technology parameters from [103]. The following parameters are used for computing aging [36]: current density $J = 1.5 \times 10^6 A/cm^2$, activation energy $E_a = 0.48eV$, slope parameter $\beta = 2$, temperature $T = 295K$ and $n = 1.1$. HotSpot [87] has been used to characterize the temperature of the cores to account both the self-activity and the temperature of neighbour cores. The considered performance requirement is computed by taking the best performance possible on the given architecture (computed at design-time through a design space exploration) and by adding to it an extra 20%. This value has been chosen to be consistent with the application scenario and with the state of the art [36].

The introduction of an orchestrator providing self-adaptiveness may cause both a performance and energy communication overhead. The former one is due to the actual time needed by the orchestrator to compute the next move; in the considered system, it is completely hidden since such computation is performed during the applications' execution. However, changing the mapping of the architecture implies communicating information to the cores, thus introducing delays and increasing the used communication energy. Indeed, although this aspect has been partially neglected, we expect the impact to be limited, if not negligible, with respect to the overall performance/energy consumption, being the amount of data very small.

Figure 5.19 plots the normalized aging of the proposed dynamic technique with respect to the maximum MTTF obtained using the static approach for the adopted experimental setup. In this experiment, energy optimization was disabled, and the proposed dynamic orchestrator migrates tasks on nodes to optimize MTTF. The reported MTTF values are normalized with respect to the MTTF of an unstressed architecture.

A few considerations can be drawn. First of all, for all applications con-

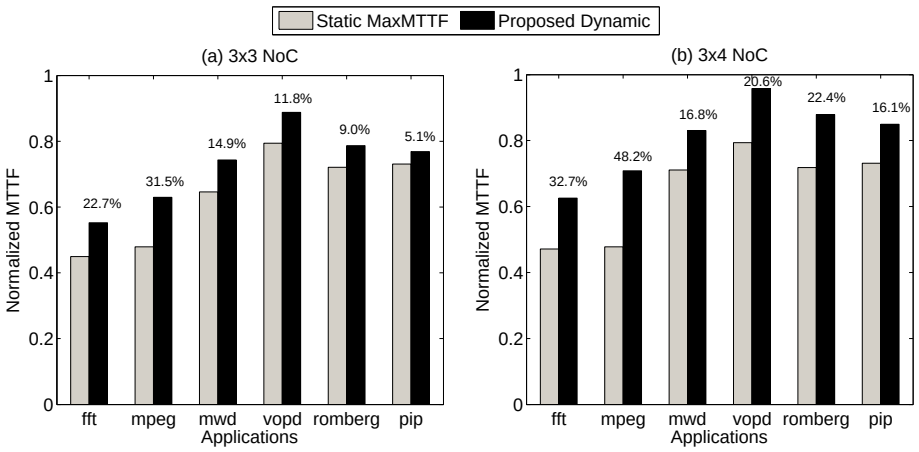


Figure 5.19: MTTF performance of the proposed approach.

sidered (including those not shown explicitly in the figure), the MTTF obtained using the proposed dynamic approach is better than the one achieved in the *Static MaxMTTF* scenario, as indicated on the bars for the proposed technique. This is due to the adaptation of the architecture to balance the stress of different units thereby improving the overall MTTF. For the six considered applications, the proposed technique improves MTTF by 16% on average. Moreover, the MTTF improvement increases as the number of cores in the architecture grows. This is because the adaptation engine is able to better balance the architecture load/aging by switching between the unused units.

Table 5.2 reports a comparative analysis of the communication energy between the static and the proposed dynamic technique (with energy optimization turned off) on the two different architectures. Energy consumption increases by an average of 9% and of 15%, on the two architectures, due to the adaptation engine that uses extra energy for task migration.

Figure 5.20 plots the result for three multi-application and three multi-throughput test cases on the 3×3 architecture. The multi-application and multi-throughput scenarios are generated by randomly selecting the applications to increase the workload, in a not homogeneous way. The number of iterations for each application in the test cases are specified in Table 5.3. As an example, one iteration of MultiApp01 consists of 2.000 iterations of VOPD, 10.000 of Romberg, and 4.000 of FFT; this workload is repeated infinitely. The iteration of multi-throughput appli-

Applications	Static MaxMTTF	Proposed Dynamic	
		3×3	3×4
FFT	1	1	1.162704082
MPEG	1	1.017674265	1
MWD	1	1.040284091	1
VOPD	1	1.138318841	1.286558603
Romberg	1	1	1.052919355
PiP	1	1.346435185	1.430726496
Average	1	1.090452064	1.155484756

Table 5.2: Energy performance of the proposed dynamic approach with MTTF optimization only

MultiApp01	MultiApp02	MultiApp03
VOPD (2,000)	MPEG (5,000)	MPEG (10,000)
Romberg (10,000)	MPEG (1,000)	PiP (5,000)
FFT (4,000)	Romberg (4,000)	VOPD (4000)
	MPEG (1,000)	PiP (8,000)

MultiThr01	MultiThr02	MultiThr03
$MPEG_1$ (10,000)	$VOPD_1$ (10,000)	PiP_1 (10,000)
$MPEG_2$ (10,000)	$VOPD_2$ (10,000)	PiP_2 (10,000)

Table 5.3: Parameters for multi-application and multi-throughput

cation MultiThr01 consists of 10.000 iterations of MPEG with the same deadline used in the static scenario, and 10.000 iterations with deadline relaxed by 2×.

The static approach generates application mappings considering an unused architecture, i.e., age of all the nodes are 0. As a result, when applications are switched at run-time, the static pre-computed mappings are no-longer optimal in terms of MTTF because they ignore the current age of the different cores. Such information, on the other hand, is exploited by the dynamic approach that can adapt accordingly and produce MTTF optimized solutions. For all the three multi-applications considered, the proposed technique improves the architecture MTTF by 27% on average.

Applications are often characterized by varying throughput require-

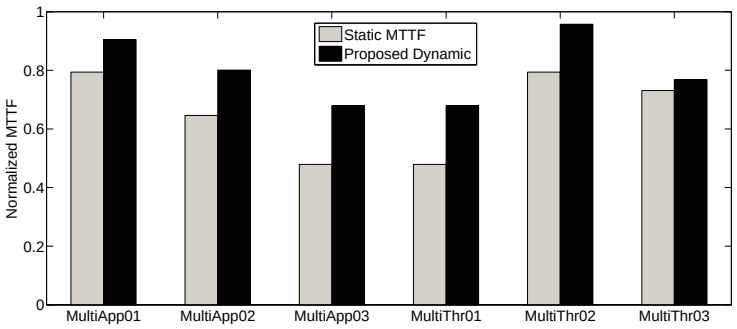


Figure 5.20: MTTF performance with multi-application and multi-throughput scenarios.

ments. Computing the application mapping at design-time for all applications and for all throughput requirements results in an explosion in the design space. With 20 applications having on average two different throughput requirements, the number of application mappings to be pre-computed at design-time is 2^{20}. This imposes a significant overhead in terms of storage for the mapping, and of time to retrieve the information at runtime. An alternative approach adopted for most design-time approaches is to generate one mapping for each of these applications by using the stricter throughput requirement. Therefore, such approach gives good results when the throughput requirement is strict, but is not able to optimize the MTTF when the requirement is relaxed. Indeed, the dynamic approach is able to adapt to this changing scenario. Although, the initial mapping for the dynamic approach is the one pre-computed at design-time with a stricter deadline, the adaptive engine is able to explore different other mappings fulfilling the relaxed throughput requirement, possibly achieving better results. Experimental results for the three multi-throughput applications indicate that the proposed technique improves system MTTF by 22%, on average, when compared to the one using static mapping without adaptation.

Figure 5.21 plots the normalized communication energy of the proposed technique compared to the static approach, when optimizing both communication energy and MTTF. Similarly to the previous experiments, the initial mapping for this experiment is the best MTTF mapping coming from the static approach. However, the orchestrator incorporates communication energy and system lifetime when selecting a local move. For a more comprehensive comparison, we report results with and without the communication energy optimization, to highlight how

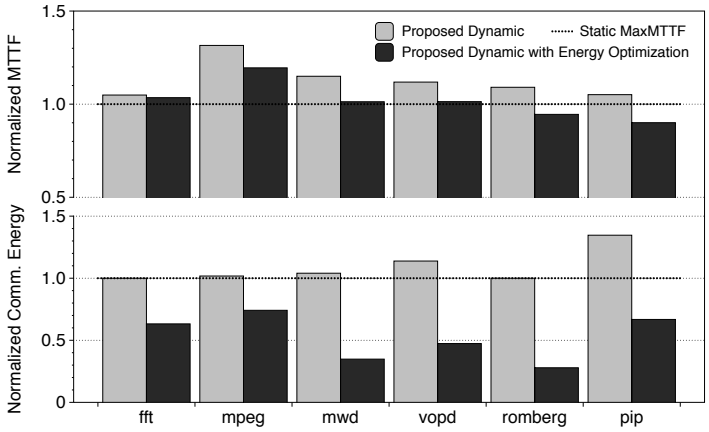


Figure 5.21: Communication energy performance for the proposed technique.

MTTF and energy consumption are affected. All values are normalized with respect to the data obtained using the static approach. As it can be seen, MTTF-communication energy joint optimization results in energy savings between 25% to 75% for all the applications considered with an average 5% MTTF improvement. Thus by trading-off MTTF, the communication energy can be minimized by 50%, if considering the average of the results obtained on the considered benchmark applications.

5.3.4 Putting it all together

The last step in optimizing the lifetime/energy consumption trade-off is to consider communication and computation energy together in a multi-node architecture.

Each node has a ARM big.LITTLE architecture and the nodes are connected through a NoC communication infrastructure (both these aspects are relevant in this case). Applications are modeled as fork-join task-graphs and not as serial applications, since data exchange among tasks is fundamental for evaluating communication energy consumption. Performance constraints are considered both in terms of worst case heartrate and a deadline not to be missed. The communication energy model is the same as the ones presented in Section 5.3.3, while computation energy consumption is modeled as described in Section 5.3.2. In particular, the communication energy consumption will be considered for the NoC only, neglecting internal contributions, since bus-based architectures usually

consume way less energy than NoC-based ones, since they consist of fewer elements and do not have links and routers to be powered-up [97].

Orchestrator Design

At the multi-node level, the orchestrator behavior is inspired by the hybrid technique described in Section 5.3.3, where good mapping solutions computed at design-time are optimized at runtime. The same state-of-art static strategy [36] has been exploited for computing such design time solutions. It performs an accurate design space exploration by considering the information available at design-time (the applications task-graphs, the architecture topology, and the performance constraint). The fact that computation energy is not considered during the design space exploration does not represents a limitation, since at this level all the nodes are homogeneous and, at time 0, they share the same initial status. Computation energy optimization is considered in the runtime phase of the architecture, where the mappings retrieved from the database are optimized. Algorithm 4 has been modified in order to keep into consideration computation energy consumption as well when sorting the selected moves (Line 9); the score of each move is computed as in Algorithm 4 by considering an aging contribution and an energy one, where the latter one considers both communication and computation energy, in particular:

$$\Delta E_{t_j,C_{n_i}} = \alpha \cdot \Delta CompE_{t_j,C_{n_i}} + \beta \cdot \Delta CommE_{t_j,C_{n_i}} = \tag{5.17}$$

$$= \alpha \cdot (CompE_{ref} - CompE_{move}) + \beta \cdot (CommE_{ref} - CommE_{move}) \tag{5.18}$$

where α and β are weighting coefficients and $CommE_{ref}$, $CommE_{move}$ have the same meaning as CE_{ref} and CE_{move} in Algorithm 4, respectively, while $CompE_{ref}$ refers to the computation energy consumption of the current solution and $CompE_{move}$ to the one of the move under test. In this way, moves resulting in an increase of energy consumption will receive a negative score, placing them at the end of the list, while moves with the highest energy saving will be at the top. The orchestrator can find mappings characterized by a low computation energy consumption at runtime by exploiting, for example, already powered on clusters or by grouping on the same node tasks requiring the same performance level. As in the inspiring approach, the orchestrator needs to evaluate the make-span due to the new mapping by actually running the system for one cycle. In case the make-span for the current move is lower than the deadline, the move is accepted and the orchestrator will remain inactive until the activation condition will be triggered another time. If

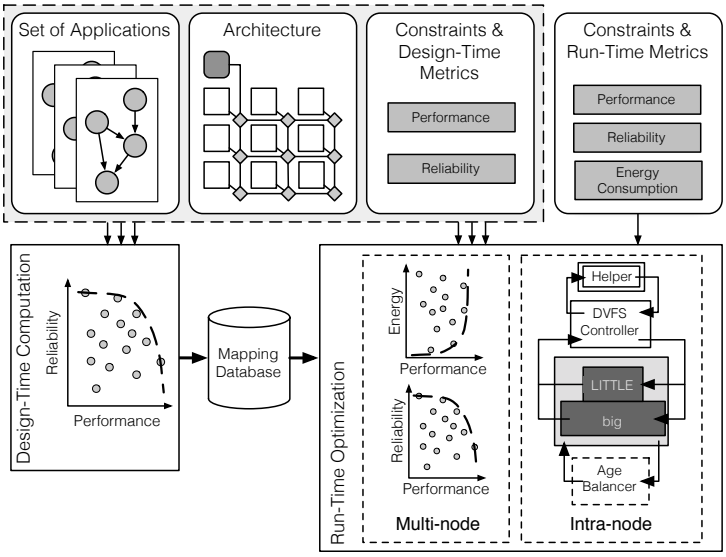


Figure 5.22: Overview of the proposed approach when both communication and computation energy consumptions are optimized.

the make-span exceeds the deadline, the current move is discarded and the next move in the sorted list is selected for evaluation, and so on.

It is worth noting that the coefficients α and β can be tuned to properly weight the contributions of computation energy and communication energy in the moves score. α should be grater than β in computing-intensive scenarios; in case of high data exchange applications the relationship between α and β will be the other way around. In particular, when $\alpha = 0$ communication energy only is optimized at the multi-node level and the approach leads back to the one described in Algorithm 4. On the other hand, when $\beta = 0$ computation energy only is considered and an approach similar to the one proposed in Section 5.3.2 is obtained.

At the single node level, since communication energy is neglected, the solution proposed in Section 5.3.1 has been exploited. Thus, an Age Balancer and a DVFS Controller (together with its Helper) are implemented, per node, to manage the energy computation/lifetime reliability trade-off. The proposed approach is summarized in Figure 5.22, where the runtime optimization box has been updated with reference to Figure 5.18 and contains both a multi-node and an intra-node optimization policy.

5.4 Final Remarks

This chapter presented a self-adaptive infrastructure for extending multi-/manycore architectures lifetime. First, an enabling tool was designed to make the lifetime estimation fast and precise. Then this tool was exploited to evaluate the effectiveness of the implemented orchestrator, aiming at maximizing lifetime, while reducing energy consumption and meeting performance constraints. To achieve this goal, the orchestrator has to properly coordinate different components acting on the applications mapping and scheduling, on the resources status and on their working voltage-frequency points. The problem is faced gradually, considering first a single node and computation energy only, then moving to a more complex architecture and to communication energy optimization as well.

In the next chapter, the idea of designing a system exploiting both the self-adaptive systems presented in this chapter and in the previous one will introduced and discussed. Moreover, some concluding remarks for the whole work described in this thesis will be drawn, together with possible future developments.

6 Conclusions & Future Works

The research presented in this thesis has proposed a novel self-adaptive, *reliable* framework for in multi/manycore architectures. The work is motivated by the necessity of dynamically exploiting reliability/energy consumption/performance trade-offs when considering highly evolving working scenarios. Runtime resource management solutions are to be designed to properly adapt the system's behavior to the changing environment; this is the reason why classical design-time (static) approaches do not suffice in providing satisfying result and the adoption of a new protocol (the Observe-Decide-Act control loop) was needed.

The objective of this research is to design a system with self-adaptive capabilities, able to make smart decisions at runtime and get the most out of the considered trade-offs. *Reliability* represents the main optimization dimension. *Energy consumption* minimization has been introduced because it is directly and considerably affected by the knobs the framework was exploiting. *Perfomance* has been considered as a constraint to be satisfied according to the soft real-time paradigm. The result is a cross-layer self-adaptive system for the combined optimization of reliability (treated both as fault management and components lifetime) and energy consumption (both communication and computation one) under performance constraints. The proposed framework autonomously takes care of the resource management problem, hiding its complexity and taking its burden away from the programmer. The overall work is organized in several layers as shown in Figure 6.1 and summarized in the following.

At the multi-node level, the designed framework employs a hybrid approach to minimize aging while optimizing communication and computation energy consumption (Layer A in Figure 6.1). A runtime orchestrator has been designed to smartly map tasks on the available nodes starting from pre-computed optimal mappings. Tasks are then re-mapped, at runtime by means of heuristics, to cope with the evolving conditions (Section 5.3.4). Transient fault management is considered at the intra-node level only, since creating, scheduling, and gathering results of redundant threads and voters/checkers benefit from a shared-memory bus-based architecture such as the intra-node one (Layer B in Figure 6.1). A rule-based system has been designed to guide the orchestrator in selecting, at each instant of time, the best redundancy-based reliable technique to

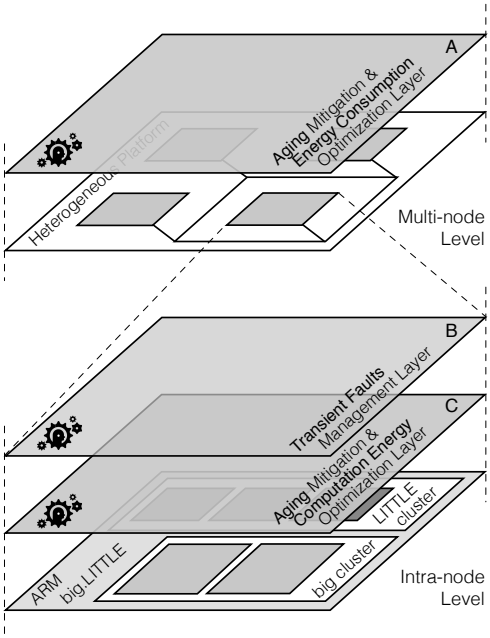


Figure 6.1: A graphical representation of the research contribution of this thesis, organized in *self-adaptive layers*.

satisfy the user's reliability requirements and minimize the perfomance overhead (Chapter 4). This layer is located, in each node, on top of another adaptation layer that takes care of aging mitigation and computation energy optimization (Layer C in Figure 6.1). This is achieved by acting on different knobs (tasks mapping and scheduling, resource switch-on/off, DVFS) through the synergic orchestration on ad-hoc designed controllers (Section 5.3.1). Each adaptive layer has been validated in a simulation environment by executing application traces collected from execution on real architectures. The obtained results proved the effectiveness of the proposed approach, obtaining remarkable improvements in terms of lifetime extension and energy consumption reduction, while meeting performance constraints.

The envisioned framework and the design of the presented adaptation layers represent the main innovative contribution of this research work. The preliminary investigation on self-adaptive systems led to the formalization of a model for describing and organizing this kind of systems in a structured and systematic way, as well as for preliminary validating them. Moreover, the need for estimating complex architectures lifetime motivated the development of a lightweight framework for MTTF esti-

mation, based on Monte Carlo simulations and random walks. These topics, discussed in Chapter 3 and Section 5.2.3, respectively, represent further contributions of this thesis.

Although many aspect have been covered in this research work, some others are yet to be explored. Thus, there are several directions for future work; the more relevant ones are listed below.

Extend the proposed framework to consider process variability. With technology scaling, core-to-core parameters variations pose a major challenge to high-performance microprocessor design, negatively impacting cores frequency and leakage power [82], thus motivating the need for variation-aware optimization algorithms to be integrated in the proposed framework. The literature on self-adaptive variability-aware approaches should be surveyed (i.e., [38, 13]) and the possibility to integrate variability management within the proposed framework investigated.

Further integration of the system components. A possible extension of the various designed components is their integration in a unique resource manager. This would allow to overcome an important disadvantage of the proposed layered adaptive system: an application cannot be mapped in more than one node due to the node-granularity solution of layer B for transient faults. A mechanism to manage and coordinate architecture across nodes is to be design, in order not to limit the application parallelism. Moreover, the MTTF evaluation tool presented in Section 5.2.3 could be used at runtime to make the decision making process even more aware, and not only for the off-line estimation. Last, the model for self-adaptive systems presented in Chapter 3 could be exploited in the system implementation phase, for the development of a framework able to generate, in a template-fashion, a stub of the orchestrator according to the driving dimensions value.

Exploration of new reliable techniques. The reliable scheduling techniques considered in Chapter 4 are all based on the idea of indiscriminately replicating all the tasks in a task-graph, even if with different granularities or a varying number of times. An interesting development is to consider scheduling techniques based on *selective replication*; according to this approach, only a given percentage of tasks in the elaboration phase is actually replicated. This idea can be applied only if the required fault tolerance coverage is not 100%, but it allows to further exploit the reliability/performance trade-off.

Improvement of the decision making process. The decision making process was carried out mainly through heuristics and rule-based algorithms, which fail in providing any kind of guarantee about their convergence and the obtained system's stability; the only exception is represented by the DVFS controller that exploits a control theory-based approach. More advanced control theoretical approaches [72, 71], machine learning algorithms [70] and even price theory-based techniques [88] proved their effectiveness in dealing with the resource management problem and the possibility to integrate them in the proposed framework should be investigated.

Bibliography

[1] Accelera Systems Initiative. `http://www.accellera.org`.

[2] N. Aggarwal, P. Ranganathan, N. P Jouppi, and J. E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. In *International Symposium on Computer Architecture*, pages 470–481, 2007.

[3] ARM Ltd. `http://www.arm.com/products/tools/development-boards/versatile-express/index.php`, 2011.

[4] M. Auslander, D. Dasilva, D. Edelsohn, O. Krieger, M. Ostrowski, B. Rosenburg, R.W. Wisniewski, and J. Xenidis. K42 overview. Technical report, IBM T. J. Watson Research Center, August 2002.

[5] P. Bailis, V. J. Reddi, S. Gandhi, D. Brooks, and M. Seltzer. Dimetrodon: Processor-level preventive thermal management via idle cycle injection. In *Design Automation Conference*, pages 89–94, 2011.

[6] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *International Journal on Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007.

[7] A. Baumann, P. Barham, P.E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoey, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Syposium on Operating Systems Principles*, pages 29–44, 2009.

[8] R.C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Transactions on Device and Materials Reliability*, 5(3):305–316, Sept 2005.

[9] T. Becker, A. Agne, P. R. Lewis, R. Bahsoon, F. Faniyi, L. Esterle, A. Keller, A. Chandra, A. R. Jensenius, and S. C. Stilkerich. EPiCS: Engineering Proprioception in Computing Systems. In *International Conference on Computational Science and Engineering*, pages 353–360, 2012.

[10] L. Benini and G. De Micheli. Networks on chips: A new soc paradigm. *IEEE Computer*, 35(1):70–78, 2002.

[11] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli. Noc synthesis flow for customized domain specific multiprocessor systems-on-chip. *Transactions on Parallel and Distributed Systems*, 16(2):113–129, 2005.

[12] L. Bin, P. Li-Shiuan, and P. Patra. Impact of process and temperature variations on network-on-chip design exploration. In *International Symposium on Networks-on-Chip*, pages 117–126, April 2008.

[13] G. Bizot, F. Chaix, N.E. Zergainoh, and M. Nicolaidis. Variability-aware and fault-tolerant self-adaptive applications for many-core chips. In *International On-Line Testing Symposium*, pages 37–42, July 2013.

[14] G. Blake, R.G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine*, 26(6):26–37, November 2009.

[15] J. Blome, F. Shuguang, S. Gupta, and S. Mahlke. Self-calibrating online wearout detection. In *International Symposium on Microarchitecture*, pages 109–122, Dec 2007.

[16] C. Bolchini and M. Carminati. Multi-core emulation for dependable and adaptive systems prototyping. In *Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale*, pages 1–4, March 2014.

[17] C. Bolchini, M. Carminati, M. Gribaudo, and A. Miele. A lightweight and open-source framework for the lifetime estimation of multicore systems. In *International Conference on Computer Design*, pages 1–7, October 2014.

[18] C. Bolchini, M. Carminati, and A. Miele. Towards the design of tunable dependable systems. In *Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale*, pages 17–21, June 2012.

[19] C. Bolchini, M. Carminati, and A. Miele. Self-adaptive fault tolerance in multi-/many-core systems. *Journal of Electronic Testing*, 29(2):159–175, 2013.

[20] C. Bolchini, M. Carminati, A. Miele, A. Das, A. Kumar, and B. Veeravalli. Run-time mapping for reliable many-cores based on energy/performance trade-offs. In *International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, pages 58–64, 2013.

[21] C. Bolchini, M. Carminati, A. Miele, and E. Quintarelli. A framework to model self-adaptive computing systems. In *Conference on Adaptive Hardware and Systems*, pages 71–78, 2013.

[22] C. Bolchini, M. Carminati, T. Mitra, and T. Somu Muthukaruppan. Combined on-line lifetime-energy optimization for asymmetric multicores. In *Technical Report*, 2014.

[23] C. Bolchini, C. Curino, E. Quintarelli, F. A. Schreiber, and L. Tanca. Context information for knowledge reshaping. *International Journal of Web Engineering and Technology*, 5(1):88–103, 2009.

[24] C. Bolchini, A. Miele, and C. Sandionigi. A novel design methodology for implementing reliability-aware systems on SRAM-based FPGAs. *Transactions on Computers*, 60(12):1744–1758, 2011.

[25] C. Bolchini, A. Miele, and D. Sciuto. An adaptive approach for online fault management in many-core architectures. In *Conference on Design, Automation Test in Europe*, pages 1429–1432, March 2012.

[26] S. Borkar, T. Karnik, and V. De. Design and reliability challenges in nanometer technologies. In *Design Automation Conference*, pages 75–75, 2004.

[27] E. Carvalho, N. Calazans, and F. Moraes. Heuristics for dynamic task mapping in NoC-based heterogeneous MPSoCs. In *International Workshop on Rapid System Prototyping*, pages 34–40, 2007.

[28] T. Chantem, R. P. Dick, and X. S. Hu. Temperature-aware scheduling and assignment for hard real-time applicationson MPSoCs. In *Conference on Design, Automation and Test in Europe*, pages 288–293, 2008.

[29] T. Chantem, Y. Xiang, X.S. Hu, and R.P. Dick. Enhancing multicore reliability through wear compensation in online assignment and scheduling. In *Conference on Design, Automation and Test in Europe*, pages 1–6, 2013.

[30] Z. Chen, M. Yang, G. Francia, and J. Dongarra. Self adaptive application level fault tolerance for parallel and distributed computing. In *International Symposium on Parallel and Distributed Processing*, pages 1–8, 2007.

[31] C.L. Chou and R. Marculescu. FARM: Fault-aware resource management in NoC-based multiprocessor platforms. In *Conference on Design, Automation Test in Europe*, pages 1–6, 2011.

[32] S. Corbetta, D. Zoni, and W. Fornaciari. A temperature and reliability oriented simulation framework for multi-core architectures. In *International Symposium on VLSI*, pages 51–56, 2012.

[33] A.K. Coskun, T. Simunic, K. Mihic, G. De Micheli, and Y. Leblebici. Analysis and optimization of MPSoC reliability. *Journal of Low Power Electronics*, pages 56–69, 2006.

[34] A.K. Coskun, R. Strong, D.M. Tullsen, and T. Simunic Rosing. Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors. In *International Conference Measurement and Modeling Computer Systems*, pages 169–180, 2009.

[35] A. Das, A. Kumar, and B. Veeravalli. Communication and migration energy aware task mapping for reliable multiprocessor systems. *Future Generation Computer Systems*, 2013.

[36] A. Das, A. Kumar, and B. Veeravalli. Reliability-driven task mapping for lifetime extension of networks-on-chip based multiprocessor systems. In *Conference on Design, Automation and Test in Europe*, pages 689–694, 2013.

[37] A. Das, A. Kumar, and B. Veeravalli. Temperature aware energy-reliability trade-offs for mapping of throughput-constrained applications on multimedia MPSoCs. In *Conference on Design, Automation and Test in Europe*, pages 1–6, 2014.

[38] S. Dighe, S.R. Vangal, P. Aseron, S. Kumar, T. Jacob, K.A. Bowman, J. Howard, J. Tschanz, V. Erraguntla, N. Borkar, V.K. De, and S. Borkar. Within-die variation-aware dynamic-voltage-frequency-scaling with optimal core allocation and thread hopping for the 80-core teraflops processor. *Journal of Solid-State Circuits*, 46(1):184–193, Jan 2011.

[39] E. Fernandez-Alonso, D. Castells-Rufas, J. Joven, and J. Carrabina. Survey of NoC and programming models proposals for MP-SoC. *International Journal of Computer Science Issues*, 9(2):22–32, 2012.

[40] European Cooperation for Space Standardization. Methods for the calculation of radiation received and its effects, and a policy for design margins. Technical Report ECSS-E-ST-10-12C, November 2008.

[41] 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 *Conference on Design, Automation and Test in Europe*, pages 533–538, 2011.

[42] Z. Gu, C. Zhu, L. Shang, and R. P. Dick. Application-specific MPSoC reliability optimization. *Transactions on VLSI Systems*, 16(5):603–608, 2008.

[43] A.S. Hartman and D.E. Thomas. Lifetime improvement through runtime wear-based task mapping. In *International Conference on Hardware/software codesign and system synthesis*, pages 13–22, 2012.

[44] K. Henricksen and J. Indulska. A software engineering framework for context-aware pervasive computing. In *Conference on Pervasive Computing and Communications*, pages 77–86, 2004.

[45] H. Hoffmann, J. Eastep, M.D. Santambrogio, J.E. Miller, and A. Agarwal. Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *International Conference on Autonomic Computing*, pages 79–88, 2010.

[46] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology, 2001.

[47] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. S. Borkar. A 5-ghz mesh interconnect for a teraflops processor. *Micro, IEEE*, 27(5):51–61, Sept 2007.

[48] J. Hu and R. Marculescu. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In *Conference on Design, Automation and Test in Europe*, pages 234–239, 2004.

[49] J. Huang, J.O. Blech, A. Raabe, C. Buckl, and A. Knoll. Analysis and optimization of fault-tolerant task scheduling on multiprocessor embedded systems. In *International Conference on Hardware/software codesign and system synthesis*, pages 247–256, 2011.

[50] L. Huang and Q. Xu. AgeSim: A simulation framework for evaluating the lifetime reliability of processor-based SoCs. In *Conference on Design, Automation and Test in Europe*, pages 51–56, 2010.

[51] L. Huang and Q. Xu. Lifetime reliability for load-sharing redundant systems with arbitrary failure distributions. *Transactions on Reliability*, 59(2):319–330, 2010.

[52] L. Huang, R. Ye, and Q. Xu. Customer-aware task allocation and scheduling for multi-mode MPSoCs. In *Design Automation Conference*, pages 387–392, 2011.

[53] L. Huang, F. Yuan, and Q. Xu. Lifetime reliability-aware task allocation and scheduling for MPSoC platforms. In *Conference on Design, Automation and Test in Europe*, pages 51–56, 2009.

[54] L. Huang, F. Yuan, and Q. Xu. On task allocation and scheduling for lifetime extension of platform-based MPSoC designs. *Transactions on Parallel and Distributed Systems*, 22(12):2088–2099, 2011.

[55] International Technology Roadmap for Semiconductors – Emerging Research Devices Section. `http://public.itrs.net/`, 2010.

[56] A. Jantsch and H. Tenhunen. Will networks on chip close the productivity gap? In *Networks on Chip*, pages 3–18. Springer US, 2003.

[57] JEDEC Solid State Technology Association and others. Failure mechanisms and models for semiconductor devices. *JEDEC Publication JEP122G*, 2010.

[58] E. Karl, D. Blaauw, D. Sylvester, and T. Mudge. Multi-mechanism reliability modeling and management in dynamic systems. *Transactions on VLSI Systems*, 16(4):476–487, 2008.

[59] K.M. Kavi, R. Giorgi, and J. Arul. Scheduled dataflow: execution paradigm, architecture, and performance evaluation. *Transactions on Computers*, 50(8):834–846, 2001.

[60] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36:41–50, 2003.

[61] I. Koren and C.M. Krishna. *Fault-Tolerant Systems*. Elsevier Science, 2010.

[62] A. Kouadri, O. Heron, and R. Montagne. A lightweight API for an adaptive software fault tolerance using POSIX-thread replication. In *International Conference on Architecture of Computing Systems*, pages 16–19, 2011.

[63] Y.K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *Computing Surveys*, 31(4):406–471, 1999.

[64] C. Kyriacou, P. Evripidou, and P. Trancoso. Data-driven multithreading using conventional microprocessors. *Transactions on Parallel and Distributed Systems*, 17(10):1176–1188, 2006.

[65] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Conference on Dependable Systems and Networks*, pages 317–326, 2007.

[66] J.C. Laprie. Dependable computing: Concepts, limits, challenges. In *International Conference on Fault-tolerant Computing*, pages 42–54, 1995.

[67] M. Lattuada, C. Pilato, A. Tumeo, and F. Ferrandi. Performance modeling of parallel applications on MPSoCs. In *International Conference on System-on-Chip*, pages 64–67, 2009.

[68] Linaro Ubuntu release for Vexpress. `http://releases.linaro.org`, 2014.

[69] H. Liu. Reliability of a load-sharing k-out-of-n:G system: non-iid components with arbitrary distributions. *Transactions on Reliability*, 47(3):279–284, 1998.

[70] M. Maggio, H. Hoffmann, A.V. Papadopoulos, J. Panerati, M.D. Santambrogio, A. Agarwal, and A. Leva. Comparison of decision-making strategies for self-optimization in autonomic computing systems. *Transactions on Autonomous and Adaptive Systems*, 7(4):36:1–36:32, 2012.

[71] M. Maggio, H. Hoffmann, M.D. Santambrogio, A. Agarwal, and A. Leva. Power optimization in embedded systems via feedback control of resource allocation. *Transactions on Control Systems Technology*, 21(1):239–246, Jan 2013.

[72] M. Maggio, F. Terraneo, and A. Leva. Task scheduling: A control-theoretical viewpoint for a general and flexible solution. *Transactions on Embedded Computing Systems*, 13(4):76:1–76:22, 2014.

[73] 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 *Conference on Digital System Design*, pages 517–524, 2012.

[74] P. Mercati, A. Bartolini, F. Paterna, T.S. Rosing, and L. Benini. Workload and user experience-aware dynamic reliability management in multicore processors. In *Design Automation Conference*, pages 2:1–2:6, 2013.

[75] S.S. Mukherjee, M. Kontz, and S.K. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *International Symposium on Computer Architecture*, pages 99–110, 2002.

[76] H.V. Nguyen. *Multilevel Interconnect Reliability: On the Effects of Electro-Thermomechanical Stresses*. PhD thesis, University of Twente, Twente, Netherland, 2004.

[77] E. Normand. Single event upset at ground level. *Transactions on Nuclear Science*, 43(6):2742–2750, 1996.

[78] NVidia Corporation. The benefits of multiple cpu cores in mobile devices., 2011. `http://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf`.

[79] J. Panerati, M. Maggio, M. Carminati, F. Sironi, M. Triverio, and M.D. Santambrogio. Coordination of independent loops in self-adaptive systems. *Transactions on Reconfigurable Technology and Systems*, 7(2):12:1–12:16, 2014.

[80] Politecnico di Milano. ReSP web site: `http://code.google.com/p/resp-sim/`, 2011.

[81] E.T. Salehi, M. Asadi, and S. Eryilmaz. Reliability analysis of consecutive k-out-of-n systems with non-identical components lifetimes. *Journal of Statistical Planning and Inference*, 141(8):2920–2932, 2011.

[82] S.R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. Varius: A model of process variation and resulting timing errors for microarchitects. *Transactions on Semiconductor Manufacturing*, 21(1):3–13, Feb 2008.

[83] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Workshop on Mobile Computing Systems and Applications*, pages 85–90, 1994.

[84] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Chita. METE: meeting end-to-end QoS in multicores through system-wide resource management. In *International Conference on Measurement and modeling of computer systems*, pages 13–24, 2011.

[85] A. Shye, B. Scholbrock, and Gokhan G. Memik. Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures. In *International Symposium on Microarchitecture*, pages 168–178, 2009.

[86] F. Sironi, D. B. Bartolini, S. Campanoni, F. Cancare, H. Hoffmann, D. Sciuto, and M. D. Santambrogio. Metronome: operating system level performance management via self-adaptive computing. In *Design Automation Conference*, pages 856–865, 2012.

[87] K. Skadron, M.R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *Transactions on Architecture Code Optimization*, 1(1):94–125, 2004.

[88] T. Somu Muthukaruppan, A. Pathania, and T. Mitra. Price theory based power management for heterogeneous multi-cores. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 161–176, 2014.

[89] T. Somu Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin. Hierarchical power management for asymmetric multi-core in dark silicon era. In *Design Automation Conference*, pages 174:1–174:9, 2013.

[90] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The case for lifetime reliability-aware microprocessors. In *International Symposium on Computer Architecture*, pages 276–287, 2004.

[91] S. Srinivasan and N.K. Jha. Safety and reliability driven task allocation in distributed systems. *Transactions on Parallel and Distributed Systems*, 10(3):238–251, 1999.

[92] K. Stavrou, D. Pavlou, M. Nikolaides, P. Petrides, P. Evripidou, P. Trancoso, Z. Popovic, and R. Giorgi. Programming abstractions and toolchain for dataflow multithreading architectures. In *International Symposium on Parallel and Distributed Computing*, pages 107–114, 2009.

[93] STMicroelectronics and CEA. Platform 2012: A many-core programmable accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology. In *Research Workshop on STMicroelectronics Platform 2012*, 2010.

[94] Teraflux. Definition of ISA extensions, custom devices and External COTSon API extensions. In *Teraflux: Exploiting dataflow parallelism in Tera-device Computing*, 2011.

[95] The OpenMP API specification for parallel programming. `http://openmp.org/wp/`, 2011.

[96] L. Thiele, L. Schor, Y. Hoeseok, and I. Bacivarov. Thermal-aware system analysis and software synthesis for embedded multiprocessors. In *Design Automation Conference*, pages 268–273, 2011.

[97] A.N. Udipi, N. Muralimanohar, and R. Balasubramonian. Towards scalable, energy-efficient, bus-based on-chip networks. In *International Symposium on High Performance Computer Architecture*, pages 1–12, Jan 2010.

[98] Various Authors. The MIT Angstrom Project: Universal Technologies for Exascale Computing - `http://projects.csail.mit.edu/angstrom/index.html`, 2011.

[99] P. M. Wells, K. Chakraborty, and G. S. Sohi. Mixed-mode multicore reliability. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 169–180, 2009.

[100] M. Wirthlin, E. Johnson, N. Rollins, M. Caffrey, and P. Graham. The reliability of FPGA circuit designs in the presence of radiation induced configuration upsets. In *Symposium on Field-Programmable Custom Computing Machines*, pages 133–142, 2003.

[101] Y. Xiang, T. Chantem, R.P. Dick, X.S. Hu, and L. Shang. System-level reliability modeling for MPSoCs. In *International Conference on Hardware/Software Codesign and System Synthesis*, pages 297–306, 2010.

[102] T.T. Ye, L. Benini, and G. De Micheli. Packetized on-chip interconnect communication analysis for MPSoC. In *Conference on Design, Automation and Test in Europe*, pages 344–349, 2003.

[103] W. Zhao and Y. Cao. Predictive technology model for nano-CMOS design exploration. *Journal on Emerging Technologies in Computing Systems*, 3(1):1–17, 2007.