TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Echtzeitsysteme und Robotik

Towards an Integrated Framework for Reliability-Aware Embedded System Design on Multiprocessor System-on-Chips

Jia Huang

Vollständiger Abdruck der von der Fakultät der Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Hans Michael Gerndt
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Alois Knoll
	2. Univ.-Prof. Dr. Petru Eles, Linköping University/Schweden

Die Dissertation wurde am 10.12.2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 21.07.2014 angenommen.

Abstract

Today's integrated circuits are becoming more susceptible to faults due to effects caused by aggressive technology scaling. However, to meet the ever-increasing reliability requirements of safety-related applications, the system has to function correctly even in the presence of faults. This leads to the challenging problem of fault-tolerant system design. Here, the goal is to meet the required level of reliability with minimum overhead, while guaranteeing the system's timing, resource and other constraints.

This thesis presents an integrated framework tackling the problem above. The framework features modeling, analysis, optimization and code generation tools to support a complete reliability-aware design flow. Its modeling approach allows for abstract description of the application and the underlying execution platform. Using the models as input, the reliability analysis and multi-criteria optimization techniques enable automated exploration of design alternatives. Fault Tolerant Mechanisms (FTMs), including spatial/temporal redundancy, fault detection and voting are actively embedded into the design to meet the reliability goal. We extend the existing approaches by dropping inappropriate assumptions in the fault model and supporting a larger set of FTMs. Finally, our framework supports automatic generation of source code and platform configuration files from the abstract model, thereby accelerates the development process. Compared with existing work, our DSE techniques and the integrated tool framework advance the research results on reliability-aware design further into practice. The analysis and optimization techniques are evaluated with extensive experiments and the overall framework is applied on real-world case studies and demonstrators.

Zusammenfassung

Integrierten Schaltungen werden heutzutage aufgrund des fortwährenden Schrumpfens der Strukturgrößen immer anfälliger für Fehler. Um die ständig steigenden Anforderungen an die Zuverlässigkeit von sicherheitskritischen Anwendungen zu erfüllen, muss ein System jedoch auch Im Fehlerfall weiterhin funktionieren. Dies begründet die herausfordernde Aufgabenstellung des fehlertoleranten Systemdesigns. Hierbei ist es das Ziel, die erforderliche Zuverlässigkeit mit minimalem Overhead zu erreichen, und gleichzeitig die weiteren Randbedingungen an das System wie etwa Echtzeitanforderungen, die beschränkte Verfügbarkeit von Ressourcen sowie andere nicht-funktionale Anforderungen zu gewährleisteten.

Diese Dissertation stellt einen integrierten Ansatz zur Lösung des oben skizzierten Problems vor. Dieser besteht aus Werkzeugen zur Modellierung, Analyse, Optimierung und Codegenerierung, um so einen durchgängigen Entwicklungsprozess zu ermöglichen, der den Aspekt der Zuverlässigkeit berücksichtigt. Der Modellierungsansatz ermöglicht eine abstrakte Beschreibung der Anwendung sowie der zugrundeliegenden Ausführungsplattform. Dieses Systemmodell dient als Ausgangspunkt für unsere Zuverlässigkeitsanalyse und den multi-kriteriellen Optimierungsansatz, der eine automatische Exploration des Entwurfsraums ermöglicht. Hierzu wurden Fehlertoleranzmechanismen (z.B. räumliche bzw. zeitliche Redundanz, Fehlererkennung und Voter) in den Entwicklungsprozess integriert, um die Zuverlässigkeitsanforderungen zu erfüllen. Die vorliegende Arbeit erweitert dabei bestehende Ansätze durch ein realistischeres Fehlermodell und die Unterstützu einer größeren Anzahl von Fehlertoleranzmechanismen. Unser Ansatz unterstützt außerdem die automatische Generierung von Quellcode un Plattformkonfigurationsdateien aus dem abstrakten Systemmodell, was zu einer Beschleunigung des Entwicklungsprozesses führt. Unsere Techniken zu Entwurfsraumexploration und die Umsetzung des vorgeschlagenen Ansatzes in einer Werkzeugkette tragen somit zur praktischen Anwendbarkeit der Methoden zum Entwurf zuverlässiger Systeme bei.

Die Analysen und Optimierungstechniken wurden einerseits mit umfangreichen Experimenten ausgewertet. Und andererseits wurde die Anwendbarkeit des Ansatzes anhand von Fallstudien und Demonstratoren untersucht.

Acknowledgements

All the way along the great four years that I spend at the chair of robotic and embedded systems and the fortiss institute, I have been helped and supported by many outstanding people, without whom my work in the field and completion of this thesis would not have been possible.

My first and greatest gratitude must go to my advisor Prof. Alois Knoll. You have been advising me during all phases of the PhD work and also give me a lot of freedom in the research field. You encourage me to find the tackle the most interesting research topic. I benefit from the experience working with you for the life time.

I am especially grateful to my second advisor, Prof. Petru Eles. Your fundamental contribution in the field is the basis of my work. It is my luck and great honor to meet and work with you.

Many thanks to my supervisor at fortiss institute, Dr. Harald Ruess, Dr. Christian Buckl and Dr. Andreas Raabe. We have had so many fruitful discussions. You help me to target the right direction on my way to pursue the PhD and you give me the best support in tackling the technical challenges.

I would like to thank all my colleagues at the chair of robotic and embedded systems and the fortiss institute for their great collaboration. Especially, I thank Simon Barner, with whom I collaborated from the first day till the last day of my PhD journey. I benefit from your great knowledge and sense of responsibility.

Last but not least, I would like to thank my family: Mom, Dad, Grandparents and my wife Yan, who have been always loving and supporting me.

Contents

List of Figures xiii

1 Introduction 1

1.1 Reliability-Aware Embedded System Design 2

1.2 Tackling Reliability in Multiprocessor Systems 4

1.3 Motivation and Contribution 5

1.4 Overview 7

1.5 Organization 9

2 Preliminaries 11

2.1 Modeling Approach 11

2.1.1 Application Model 12

2.1.2 Platform Model 14

2.1.3 Mapping Application to the Platform 14

2.1.4 DSE Configuration Model 15

2.1.5 Implementation 16

2.2 Fault Model 17

2.3 Fault Tolerance Mechanisms 19

2.4 System Models 22

2.5 Scheduling Models 23

3 Related Work 27

3.1 Reliability-Aware Embedded System Design 27

3.1.1 Fault-tolerant system design focusing on permanent faults 29

3.1.2 Fault-tolerant system design considering transient faults 30

3.1.3 Qualitative Comparison of Related Work 33

3.2 Model-Driven Software Development . . . 34

4 Reliability Aware Design Space Exploration 39

4.1 Reliability Analysis . . . 39

4.1.1 Complexity and Approximation . . . 44

4.2 Static Priority Slack Sharing for Multiple Jobs . . . 46

4.3 Optimization Procedure . . . 49

4.3.1 Schedule Reconstruction . . . 50

4.3.2 Encoding of *TT-FS* Schedules . . . 51

4.3.3 Crossover and Mutation . . . 52

4.4 Tolerating Permanent Faults using Virtual Mapping . . . 53

4.5 Simulation Results . . . 58

4.5.1 Architecture Exploration Case Study . . . 59

4.5.2 Comparison of *TT-SP* and *TT-FS* . . . 61

4.5.3 The Case with Permanent Faults . . . 64

4.5.4 Comparing Slack Sharing Schemes . . . 65

5 Fault Tolerant System Design using Imperfect Fault Detectors 67

5.1 System Models . . . 67

5.2 Motivation . . . 69

5.3 Experimental Analysis on the Impact of Imperfect Fault Detection on System Reliability . . . 72

5.4 Reliability Analysis . . . 76

5.5 Optimization Procedure . . . 78

5.5.1 ILP Based Optimization for Single-Objective Case . . . 78

5.5.2 Multi-Objective Optimization . . . 80

5.6 Experiments . . . 82

5.7 Summary . . . 86

6 Automatic Code Generation and Platform Configuration 89

6.1 Template Based Code Generation . . . 89

6.1.1 Code Generation Strategy . . . 90

6.1.2 Code Generation for KPN . . . 93

6.1.2.1 Functional Code Generation . . . 94

6.1.2.2 Structural Code Generation . . . 96

6.2 Platform Configuration 99
6.2.1 Configuring a Time-Triggered Network-on-Chip 99
6.2.1.1 Related Work 100
6.2.1.2 Problem Definition and Transformation 101
6.2.1.3 Problem Transformation 102
6.2.1.4 SMT Specification 104
6.2.1.5 Heuristic Approach 107
6.2.1.6 Experiments 111
6.2.2 Integration of TTNoC Configurator 114
6.3 Summary 117

7 Case Study **119**
7.1 The Adaptive Cruise Control Case Study 119
7.1.1 Application and Platform Modeling 120
7.1.2 Design Space Exploration 122
7.1.3 Code Generation and Execution 125
7.2 The Industrial Control Demonstrator 127
7.3 Discussion of Usability 132

8 Conclusion and Outlook **135**
8.1 Main Results 135
8.2 Outlook 136

References **139**

List of Figures

1.1 Overview of the Design Flow 8
1.2 Comparison of traditional and MDD Flow 10

2.1 Example Model-driven Design Scenario 12
2.2 An Example of KPN Model 13
2.3 An Example Scenario of Mapping Application Model to Platform Model 15
2.4 Example of Temporal and Spatial Redundancy 20
2.5 Example of Combined Active Redundancy and Fault Detection 21
2.6 Example of *TT-SP* and *TT-FS* 24

4.1 Example Execution Scenarios for *TT-SP* 41
4.2 An Example of Binary Tree Analysis 42
4.3 Example of Static Priority Slack Sharing 47
4.4 Workflow of EA-based Optimization 50
4.5 Encoding and Reconstruction of Schedule 51
4.6 Example of Message Placement for *TT-FS* Schedules 52
4.7 Task Implementation Crossover Example 53
4.8 Motivating Example for Considering Transient and Permanent Faults Together . 54
4.9 Example of Virtual Mapping 56
4.10 Influence of Data Dependency on Task Migration 56
4.11 Evaluation of BTA with Approximation 59
4.12 Pareto Optimal Solutions under Different Platform Configurations 60
4.13 Achievable Reliability Comparison 61
4.14 Comparing *TT-SP* and *TT-FS* using *mpeg2* 63
4.15 An Example for Comparing *TT-SP* and *TT-FS* 63

4.16 Comparing *TT-SP* and *TT-FS* using Random TG 64
4.17 Performance Comparison of Step-wise and Unified Approaches 65
4.18 Example of Pareto Optimal Results 66
4.19 Comparison of Slack Sharing Schemes 66

5.1 Example Fault Scenario 69
5.2 Example Scenario 71
5.3 Reliability of the Example Schedules 72
5.4 Effect of Fault Detection with Fixed Replication: replication = 1 73
5.5 Effect of Fault Detection with Fixed Replication: replication = 2 73
5.6 Effect of Fault Detection with Fixed Replication: replication = 3 74
5.7 Effect of Replication with Fixed Fault Detection Coverage 75
5.8 Example of Reliability Function 79
5.9 Example of Encoding Scheme 81
5.10 The Resource Consumption Objective 82
5.11 2D Projection of Optimization Results 83
5.12 2D Projection of Optimization Results: FIT of DUF vs Resource Consumption . 85
5.13 2D Projection of Optimization Results: FIT of SDC vs Resource Consumption . 85
5.14 Comparing Results of three Architectures 87

6.1 Code Generation Flow 90
6.2 Example Code Template 91
6.3 An Example for Including User Code for Code Generation 96
6.4 An Example Scenario for Structural Code Generation 97
6.5 An Example for Structural Code Generation Procedure 98
6.6 A TTNoC Scheduling Scenario Example 100
6.7 The TTNoC Architecture 103
6.8 Message Scheduling to Bin Packing Transformation 104
6.9 Segmentation of Long Messages 105
6.10 PE-to-Switch Allocation Optimization 108
6.11 Example of Strip Packing and Level Packing 108
6.12 Example of Level Packing 110
6.13 Execution Time Comparison: 3x3 NoC 111
6.14 Execution Time Comparison: 5x5 NoC 112

6.15 Execution Time Comparison: 7x7 NoC 112
6.16 Error Rate of Heuristic Algorithms: 3x3 NoC 112
6.17 Error Rate of Heuristic Algorithms: 5x5 NoC 113
6.18 Error Rate of Heuristic Algorithms: 7x7 NoC 113
6.19 Feasibility Check Using TTNoC Configurator 117

7.1 Task Graph of ACC 120
7.2 The Appliation Model for ACC 121
7.3 The Platform Model for ACROSS Architecture 122
7.4 The DSE Configuration Model for ACC Case Study 123
7.5 DSE Result of the ACC Case Study 124
7.6 DSE Result of the ACC Case Study 125
7.7 Mechanical Setup of Industrial Control Demonstrator 128
7.8 Application Model of Sorting Application 129
7.9 Application Logic of Sorting Task 130
7.10 DMR implementation for the Industrial Control Demonstrator 131
7.11 Deployment of Industrial Control Demonstrator 131

Chapter 1

Introduction

Embedded systems are special-purpose computers responsible for dedicated functions in a large technical system. One important category of embedded systems is safety-critical real-time systems, which implement critical control applications in aerospace, automobile, industrial control, health care and other domains. A failure of such systems may cause catastrophic loss of property or human life. Hence, one of the most important design goals for safety-critical applications is to guarantee sufficient **reliability** of the system. Unfortunately, even if the system is designed correctly, the underlying hardware may subject to *faults*, leading to deviation of the system behavior from the specification. The hardware faults are incurred by certain physical effects such as wear-out and energetic particle strike that are infeasible to be avoided completely. Hence, to reach the desired level of reliability, the designer must guarantee, to a level of satisfaction, that the system functions correctly even in the presence of faults. Fault-Tolerant Mechanisms (FTMs) [1] are developed to achieve this goal.

As technology scales, the situation becomes even more challenging. On the one hand, modern integrated circuits are more susceptible to faults. As discussed in [2, 3], the probability of faults are expected to increase significantly for each technology generation in deep-submicron era, due to decreased feature size, higher power density and other effects caused by aggressive advances in technology scaling. On the other hand, reliability is becoming a first-order design criteria and a key marketing factor in today's safety-related applications, resulting in constantly higher reliability requirements [4, 5]. In this case, it is increasingly important to apply fault-tolerant techniques in the design of such systems.

We consider fault-tolerant embedded system design using commonly accepted FTMs, such as fault detection, active replication and voting. These FTMs can be applied stand-alone or

combined to meet the design objectives. In general, the design problem comprises several challenging tasks. First of all, the designer needs to analyze the reliability gain after applying certain FTMs and check if the reliability requirements are met. Second, since FTMs typically involve redundancy and come at a price, it is critical to find the optimal design alternative from the possibly huge design space. Finally, fault-tolerant techniques may negatively influence other system properties, e.g., timing, energy consumption and cost. It is also important to guarantee all other design requirements in the presence of FTMs.

This thesis deals with the aforementioned problem by presenting an integrated framework that supports a complete fault-tolerant system design flow. It features modeling, analysis, optimization and platform configuration tools to enable automatic synthesis of reliable designs adhering to user-specified requirements. We focus on supporting modern Multiprocessor System-on-Chip (MPSoC) platforms due to their increasing importance in the embedded domain [6, 7]. In this introductory chapter, we first review relevant background and explain motivation of our work. Afterwards, a short overview of the framework is given.

1.1 Reliability-Aware Embedded System Design

Reliability-aware system design deals with the problem of utilizing appropriate techniques to manage the hardware faults, so that their harm to the system is reduced to an acceptable level. The hardware faults can be categorized as permanent (hard errors), transient (soft errors), or intermittent faults [8]. Permanent faults cause non-recoverable device defects once they manifest and therefore reduce the system's lifetime. In contrast, transient faults do not fundamentally damage the device but may corrupt the application execution. They typically arise due to cosmic particle strikes on the circuit. Intermittent faults represent malfunctions of the device that appear and disappear repeatedly.

Spatial redundancy (also known as hardware redundancy) is a traditional technique used to handle both transient and permanent faults. A Triple-Modular-Redundant (TMR) system replicates the critical components three times and votes the results to produce an output. Hardware replication has less timing overhead since the replicas can typically run in parallel. However, extra hardware comes with high design and production cost. An alternative to handle permanent faults is task migration, i.e. to re-map the tasks running on a faulty processor to other working ones as soon as a defect is detected. Naturally, task migration is only possible if adequate hardware resources are still available after the failure. The re-mapping schemes

need to be designed carefully to guarantee the application requirements and to minimize the migration cost [9, 10]. If the remaining resources are not sufficient to execute all applications, graceful degradation may be applied. The idea is to stop some less important tasks to free resources for more critical tasks [11].

Temporal redundancy (or software redundancy) is more cost-efficient to handle transient faults [12]. One possible approach is to schedule critical tasks multiple times and perform voting of the results [13]. Another common technique is to insert checkpoints into the software and rollback the execution from a safe state if faults are detected [14, 15]. For real-time applications, software redundancy must be used with utmost care, since the overhead in time may lead to deadline violations. The schedulability issue in the context of software redundancy has therefore become a very important topic [16, 17, 18].

If multiple copies of a component are available, the outputs can be collected and voted to produce a more reliable output for the successor tasks. A common voting strategy is majority voting. Since faults are considered as rare events, the majority among the set of inputs received by the voter is considered to be correct. In this way, a voting system with N inputs can tolerate up to $\lfloor \frac{N}{2} \rfloor$ component failures.

Fault detection can also help to improve the system reliability, since, up on detecting a fault, the task can take appropriate actions to stop error propagation. Consider again the voting system as an example and assume each of its input components features a perfect fault detector that captures all possible faults. The failed component can simply perform a safe shut-down to stop sending the incorrect output to the voter. In this way, the voter will only receive correct results and the overall system is correct as long as at least one of the replicas executes correctly. In general, if combined with active redundancy, fault detection enables the system to recover from faults which could otherwise only be detected, and to detect faults that would have been undetectable.

In many utilization scenarios, the optimal implementation under multiple design constraints can only be achieved with simultaneous application of several fault-tolerance techniques. For example, the authors in [12] show how schedulable and cost-efficient solutions can be achieved by combining spatial and temporal redundancy. Also, the safety standards sometimes have special requirements or recommendations of the fault-tolerance techniques applied. For example, [19] requires a device certifiable to Safety Integrity Level (SIL) 4 [20] to implement at least hardware fault tolerance of one. This means, pure software techniques even with a large amount

of redundancy are not sufficient to achieve the desired level of reliability for SIL4. As the counterpart, a pure hardware-based solution might be prohibitively expensive. Therefore, **one of the major challenge of reliability-aware design is to find the optimal combination and configuration of FTMs**.

1.2 Tackling Reliability in Multiprocessor Systems

MPSoC is a modern architecture that integrates multiple (heterogeneous) Processing Elements (PEs) into a single chip. The PEs make use of fast on-chip interconnects for efficient communication. In contrast to traditional uni-core processors, it provides massive computational power by exploiting parallelism instead of driving a higher clock frequency. This translates to significant advantage in scalability and power consumption, especially because the clock scaling in current processors is already hitting the power wall [21]. For this reason, MPSoC is believed to be one of the major solutions to cope with the increasing complexity of future embedded systems [6, 7]. It is predicted that MPSoCs will be deployed in 45% of industrial applications by 2015 [22].

Another major advantage of MPSoC is the capability of co-hosting multiple applications. Many of the today's embedded systems involve a large number of interacting components and are facing problems with complexity management. For example, a modern car today is usually equipped with more than 100 processors to execute applications [23] such as engine control, driver assistant and entertainment. The current distributed systems based implementation is facing problems in cost and scalability. Hence, an envisioned trend is to integrate multiple functions of the system into a centralized computer. MPSoC is an ideal candidate for this purpose since it provides both high performance and high flexibility to execute a wide range of applications.

Due to the increasing relevance in the target application domains, we focus on supporting MPSoC platforms in our approach. While offering adequate hardware resources to implement advanced FTMs, MPSoCs also raise several new challenges. For our reliability-aware design scenario, two key challenges are especially important: 1) the huge design space and 2) the high complexity in system development.

Compared with uni-processor systems, the MPSoC architecture has significantly greater complexity and features a much larger number of configuration parameters, e.g., mapping of

tasks to processors, synchronization mechanisms between cores and configuration of the on-chip communication media. This translates to a large number of design alternatives and a huge design space. Configuring these parameters are not only inevitable for realizing the application functionality but also important for the non-functional performance of the system, including the timing and reliability properties that we are focusing on. The process of tuning these parameters involves evaluation/selection of various design alternatives and is commonly known as Design Space Exploration (DSE). In the reliability-aware design scenario, the DSE must consider the configuration of FTMs as part of the problem and examine its influence on the system, not only the reliability itself but also all other system properties. For example, redundancy-based FTMs increase the reliability of the system but may introduce timing overhead that influences the schedulability. In principle, the FTM configuration must be considered jointly with the classical multiprocessor mapping/scheduling problem. Thus, to tackle the first key challenge, **a Reliability-aware Design Space Exploration (DSE) approach is needed**.

The second key challenge concerns mainly the high complexity for software development. Besides the application coding, MPSoC based system development involves several extra tasks, such as Multiprocessor programming (parallelization), inter-core communication, synchronization and platform configuration, which are non-trivial for the developers. **EDA tools that automate these complex tasks are highly desirable to support the system development** [7].

1.3 Motivation and Contribution

Over the past decades, fault-tolerant embedded system design has drawn a lot of attention in the research community. In particular, reliability issues in the context of multiprocessor systems are extensively studied (see Chapter 3 for a literature review). Despite the significant progress in the field, we still observe a gap between theory and practice:

1. Most of the current work focuses only on reliability-aware DSE, aiming at calculating the mapping/scheduling of applications under reliability constraints. However, DSE is only one part of the overall problem. The designer needs approaches that cover a complete design flow, in sense other challenging tasks such as software implementation and platform configuration are also supported. To be more precise, **the current approaches lack a "front-end" tool that enables easy specification of the DSE problem,**

and a "back-end" tool, that transforms the DSE results into a real-world implementation on the target platform.

2. In the stand-alone DSE problem, the current approaches also have some limitations. The major problem is the simplifying assumptions in the fault and system models. For example, a lot of work considers either transient fault [24] or permanent fault [25] only. Also, only a certain class of FTMs is considered and the concurrent usage of multiple FTMs is not well studied. Moreover, fault detection is often assumed to be perfect, i.e., the system always detects the fault if any [12, 26, 27]. Although studying simplified versions of the entire problem constitutes an important step, these simplifications limit the practical use of the approaches. **The DSE techniques must be improved to incorporate a realistic system model [28]**(see Table 3.1 for a detailed comparison of existing approaches).

Motivated by the above observation, we propose in this thesis a new approach for fault-tolerant system design on MPSoCs. The proposed Model-Driven Development (MDD) framework supports a complete reliability-aware design flow and covers design challenges ranging from high-level system specification down to low-level implementation, thereby addressing the first concern mentioned above. For the second issue, we extend the current DSE theory by removing unrealistic assumptions and supporting advanced fault-tolerant mechanisms. With these contributions, we bring the research results in the field of reliability-aware design further into practice. In the following, the main contributions of our approach are discussed in more detail:

- We develop a binary tree based approach for probabilistic analysis of system reliability in the presence of multiple FTMs, including spatial/temporal redundancy, voting and fault detection. The analysis is generic enough to be adapted to support advanced techniques such as shared recovery slack [12].

- We propose a multi-criteria optimization approach based on Multi-Objective Evolutionary Algorithm (MOEA) for automated exploration of design alternatives. Reliability, end-to-end deadline, energy and other application-specific extra-functional constraints are supported by the DSE. Configuration of fault-tolerant mechanisms, mapping and scheduling of tasks, and configuration of the on-chip communication are considered jointly in the DSE.

- We present extended analysis and optimization techniques to combine fault detection and active redundancy to improve the overall system reliability. We propose, to the best of our knowledge, the first approach to remove the common but impractical assumption on perfection fault detection and to consider selection of fault detectors as part of the design process.
- We provide tool support for automatic generation of source code and platform configuration files in order to facilitate implementation of the design on the target MPSoC platform. In particular, we develop an approach for scheduling and configuration of a modern Time-Triggered Network-on-Chip (TTNoC) architecture.
- We are among the first to provide an MDD tool framework that supports a complete reliability-aware design flow, covering design challenges from abstract system modeling/analysis/exploration to concrete code implementation and platform configuration.

1.4 Overview

Using our approach, the design flow is separated into three major phases, namely modeling, DSE and code/configuration synthesis. Figure 1.1 shows an overview of the flow.

Modeling. In the first phase, the designer describes the system in an abstract model. We provide an MDD environment, in which the system specification is performed by instantiating objects from the meta-model library and annotating their relations. We follow the commonly accepted Y-chart paradigm [29] and explicitly separate the application and platform specification. The deployment of the application is described by mapping certain model elements (e.g., tasks) to according objects in the platform model (e.g., processors). Our modeling framework supports annotation of extra-functional properties on the modeling elements, such as the Worst-Case Execution Time (WCET) of the tasks and reliability of the hardware components. The information is required in the subsequent analysis and optimization tasks. The modeling approach serves as a user-friendly front-end tool of the reliability-aware design framework.

Design Space Exploration. The goal of the design space exploration phase is to find an optimal deployment of the application to the platform, considering all objectives and constraints configured by the designer. The DSE process is implemented as a generic optimization-evaluation loop, in which problem-specific optimization and analysis tools can be integrated. In the current implementation, we use the Multi-Objective Evolutionary Algorithm (MOEA) as the optimization engine. The optimizer traverses the design space and proposes candidate

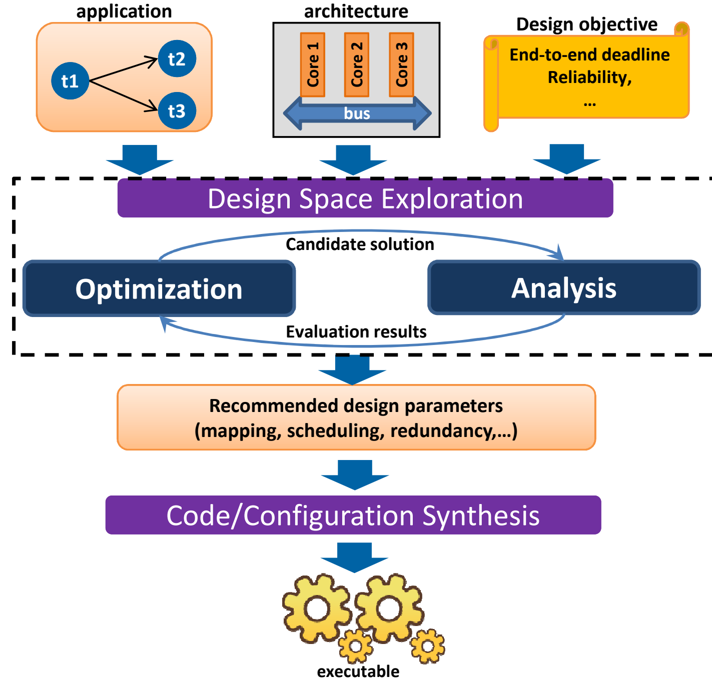


Figure 1.1: Overview of the Design Flow

solutions. The evaluator is responsible for computing the system properties, such as end-to-end latency and reliability, using according analysis tools. The evaluation results guide further optimization steps. The results of DSE are a set of recommended implementations stored also as models. In these models, the design parameters determined by DSE are automatically applied, e.g., the mapping of the application to the platform model is added and the redundant components are instantiated. The updated model is used as input for the next phase.

Code generation and platform configuration. In the final phase, the designer may select one of the implementations found by DSE and trigger automatic generation of implementation artifacts, including application source code and the platform configuration files. The source code covers both platform-independent functional code and platform-specific structural code. It is generated based on the input model and design parameters determined by the DSE. The source code directly compiles to the application image. The implementation of platform configurator is highly problem specific. For example, one of our main target platform is the ACROSS architecture [30], in which the communication is through a Time-Triggered Network-on-Chip (TTNoC). In this case, we consider TTNoC scheduling as one of the major tasks in platform configuration. The code generation and platform configuration approach constitutes

the back-end of our framework, which transforms an abstract design into a real-world implementation.

The main advantages of the proposed design flow, compared to the traditional development process for reliable systems, are accelerated development process due to high degree of automation and guaranteed consistency between development phases due to universal use of models. Fig. 1 compares the traditional approach (left) and our framework (right).

In the traditional approach, the designer extracts a scheduling model from the system specification in order to apply the reliability-aware analysis/scheduling algorithms. These algorithms may operate on different models (e.g., periodic task sets, task graphs, etc) and the designer has to guarantee the consistency. In parallel, the source code of the application is developed manually, including both functional and structural code. Finally, the scheduling results and the source code are combined to yield an executable image and to configure the execution platform.

As the counterpart, the design involves only a single manual task using the proposed framework, namely (tool-supported) system modeling. Thereafter, the system models serve as the central integration point for subsequent tasks such as analysis, DSE and the generation of implementation artifacts. Consistency is guaranteed by the fact that models are formal, type-safe description of the system. Moreover, a central representation enables easy integration of necessary tools to automate a large part of the design flow, resulting in significant speedup in the development process.

1.5 Organization

The remainder of the thesis is organized as follows:

- **Chapter 2** presents the background and system models. It starts with an introduction of the modeling approach, which is the basis of the proposed framework. Afterwards, the fault models, assumptions and the fault tolerant mechanisms considered in our approach are discussed. Finally, it presents the supported scheduling models.

- **Chapter 3** provides a literature review and a qualitative comparison of related approaches.

- **Chapter 4** focuses on reliability-aware DSE. It presents our tree-based reliability analysis and MOEA-based optimization techniques in detail, followed by simulation results.

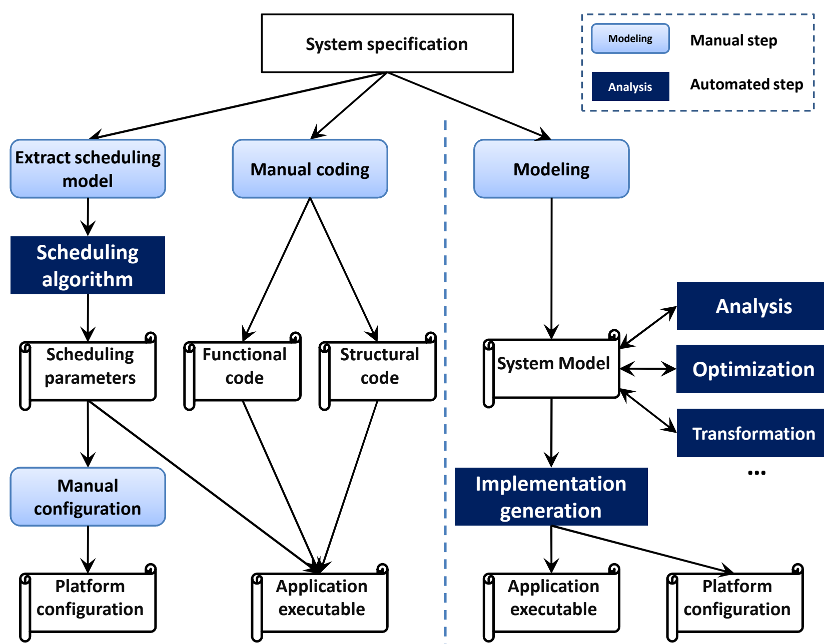


Figure 1.2: Comparison of traditional and MDD Flow

- **Chapter 5** presents an important extension to the DSE framework introduced in Chapter 4, which removes the unrealistic assumption on perfect fault detection. Updated reliability analysis and optimization techniques are discussed.
- **Chapter 6** discusses the code generation and platform configuration back-end. We present the implementation details of our template-based code generation engine. Afterwards, we use the configuration of TTNoC as an example to show how platform-specific configuration tools can be integrated to the framework.
- **Chapter 7** presents a real-world case study in automotive domain and a demonstrator in industrial automation domain. Here, we focus on demonstrating the usability of our framework in industrial design scenarios.
- **Chapter 8** concludes this thesis and provides an outlook to future work.

Chapter 2

Preliminaries

This chapter presents the background and system models used in this thesis. We start with an introduction of the underlying modeling framework that the proposed reliability-aware design approach is based on. Afterwards, we present the system models, including the fault hypothesis assumed by the analysis algorithms, the fault tolerance techniques considered in DSE and the scheduling models supported by our approach.

2.1 Modeling Approach

We realize our reliability-aware design approach as an MDD tool-chain by extending a generic modeling framework with analysis, optimization and other reliability-related techniques. The modeling approach provides us the basic **meta-models**, which define types and concepts to describe the structural and logical composition of the system. The elements in the meta-models can be **instantiated** to build concrete systems, called **models**. In an MDD environment, models are used as a common representation through individual design phases.

Figure 2.1 depicts an example MDD scenario. Here, the *meta-model* corresponds to the basic elements of the system, including component, port and channel. The *model* describes an embedded application consisting of these elements. We show a simple controller application with three components in this example. The model can be used as input to various design operations. For instance, the reliability-aware design approach may take the model as input and insert a redundant controller to enhance the reliability. The output of the operation is again a model. In this sense, a design operation can be seen as a model-to-model transformation. The MDD framework typically provides utilities to access information contained in model and to manipulate the model. The design operations can then be integrated smoothly.

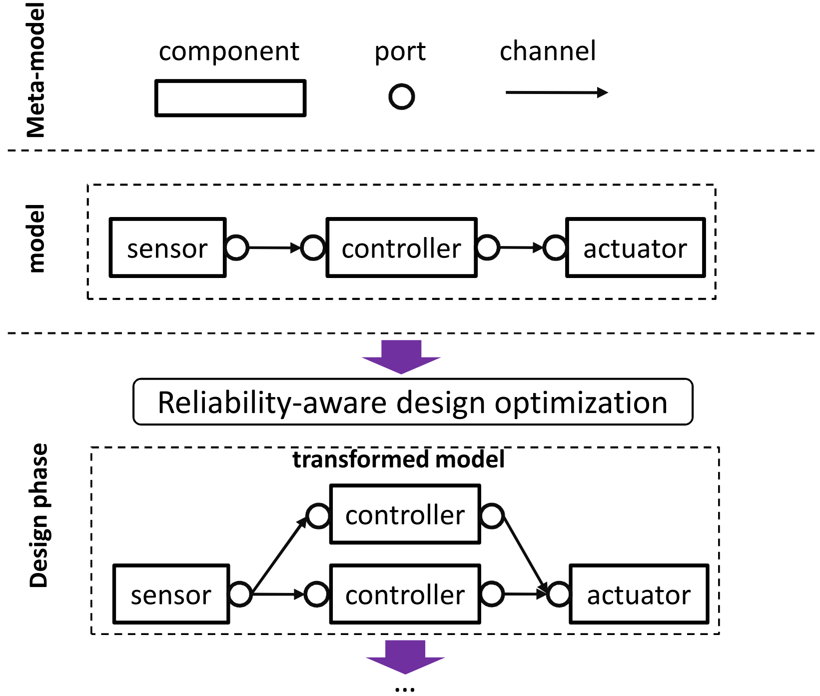


Figure 2.1: Example Model-driven Design Scenario

In our approach, we select the **capability modeling framework** proposed in [31] as the main modeling tool. It follows the Y-chart paradigm and separates the application and platform models. The overall model of the system, including both application and platform aspects, is called a **design model**. Deployment of the application model to the platform model is performed using the concept of **capability binding**(see Section 2.1.3). As the basis of the modeling framework, the *component meta-model* provides generic types, such as `Component`, `Port` and `Channel`, to describe the topological structure of the system. The system's `Components` communicate exclusively through `Ports` connected via `Channels`. The component meta-model can be refined to provide more concrete types for specific purposes, e.g. application and platform modeling, as presented in the following sections.

2.1.1 Application Model

The capability modeling framework supports several Models-of-Computation (MoCs) for application modeling. In our approach, we focus on the Kahn Process Network (KPN) model [32]. A KPN application consists of a set of concurrent *processes* representing computational kernels.

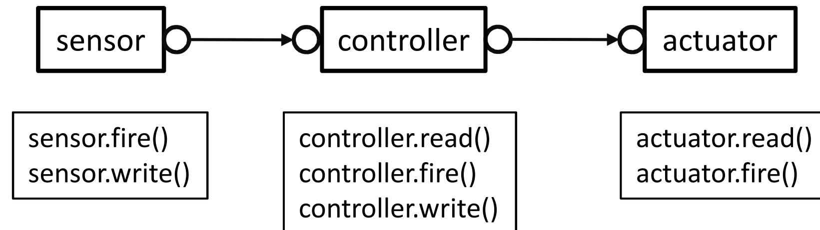


Figure 2.2: An Example of KPN Model

The processes exchange data *tokens* via point-to-point First-In First-Out (FIFO) channels. The channels have theoretically infinite buffer size. Execution of a process consists of three steps as shown in Figure 2.2:

- Read: read tokens from all input ports and store the data into a local buffer. The read function blocks the execution of the process until a valid token is available (*blocking* read).

- Fire: process the input data and store the result to a local buffer. By default, the KPN model covers only the structural information of the application (i.e., no behavior specification). It is up to the tool vendor to provide means to specify the behavior of each process. In the capability modeling framework, this can be done by associating the according model element with either a fine-grained model or an annotated C source file (see Section 6.1.2 for details).

- Write: transfer the results from the local buffer to output ports. The write operation is non-blocking.

The meta-model for KPN is refined from the basic component meta-model. It offers the following types:

- `KPNComponent`: models a process in KPN. It may contain `KPNPort`s as children for communication with other components.

- `KPNPort`: models an abstract communication port. It owns an attribute *direction*, which can be set to *input* or *output*. The data token to be transported through the port is viewed as a black box by the modeling framework. It is up to the application code to interpret the token. The token size is specified by the user so that adequate bandwidth can be allocated.

- `KPNFiFoChannel`: models a FiFo channel. It owns attributes to specify the source and destination ports as references to `KPNPorts`. It also has an attribute to specify the maximum FiFo size. Although the KPN model theoretically assumes infinite buffer, the real-world implementation must have limited buffer and handle buffer overflow.

2.1.2 Platform Model

The platform meta-model contains another set of refined types from the generic component meta-model to describe the target hardware platform. However, unlike application modeling, where generic MoCs can be used to describe many applications, platform modeling involves very specific components, e.g., a certain type of processor or a specific bus system. For this reason, the platform meta-model provides still generic types, e.g. *Processor*, *Memory* and *Link*. The platform designer can further refine the meta-model in order to describe the target platform in greater detail. As an example, we introduce derived types for the NoisII processor and time-triggered network on chip to describe the ACROSS architecture [30].

2.1.3 Mapping Application to the Platform

With the application and platform models, the designer can proceed with describing the deployment of the application onto the platform. To do this, we use the concept of *capability binding* [31] in the capability modeling framework. Here, the system is viewed from a shared-resource perspective. A component in the system may provide *resources* to other components by assigning itself as a `CapabilityResource`. As the counterpart, another component may consume the resource by generating `CapabilityRequests` to the resource owner. The request is processed by the arbitrator of the resource owner. If adequate resources are available, a portion of the resources will be assigned to serve the request (or the request is allocated to the resource). During capability binding, it is particularly important to classify the type of resources in order to guarantee that the allocation of requests to resources is constrained to matching types. The modeling framework provides the following basic resource categories:

- `Processing`: typically provided by processor cores to serve processing requests from software components, e.g. `KPNComponents`;

- `CommunicationEndpoint`: typically provided by platform ports to serve the endpoint requests from software ports, e.g. `KPNPorts`;

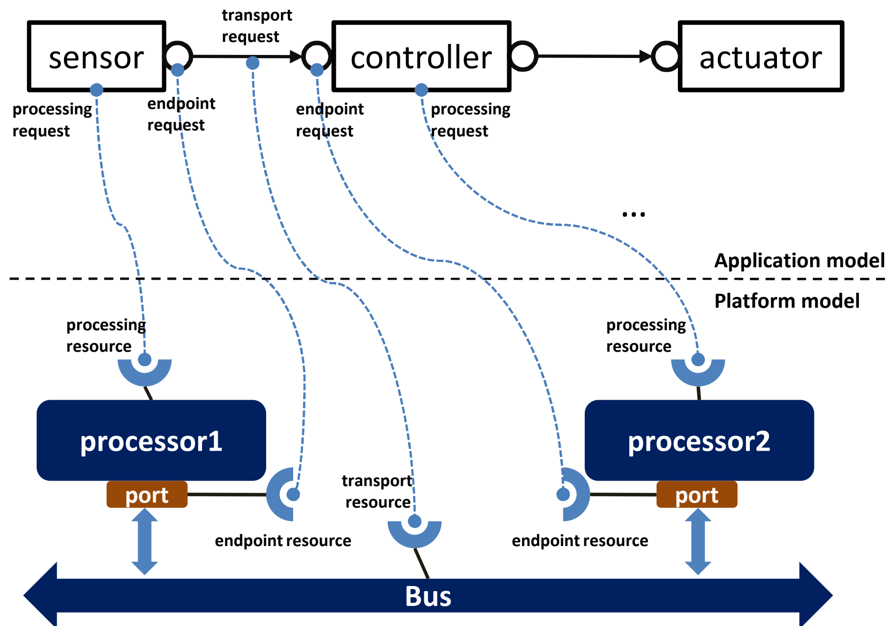


Figure 2.3: An Example Scenario of Mapping Application Model to Platform Model

- `CommunicationTransport`: typically provided by the communication media to serve requests from software channels, e.g. `KPNFiFoChannels`.

Figure 2.3 depicts a deployment scenario, where the application model consists of three tasks and the platform model comprises two processors communicating via a bus. The sensor task is allocated to *processor1* by binding its `Processing` request to the resource offered by *processor1*. Similarly, the output port of the sensor task is mapped to the hardware port by binding the `CommunicationEndpoint` request to the according resource. Finally, the software channel is implemented using the bus as transport media. For the sake of simplicity, in the rest of this thesis, mapping the application to the platform model means allocation of *all necessary requests*, unless mentioned otherwise.

2.1.4 DSE Configuration Model

The reliability-aware design space exploration process needs be configured according to the design scenario, e.g., to specify the optimization objectives and application-specific constraints. To ease this step, we extend the modeling framework to provide a new set of meta-models tailored for DSE configuration. The following configuration parameters are covered:

- Scheduling model: select the execution paradigm of the application. Currently, we support several time-triggered execution models as introduced in Section 2.5.
- Optimizer: configure the optimization algorithm used to search the design space. Currently, we use the MOEA as the main optimization engine. The MOEA-specific parameters, such as the number of iterations and size of population are configured here.
- Design objectives and constraints: depending on the design problem, the user may configure the design objectives/constraints by instantiating according objects, such as reliability objective, energy objective and deadline constraint. The objectives/constraints are equipped with respective analysis algorithms/tools. Customization of a specific objective/constraint can also be done in this step. For example, in the reliability objective, the user may select if the system requires fail-safe or fail-operational mode.

2.1.5 Implementation

The implementation of the proposed approach starts from reusing the capability modeling framework mentioned previously. The modeling framework consists of both modeling concepts as well as an implementation on the basis of the Eclipse Modeling Framework (EMF) [33]. EMF is a generic framework for building MDD tools. It allows for specification of the meta-models as well-structured data models (called the Ecore models), from which a Java implementation of the target MDD tooling can be automatically generated. The generated code covers many useful aspects, e.g., model serialization, model manipulation, model editor, etc. The tool developer then implements the application-specific functionality on top of that. For the capability modeling framework, the generic application/platform models are implemented as ECore packages and maintained as Eclipse plugins. The meta-models and models are stored in XML Metadata Interchange (XMI) format. We extend the capability modeling framework with additional meta-models that covers reliability-related aspects.

The reliability analysis, optimization algorithms and platform configuration tools are implemented in Java code and integrated to the modeling framework. The code generation engine is developed using the *Xpand* language provided by EMF. *Xpand* is specialized in specification of code templates that contain both hard-coded static contents of the output files and dynamic contents with references to objects of the input model (see Chapter 6 for details of code generator implementation). The *Xpand* and Java code may interface with each other, i.e., it is possible to invoke *Xpand* routines from Java or vice versa.

2.2 Fault Model

Following the classic terminology, a **fault** is a physical defect, imperfection, or flaw that occurs within some hardware or software component [34]. A fault may be dormant or activated. The former does not affect execution of the component whereas the later incurs an **error** [1]. The error, as the manifestation of a fault, may subsequently cause a **failure**. A failure is an observable event that the system deviates from the specified behavior. Fault-tolerance is the technique to reduce the probability of failure despite the presence of faults. It can be applied at architectural level to reduce the probability of fault-to-error transition [35], or it can be applied on application-level to reduce the error-to-failure transition. Our work focuses on the latter case.

We consider software tasks as the basic components. The DSE framework takes the error rates of tasks as input and aims at optimizing the system reliability (i.e., the system-level failure rate). The task-level error rates are obtained from an analytical **fault model**, which describes the relation between faults and its manifest in tasks. Fault models are typically proposed by reliability engineers after detailed analysis and modeling of the physical failure mechanisms [36]. The system designer may select the appropriate one for the target application domain. Our DSE approach does not have restriction on the selected fault model, as long as the task error rates can be obtained. This section discusses the fault models that we select for our experiments, concerning both transient and permanent faults.

Transient faults may cause errors in a program. It can either be that the program execution is corrupted (program hanging, segmentation error, etc) or that the program executes smoothly but delivers an incorrect output. In both cases, the task is considered as faulty. Nevertheless, since transient faults do not fundamentally damage the device, we assume that only the single task during which the faults occur is corrupted. The successor tasks can still be executed normally after a recovery process. Moreover, we focus on errors of the application program and consider the kernel software (e.g., OS scheduler, watchdog) to be fault-free [2].

For transient faults, we adopt the classical Poisson fault model, since it is well established and used in many related literature [37, 27, 24, 26]. This fault model assumes transient faults to be independent events following a Poisson distribution with a constant failure rate. Under this

[1] Dormant faults are not considered in our approach, since they are neither noticeable nor harmful. In other words, we focus on activated faults only. In this case, a fault is equivalent to an error from the designer's viewpoint.

[2] This is because we cannot apply fault-tolerant techniques such as active redundancy on the kernel software.

assumption, the following equations compute the probability that a task is executed correctly and the converse probability that the task experiences transient faults:

$$P(task\ t_i\ executes\ correctly\ on\ processor\ p) = e^{-\lambda_p w_i} \tag{2.1}$$

$$P(task\ t_i\ experiences\ transient\ faults) = 1 - e^{-\lambda_p w_i} \tag{2.2}$$

where λ_p is the failure rate of the processor p and w_i is the Worst-Case Execution Time (WCET) of task t_i. The reliability requirement concerning transient fault is given by the maximally allowable failure probability of the system.

Note that by assuming the Poisson fault model, our approach does not consider Common-Mode Failures (CMF). In reality, CMFs could cause correlation between faults, which violates the independence assumption made in the fault model. However, our approach is not intended to handle CMFs, since, as observed in [38], active redundancy is not a solution to CMF [39]. Instead, CMFs have to be mitigated by dedicated techniques, such as design diversity, architectural-level fault-containment and spatial/temporal separation [40, 41]. In general, taking CMFs into reliability analysis is relatively straightforward, e.g., using techniques mentioned in [39]. The real problem is how to estimate the probability of CMFs, which can be extremely difficult [39]. For this reason, a lot of research effort has been devoted to CMF avoidance. Here, we assume CMF avoidance techniques are systematically applied, allowing us to use the Poisson model to model transient faults.

Concerning permanent faults, we focus on defects of processing elements in the MPSoC platform and assume each individual core to fail independently. This assumption requires core-level fault containment in the underlying platform. Although fault containment is challenging to implement, it is a prerequisite to enable using MPSoCs for safety-related applications [1]. Hence, recent work [9, 42, 43] mostly make the same assumption. Also, research efforts are spent on temporal/spatial separation techniques to implement the desired fault containment property, e.g., the ACROSS architecture [30, 44]. In this context, the entire system is "alive" as long as the remaining working cores can still provide sufficient resources to execute the applications. The component-level reliability is typically given as reliability functions and the design goal is to optimize system Mean Time To Failure (MTTF) (c.f. [25]).

[1] Another possibility is to consider the entire MPSoC as a fault containment unit and apply active redundancy in distributed chips. However, this option comes at a much higher hardware cost. Moreover, it does not well exploit the benefit of the MPSoC platform, e.g., fast on-chip synchronization and communication between different redundant components.

In reliability analysis, permanent and transient faults are typically considered separately, since their physical failure mechanisms and impact on the system are significantly different. In our approach, we target on considering both types of faults in a unified manner, since this improves the system performance as shown in [45]. One obvious possibility here is to integrate the existing analysis from [25] and consider lifetime reliability as one extra optimization goal. However, additional optimization objectives reduce the optimization efficiency considerably. To overcome this problem, we assume that the reliability requirements concerning permanent faults are given as constraints. For example, the requirement could be that the system must tolerate one permanent fault of any of the processors. We develop an encoding technique to guarantee these constraints during the optimization process (see Section 4.3).

2.3 Fault Tolerance Mechanisms

We consider active redundancy as one major FTM to enhance the system reliability. Active redundancy replicates software tasks into multiple copies (replicas). The replicas can be executed on the same component (temporal redundancy) or distributed to several components (spatial redundancy). Figure 2.4 depicts an example, in which the task t_1 is replicated 3 times. In Figure 2.4a, the three copies are executed sequentially on $processor1$ to implement temporal redundancy. As the counterpart, spatial redundancy is illustrates in Figure 2.4b. Compared with 2.4a, spatial redundancy allows for parallel execution of multiple copies and therefore reduces the timing overhead. Nevertheless, extra hardware resources are needed. Temporal and spatial redundancy may also be combined to obtain compromised solutions (Figure 2.4c).

The availability of replicated software tasks allows for implementation of subsequent voters where inputs from all replicas of a task, including both temporal and spatial copies, are evaluated to produce a reliable output. By comparing the redundant results, the voter may detect or correct faults. In this thesis, we consider a **majority** voter, which generates an output if and only if more than half of the inputs have equal value. In this case, the voter can *correct* the faults, if only less than half of the replicas are faulty. If no majority is found from the voting inputs, the voter reports an error. This means a fault is *detected* but not corrected. In rare cases, more than half of the replicas can fail and send equal but incorrect result to the voter. Since the voter just selects the majority, the fault escapes. We assume that the voter features a timer to detect missing inputs, i.e., if one task instance encounters a fault and fails to send

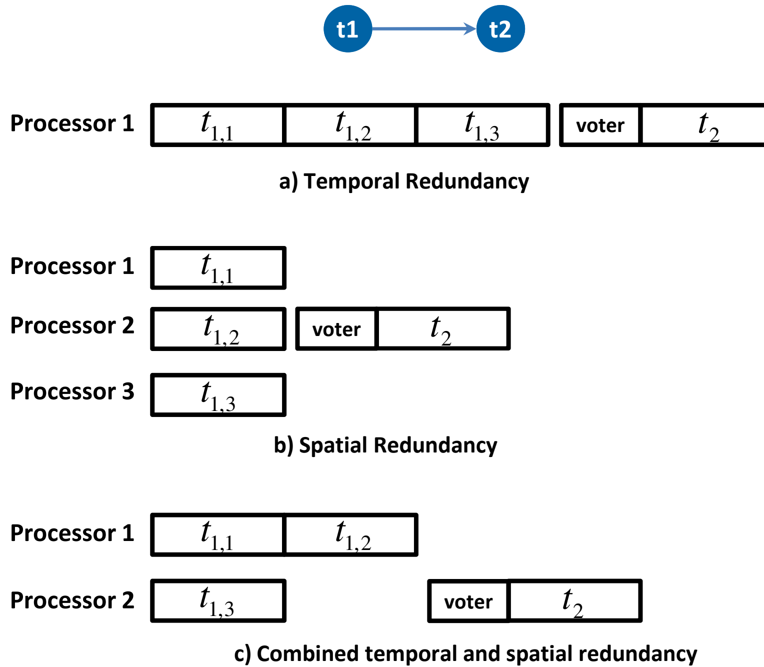


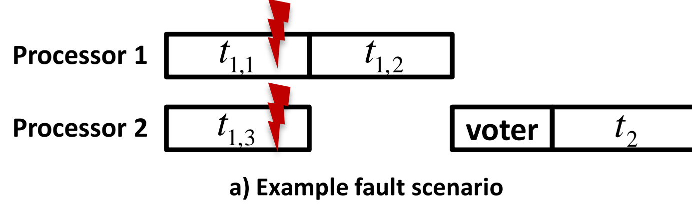
Figure 2.4: Example of Temporal and Spatial Redundancy

its result to the voter, the available data gathered from other working instances will be used for voting.

Another fault-tolerant mechanism that we consider is fault detectors embedded into tasks. This could be done in hardware, software or using a combination [46]. Software-implemented fault detection typically involves transforming the original program into an instrumented version that adds the capability to detect transient faults occurring at runtime of the program [47]. Check rules are executed at the tasks' completion to decide if faults have occurred. The arithmetic codes [47] and critical variable technique [48] are examples of this class of detectors. Hardware techniques typically introduce some monitoring functionality. For instance, the fingerprinting mechanism [49] can be used to check if the program is executed as expected.

Fault detectors implemented at individual tasks contribute to improving the system reliability, since, upon detecting a fault, the task can take appropriate actions such as safe shut-down to stop error propagation. Combining active redundancy and fault detection [1] is also beneficial in some circumstances. This can be illustrated using a simple example shown in Figure 2.5.

[1] As mentioned before, voting also has the capability of detecting faults. For the sake of simplicity, "fault detector" in the rest of this thesis stands for embedded fault detectors implemented at individual software tasks and not the voter, unless mentioned otherwise.



a) Example fault scenario

$t_{1,1}$	$t_{1,3}$	Overall
Fault undetected	Fault undetected	Fault undetected
Fault undetected	Fault detected	Fault detected
Fault detected	Fault undetected	Fault detected
Fault detected	Fault detected	Fault corrected

b) Result of voting

Figure 2.5: Example of Combined Active Redundancy and Fault Detection

In this example, the schedule from 2.4c encounters two faults on task instance $t_{1,1}$ and $t_{1,3}$ respectively. The table in 2.5b summarizes the qualitative result of voting based on the fault detection capability of individual task instances. In the first case, the fault escapes the fault detection on both $t_{1,1}$ and $t_{1,3}$, resulting in a failure of voting since incorrect results dominate. In the second scenario, the faults on $t_{1,3}$ are caught by the embedded fault detector. In this case, the faulty task $t_{1,3}$ can indicate the voter to ignore its incorrect result (or just fail silent without sending the output). The voter will consider results from $t_{1,1}$ and $t_{1,2}$ for voting. Since there are one correct result and one incorrect result, the voter detects the fault. While the third scenario is similar to the second one, the last scenario in the table shows a case that both faults on $t_{1,1}$ and $t_{1,3}$ are detected. In this case, the only correct output from $t_{1,2}$ will be recognized by the voter and the faults are corrected. As it can be seen, the 2-out-of-3 voting system in the example may detect or even correct two faults, which is not possible without using embedded fault detection.

In the presence of fault detectors, reliability analysis becomes more complicated, since the fault detection coverage also becomes a factor that influences the system reliability. To simplify the problem, most existing approaches that consider embedded fault detection assume that all faults can be detected using such fault detectors [12, 50, 51, 52, 26]. In other words, the fault detection is considered as *perfect* and all task instances can have a fail-silent behavior. Under

this assumption, only correct outputs will be sent to the successor tasks and voting becomes trivial. This assumption reduces the problem complexity significantly but raises some practical concerns (see Chapter 5 for details). In this thesis, we start with the same assumption and present our tree-based reliability analysis approach in Chapter 4. However, this assumption is relaxed and the residual error of fault detection is taken into account in the extended techniques presented in Chapter 5. Concerning error-recovery, we assume that the system will roll-back to a safe state and execute the next scheduled task after a failure. The timing overhead of recovery is considered as a constant annotated by the user [12].

2.4 System Models

The analysis and optimization algorithms take a design model as input and generate an internal mathematical representation of the system. In this way, the algorithm can be implemented in a generic way without dependency on the modeling language.

We consider the **application** $\mathcal{A}$ to be the entire software system running on the hardware platform. The application may consist of multiple independent sub-applications (also called **jobs**) sharing the platform, each of which is modeled as a directed acyclic Task Graph (TG). Each job has its own timing and reliability requirements. The TGs are extracted from the application model constructed using our modeling front-end. For a job $\mathcal{J} = (\mathcal{T}, \mathcal{E})$, the vertices $\mathcal{T} = \{t_0, t_1, ..., t_m\}$ represent a set of **tasks** to be executed and the edges $\mathcal{E} = \{e_0, e_1, ..., e_l\}$ capture data dependencies between tasks. The tasks are characterized by their WCETs preliminarily estimated using specific analysis tools (e.g., [53, 54]). Communication between tasks is via encapsulated data **tokens**, as assumed in the KPN models of computation. The vertices generate `Processing` requests and the edges generate `CommunicationTransport` requests. These requests are to be mapped to the according resources in the platform model.

As timing predictability is highly desirable for safety-related applications, we focus on heterogeneous multiprocessor architectures with predictable time-triggered communication, such as the ACROSS [30] architecture. We view the execution platform as an undirected graph. Nodes in the graph represent Processing Elements (PEs) that offer `Processing` resources. Edges of the graph represent physical link between PEs and may provide `CommunicationTransport` resources. The bandwidth of the communication media can be allocated into encapsulated messages to transfer data tokens from the application tasks. The schedule of messages $\mathcal{M}$ is described as a set of message slots $\{m_0, m_1, ..., m_k\}$. Each message slot is a four-tuple

$m = (b, f, t_{src}, t_{tgt})$, where b is the start time of the message, f is the finish time, t_{src} is the source task of the message and t_{tgt} is the sink.

In this work, we focus on tolerating faults occurred in the tasks and assume the communication to be fault-free. The main reason for making this assumption is that the FTMs we consider (redundancy, scheduling techniques, etc) are tailored for protecting tasks. Reliability of communication is typically guaranteed with other dedicated techniques such as Error Correction Code (ECC). Moreover, many architectures targeting safety-critical systems implement guarding systems that prevents interference between cores and the communication system. For example, the Trusted Interface SubSystem (TISS) in the ACROSS architecture provides interference protection in both time and value domains [30]. This enables us to consider reliability of tasks and communication separately.

2.5 Scheduling Models

We synthesize time-triggered fault-tolerant schedules for the target time-triggered MPSoC architectures. Two major scheduling models are supported, namely hierarchical combination of Time-Triggered and Static Priority scheduling (*TT-SP*) and Time-Triggered scheduling with Flexible Slack (*TT-FS*). The *TT-FS* scheme is first proposed in [12] and *TT-SP* is introduced in [45].

TT-SP. In the *TT-SP* scheme, the available processing resources are globally arbitrated in time and budgets are statically allocated to tasks. In each time slot, a set of tasks are allocated and ordered using static priorities. At runtime, the *pending* task that has the highest priority acquires the slot for execution. A task is pending if and only if 1) all the required data is available; 2) the execution is necessary, i.e., the task has not been executed successfully in previous slots [1]. Figure 2.6a shows an example of *TT-SP* schedule. The slot S_1 is allocated with two tasks and t_1 has higher priority. In this case, the re-execution of t_1 will take place in S_1 whenever necessary, e.g., as shown in Figure 2.6c where the first instance of t_1 fails. Task t_2 may execute in S_1 only if the high-priority task t_1 finishes before the start of S_1 (Figure 2.6b). A *TT-SP* schedule can be described as a set of non-overlapping slots $\mathcal{S} = \{s_0, s_1, ..., s_n\}$, each being a four-tuple $s = (b, f, p, T)$, where b is the start time of the slot, f is the finish time, p is the processor on which the slot is allocated and T is a list of tasks with decreasing priority

[1] Note that *TT-SP* and *TT-FS* both incorporate the perfect fault detection assumption. In this case, if a task is executed without faults, we are sure that a correct output is already available and it is not necessary to execute other copies of the same task.

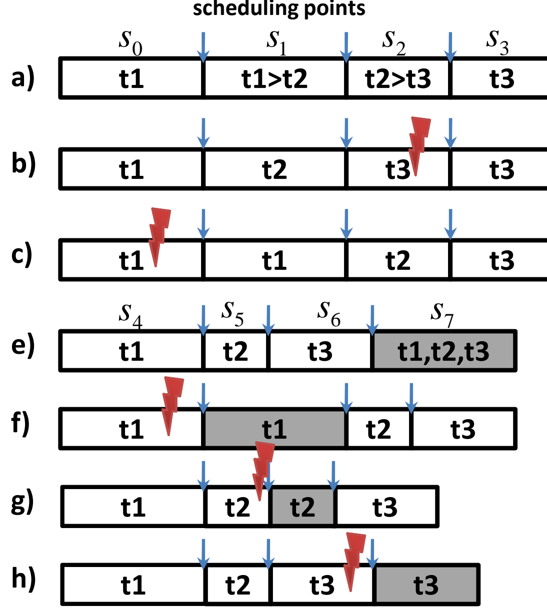


Figure 2.6: Example of *TT-SP* and *TT-FS*

assigned to s. The size of a time slot is determined by the longest worst-case execution time (WCET) of all tasks assigned to it. An important feature of *TT-SP* is that the start/end time of each slot is fixed and does not have dependency on the occurred faults.

TT-FS. In the *TT-FS* scheme, two types of time slots are scheduled, namely normal slots and *slack slots*. The latter are intended to be used for re-execution of tasks misbehaving due to transient faults. Slack slots are often shared by multiple tasks. Figure 2.6e shows an example, in which the slack slot S_7 is shared by t_1, t_2 and t_3. The slack slots only reserve time for re-execution but do not have a fixed start time. Instead, they will be utilized whenever necessary. In Figure 2.6f, the first instance of t_1 encounters a fault and S_7 is used immediately to re-execute the same task. The normal slots S_5 and S_6 are postponed in this case. Figure 2.6g and 2.6h are another two scenarios, in which S_7 is used to re-execute t_2 and t_3, respectively. Naturally, the size of the slack slots must be no smaller than the WCET of any tasks assigned to it. To describe a *TT-FS* schedule, the four-tuple $s = (b, f, p, T)$ needs to be extended with an additional binary attribute to denote if the slot is a slack slot or not.

The analysis and optimization techniques presented in this work support both *TT-SP* and *TT-FS*. For the sake of simplicity, we focus mainly on the *TT-SP* scheme for the rest of this thesis. Nevertheless, we present the details on how to utilize the same techniques for *TT-FS*. It is up to the designer to choose one of the scheduling schemes.

Implementation. To implement a *TT-SP* schedule, the complete schedule table is stored statically at all components. At each scheduling point, the pending task with highest priority is issued for execution. Once a replica of a task is finished successfully, the other replicas of the same task are removed from subsequence time slots for the current iteration to avoid duplicated executions. Another situation that must be avoided is that some task in previous slot becomes "hanging" due to fault and blocks the execution of subsequence tasks. A hardware watchdog can be used for this purpose. The implementation of a TT-FS is a bit more complicated, since the schedule has to be adapted at runtime depending on the faults occurred. In general, once a fault occurs, the scheduler has to make emergency response and try to achieve a correct execution by using the slack slots. Detailed explanation of the implementation with an example is presented in [55].

Scheduling Model with Imperfect Fault Detection and Voting. The DSE approach supporting *TT-SP* and *TT-FS* is presented in Chapter 4. Both of the scheduling models rely on the assumption that all faults are detected at completion of the task. To be more precise, the scheduler has to know concrete results of previous slots (whether faults have occurred) to determine the task to be executed in the subsequent slots. Although these two scheduling models are of great academic interest, the perfect fault detection assumption causes several practical issues [28, 56]. As we target on bringing the research results into practice, we propose extended techniques in Chapter 5 to remove this assumption. In this case, the scheduling model has to be adapted accordingly.

The updated scheduling model is called Time Triggered Schedule with imperfect Fault Detection and Voting, in short *TT-FDV*. In this model, we disable sharing of time slots between multiple tasks, i.e., each slot is assigned to only a single task. The reason is two-folds. On the one hand, the performance of slot sharing decreases with imperfect fault detection, since, if a fault escapes the fault detection, the scheduler will incorrectly assume that the task has been successfully finished and continue with other tasks without re-executing the faulty task. On the other hand, from the introduction in Section 2.3, we see that the voter needs to gather inputs from all replicas of a task in order to have the best performance. Hence, the voter must be inserted after completion of all replicas. The slot sharing causes non-determinism in the execution sequence of tasks and complicates the insertion of voter and the subsequent reliability analysis. Due to the reasons above, the *TT-FDV* model does not involve slot sharing. *TT-FDV* is introduced in more detail in Section 5.1.

Chapter 3

Related Work

This Chapter provides a literature review in the field of reliability-aware system design as well as model driven development. We compare our approach with existing work to further motivate this thesis.

3.1 Reliability-Aware Embedded System Design

Reliability-aware design consists of several challenging tasks. First of all, **reliability modeling** is a fundamental step, which aims at understanding the physical failure mechanisms and provide reliability models (also called fault models) that can be used to predict the reliability of the component prior to its implementation. These models are the foundation of subsequent **reliability analysis**, which deals with a system of components and evaluates the system-level reliability. Here, a major challenge is to take into account the influence of fault-tolerant mechanisms and the interaction between components. Finally, guided by the analysis results, reliability-aware design aims at building systems that are optimized for reliability. An overview of this area can be found in [57, 58].

Reliability Modeling. As mentioned above, reliability modeling is based on the physical fault processes. For permanent faults, several processes are observed by the researchers, such as electromigration (EM) and time dependent dielectric breakdown (TDDB). Based on a large set of experiments, the researchers develop empirical models to estimate the reliability in terms of common metrics, such as MTTF. For example, EM and TDDB are usually modeled using lognormal and Weibull distributions, respectively [59, 60]. Reliability modeling can be performed in a hierarchical manner, i.e., we obtain models for the basic components and compose them to yield models for larger components or systems. The authors in [61, 62] propose a

component-level reliability model that takes temperature estimates into account. Afterwards, they extend this model to cover multiprocessor platforms where back-up elements are available. The work [36] proposes a framework that integrates device, component and system level models.

Unlike permanent faults, which are mostly related to wear-out and the manufacturing process, transient faults mostly occur due to environmental conditions [63], e.g., energetic particles, noise and electromagnetic interference. Hence, they are usually modeled using random process. One classic model is from Shatz and Wang [37], which assumes that the occurrence of transient faults follows a Poisson law with a constant error rate. The reliability model is also extended to cover the effects of voltage scaling on reliability [64, 51].

Reliability Analysis. One classical approach for reliability analysis is Reliability Block Diagram (RBD) [65]. A RBD is a Directed Acyclic Graph (DAG), where each node represents an element of the system and each edge models a causality relationship between two nodes. RBD can be used to compute the reliability of the entire diagram based on known reliability of the nodes. Replicated components can also be handled. RBD assumes that the system is operational as long as at least one replica of each component is non-faulty. One main drawback of RBD is that the computational complexity is exponential in the size of the diagram. The authors in [66] propose a minimal cut set method to efficiently approximate the reliability computation. Our tree-based analysis presented in Section 4.1 is similar to RBD but is much more generic. It supports advanced scheduling techniques such as shared recovery slack and handles a larger set of FTMs.

Fault Tree Analysis (FTA) [67] and Failure Mode and Effect Analysis (FMEA) [68] are two techniques commonly used in industry. FTA is a top-down approach that uses boolean logic to model how undesired state of the system could be caused by low-level events. In contrast, FMEA is a bottom-up method that systematically analyzes how hypothesized component failures may affect the system. In many application domains, such as aerospace, the common practice is to perform both FTA and FMEA.

As mentioned above, reliability analysis must take the influence of fault-tolerant mechanisms into account. Hence, researchers also propose dedicated analysis techniques for a specific set of FTMs, e.g., in [69, 24, 50]. These analysis methods are tightly coupled with the respective design approach. We will discuss them in the following sections when the associated design approaches are reviewed.

3.1.1 Fault-tolerant system design focusing on permanent faults

When focusing on permanent faults, the system reliability is often referred to as lifetime reliability. The common measure is Mean Time To Failure (MTTF). Popular mechanisms to increase the MTTF of the system include hardware hardening, hardware redundancy and task migration [58].

The work [42] presents a lifetime-aware task mapping approach on chip multiprocessors. The authors focus on wear-out related permanent faults and take into account temperature-dependent failure mechanisms. The Ant Colony Optimization (ACO) algorithm is used to search for the optimal task mapping schemes. The authors in [43] consider a similar problem. In particular, the aging effect of components in a multiprocessor system is taken into account. *Feldmann et al* [70] presents an approach that focuses on analyzing the feasibility of the system under permanent faults. They define a new metric *k-bindability*, which specifies the property that a feasible binding of the application to the platform exists even if any k components fail. Quantified boolean formulas are used to calculate the k-bindability of a system. *Glaß et al* [25] extend the approach in [70] and consider redundant binding of a task to multiple resources for the sake of reliability improvements. The system behavior in the presence of redundancy is described using the so-called *structure function* and represented as Binary Decision Diagrams (BDDs). A path in the BDD towards *true* represents a combination of faults that is tolerable with the current system setup. They present techniques to evaluate the system-level reliability based on component-level reliability models. The analysis is integrated into an MOEA based optimization framework to calculate the best task bindings [69]. The same authors further consider the automatic insertion of voting components in [71]. In another approach proposed by *Pinello et al* [72], spatial redundancy is utilized to handle permanent faults in distributed systems. A heuristic algorithm is provided to explore solutions that mask certain faults based on the user-specified fault hypothesis.

In [73] the authors utilize online fault detection and task migration to maximize the expected MTTF. On detecting certain faults, the system is restarted and the tasks are re-allocated to the remaining non-faulty components according to a pre-computed plan. The task migration cost is not considered, instead, the focus is on increasing the MTTF as much as possible. The work [74, 9] addresses a similar problem but considers the migration cost. In particular, *Lee et al* [9] focus on static task re-mappings under throughput constraints for streaming applications. They also adopt a semi-static approach that loads a pre-determined migration plan upon failure of certain processor. The work [75] proposes to extend the MTTF by active allocation of

slack in the system. For example, some processors can be intentionally replaced by high-performance ones, so that tasks from failed processors have higher possibility to be migrated. When task migration is used, an important issue is to guarantee the application requirements after the recovery while minimizing the migration cost. *Yang et al* [10] propose an approach for generating schedules with predictable response to faults. They partition the initial schedule into several bands, which are designed in a way that the capability of re-mapping tasks is embedded. The work is extended in [76] to minimize the latency of applications.

Our approach considers permanent faults but handles it differently from all approaches mentioned above. The major reason is that we aim at supporting both transient and permanent faults. In general, transient faults are more complex to handle from the design perspective, mainly because 1) they are probabilistic events that may appear multiple times and 2) they can be mitigated using a larger number of FTMs. The FTMs that tolerate permanent faults, e.g., hardware replication, typically tolerate transient faults as well, but not vice versa. For this reason, our DSE approach is designed primarily for transient faults. We use an analysis-optimization based approach to search for the best design. The support of permanent faults is built on top of that. Here, we consider permanent fault tolerance as hard design constraints instead of an extra optimization objective, in order to improve the efficiency of the overall approach (see Section 4.4).

3.1.2 Fault-tolerant system design considering transient faults

Transient faults have also drawn a lot of attention in the research community. The problem is often considered jointly with real-time scheduling, since scheduling techniques can be used to embed temporal redundancy, which is an efficient mean to handle transient faults. Hence, many researchers consider the problem as "fault-tolerant scheduling" and base their work on advanced scheduling algorithms [24, 77]. From another angle, the problem can also be seen as an optimization problem that features some degrees of freedom (e.g., configuration of FTMs, mapping and scheduling of tasks) and has some constraints (e.g., reliability goal, deadline, scheduling length). Therefore, many other approaches try to solve it using design space exploration. The scheduling-based approaches typically provide efficient heuristic algorithms that are tailored for a specific setup. In particular, they are restricted to the selected fault model, fault-tolerant techniques and design objectives. In contrast, the DSE approach is more generic and may handle multiple (in some approaches also configurable) FTMs and user-specified de-

sign constraints. We have followed the DSE approach in this thesis since it aligns better to the overall objective of our model-based fault-tolerant development framework.

The work [13] presents an approach using scheduling techniques. Here, replicated tasks are selectively inserted using the otherwise wasted resources to enhance the system reliability in a best-effort manner. The authors in [78] present an approach for static scheduling with fixed fault-tolerant mechanism assignment. To be more specific, each task is replicated twice so that a single processor failure can be handled. *Girault et al* [24] consider fault-tolerant scheduling with active task replications and present a bicriteria heuristic algorithm. Besides task scheduling, the algorithm also determines the number of replications needed to achieve certain reliability goal. Only spatial redundancy is considered and the replicas of a task are always scheduled on different cores.

A series of work from *Izosimov et al* tackles the problem using an optimization approach. In [12], they combine spatial and temporal redundancy and propose novel techniques to share the re-execution slack among multiple tasks. For optimization, the single-objective tabu-search algorithm [79] is adopted to minimize the scheduling length. In [14] *Pop et al* study the problem using check-pointing and roll-back techniques. The authors in [52] utilize a hybrid scheduling approach to handle mixed hard and soft real-time tasks. The aforementioned work [12, 14, 52] is based on a simplified fault model. Instead of modeling faults as probabilistic events, they assume that the system may experience at most N faults and these faults may occur in any component of the system. Under this assumption, the authors focus on automatic derivation of the optimal task mapping, scheduling and FTM configuration (e.g., the amount of replication and placement of check points). The simplified fault model has the limitation that the distinct failure probabilities of the underlying hardware components are not taken into account. In the follow-up work [50], a more accurate probabilistic analysis is presented. Nevertheless, this analysis considers only temporal redundancy. Our DSE approach considers a similar problem as the work mentioned above [12, 14, 52, 26]. We propose a generic probabilistic reliability analysis to compute the system reliability in presence of both spatial/temporal redundancy and shared re-execution slacks. In addition, our evolutionary algorithm based optimization approach supports multiple optimization objectives, e.g., reliability, schedule length and resource utilization. Another major advantage of our approach is that permanent faults can be taken into account efficiently using the proposed virtual mapping technique (see Section 4.4).

One of the early approach COFTA [80] considers automatic synthesis of reliable systems using software assertion and duplicate-with-compare techniques. FTMs are embedded to the

application's task graph using heuristic algorithms before the scheduling phase. The authors aim at tolerating a single fault in the system, either transient for permanent. *Kandasamy et al* [81] present an approach for transient fault tolerance using temporal redundancy. They propose the concept of transparent recovery, which ensures that recovery process does not affect other operations of the system. Heuristic algorithms are developed to compute fault-tolerant schedules under a fault model that assumes a single fault per processing unit. The authors in [55] adopt a similar system model and focus on the tradeoff between performance and transparency. *Jhumka et al* [82] propose a Genetic Algorithm (GA) based approach for DSE under reliability, performance and cost constraints. They use a simple fault model and assume that the failure probability of a task is equal to the failure probability of the processor it is running on. Spatial replication is utilized to improve the system reliability. *Stralen* and *Pimentel* present a DSE based approach for fault-tolerant deployment of applications on MPSoCs [83]. The FTMs are described as patterns that are applied to the application model. Only spatial redundancy (DMR and TMR) patterns are considered so far.

A recent work that is close to our approach is from *Bolchini et al* [84, 85, 86]. They also propose a generic DSE framework that supports a configurable set of FTMs, such as active redundancy, fault detection and voting. Moreover, they also synthesize time-triggered fault-tolerant schedules using GA. One major difference between their work and ours is the fault model. They adopt a similar fault model as *Izosimov et al* [12] and aim at handling a maximum number of concurrent faults. The reliability of the execution platform is modeled using a simple *qualitative* tag, e.g., if the processor supports fault detection or fault tolerance. Such a simplified fault model ignores a lot of *quantitative* reliability information, e.g., the intrinsic failure rate of the processors and the coverage of the fault detection utility. Only coarse evaluation of the system reliability can be provided in this case. In contrast, our probabilistic reliability analysis takes all these factors into account and provides precise quantitative results to guide the optimization process.

Other work also studies the tradeoff between reliability and other design objectives, such as energy [26] and cost [18]. The work [87] considers reliability-energy joint optimization. They focus on optimizing the mapping of applications to reduce the number of soft errors. No FTMs are considered. In [88], the authors present a Constraint Logical Programming (CLP) based approach for scheduling and voltage scaling of fault-tolerance systems. *Zhu et al* show that voltage scaling has direct and adverse effects on system reliability [64]. They study static scheduling approaches for energy minimization under reliability constraints [26]. The

core idea is, instead of using all available slack time for energy management, a portion of the slack is especially reserved to schedule task re-executions, such that the reliability loss can be recuperated. In [51] the same authors also consider a shared recovery slack technique similar to the one proposed [12].

An important limitation of the work mentioned above [12, 14, 52, 64, 26, 88] is the assumption on perfect fault detection. To reduce the problem complexity, the authors assume that all transient faults can be detected when a task is completed and the timing overhead of fault detection is contained in the WCETs of tasks. However, fault detectors, especially those have high detection coverage, may come with high resource and timing overheads [47]. These resources could potentially be used for other purposes, e.g., to implement more replicas. Seen from another angle, the overhead in fault detection may limit the resource available for active redundancy, resulting in sub-optimal system reliability. Hence, it is important to consider optimization of fault detector implementation in the design flow. The previous work [28] discusses in particular the selection of error detectors. Experimental results verify that certain configuration using imperfect fault detectors combined with active redundancy can outperform the approaches that utilize only perfect fault detectors. For this reason, we consider supporting imperfect fault detection as an important goal of the DSE approach (see Section 5).

In [46, 89], the authors consider another important tradeoff, namely the tradeoff between hardware-implemented and software-implemented fault detection. They propose to selectively implement fault detectors in an FPGA fabric tightly coupled with the processor, so that the fault detector can run in parallel with the original program and the timing overhead of fault detection can be reduced. Given limited FPGA resource, it is critical to decide which fault detector goes to hardware. Optimization techniques are proposed for this purpose. FPGA-accelerated fault detection is currently not considered in our work. To take this issue into account, the problems considered in [46] and [28] have to be combined. Here, the design goal is to decide both *which* fault detector to implement and *where* to implement. We consider this combined problem as part of the future work.

3.1.3 Qualitative Comparison of Related Work

Table 3.1 provides a qualitative comparison of representative related work. The first part of the table (columns 2 and 3) summarizes the fault model utilized by the individual approaches. Some early work in the field [13, 81] considers a single-fault model. This is a reasonable simplification since faults typically occur with very low probability. These initial approaches

have been extended with a fault model that covers multiple faults [12, 50, 86]. Still, probabilistic events are the most precise way to describe the physical properties of faults. The major challenge of using a probabilistic fault model is the complexity of corresponding reliability analyses. For approaches based on a fault model that covers a certain number of faults, no detailed reliability analysis is needed, since the design objective is merely to add sufficient FTMs to tolerate all assumed faults. In contrast, a probabilistic approach needs quantitative evaluation of the system reliability after applying the FTMs. Recent approaches contribute appropriate reliability analysis techniques, but supports only very limited FTMs [24, 50]. For this reason, we aim at providing a generic reliability analysis in this paper. Also, our framework is the only one that supports both transient and permanent faults[1].

The second part of the table lists the supported FTMs in the individual approaches. As it can be seen, a major limitation of current approaches is that only a small set of FTMs is supported, making it infeasible to evaluate the tradeoff between various FTMs to find the system-wide optimal solution. Only the recent work presented in [86] and our approach try to support a *configurable* set of FTMs. The configurability enables the user to select candidate FTMs for the specific application domain and is therefore essential for the practical applicability of the approach. In both [86] and our approach, the configurability is achieved by encoding FTMs as some kind of model transformation. In [86], the FTMs are applied as application task graph transformation before the classical mapping/scheduling phase. In our approach, they are inserted in the encoding phase of the evolutionary algorithm (see Section 4.3). The transformation-based implementation guarantees the extensibility to new FTMs. Nevertheless, as discussed before, [86] is still significantly different from our approach due to the use of a simplified qualitative fault model.

3.2 Model-Driven Software Development

As mentioned before, our reliability-aware design flow is based on a Model-Driven Development (MDD) approach. This section reviews related work in this area and provides a qualitative comparison.

MARTE [90] (Modeling and Analysis of Real-Time and Embedded Systems) is a UML profile tailored for embedded systems. It provides means to model software and hardware components as well as mapping of the software to the platform. MARTE puts special emphasis

[1]The COFTA approach [80] considers both transient and permanent faults. However, the fault model assumes a single fault, i.e., it is either a transient or a permanent fault.

Approach	Fault Model		Reliability Analysis	Fault-Tolerant Mechanisms Supported				
	transient fault	permanent fault		spatial redundancy	temporal redundancy	fault detection	voting	other
COFTA [80]	single	single	none	√	×	×	×	software assertion
Lee [9]	×	single	none	×	×	×	×	task migration
Glaß [25]	×	probabilistic	quantitative	√	×	×	×	
Reimann [71]	×	probabilistic	quantitative	√	×	×	√	
Pinello [72]	×	multiple	none	√	×	×	√	
Xie [13]	single	×	none	×	√	×	×	
Kandasamy [81]	single	×	none	×	√	×	×	
Zhu [26]	multiple	×	none	×	√	×	×	
Jhumka [82]	multiple	×	quantitative	√	×	×	×	
Girault [24]	probabilistic	×	quantitative	√	×	×	×	
Izosimov [12]	multiple	×	none	√	√	×	×	shared re-execution
Izosimov [50]	multiple	×	quantitative	√	√	×	×	hardware hardening
Lifa [46]	multiple	×	none	√	√	√	×	
Pop [14]	multiple	×	none	√	√	×	×	check-point roll back
Stralen [83]	multiple	×	none	√	×	×	×	
Bolchini [86]	multiple	×	qualitative	√	√	√	√	
Proposed	probabilistic	multiple	quantitative	√	√	√	√	shared re-execution

Table 3.1: Qualitative Comparison of Related Work in the Area of Reliability-Aware Design

on supporting analysis of system properties. As analysis domains typically have different terminology, it defines generic concepts as foundation to integrate refined modeling and analysis tools. Although MARTE has the potential for integration of DSE and code generation tools to support reliability-aware design, no such tools are shipped as default packages. SysML (Systems Modeling Language) is another general purpose modeling language based on UML. It removes many of UML's software-centric constructs to reduce the language size and adds more support to non-software components. Nevertheless, the platform modeling support is very basic. AADL [91] is a similar approach that covers application and platform aspects. However, it only provides coarse-grained models. The application components are described using data, subprogram, thread, thread group and process. The execution platform includes processor, memory, bus and device. In comparison with our approach, MARTE, SysML and AADL focus on providing a generic MDD framework instead of a concrete design flow. In this sense, they are more comparable with the capability model framework, which is the underlying modeling concept of our approach (cf [31]). In principle, our DSE and code generation tools could also be integrated to other modeling environments. We select the capability modeling framework because it provides a self-contained fine-grained application/platform description that fits our purpose.

Ptolemy II [92] is a modeling framework focusing on component-based heterogeneous systems. It uses directors to regulate execution and communication of software components. This allows for construction of systems using different MoCs. Ptolemy focuses on behavioral modeling and does not consider mapping of software components to a concrete platform. The Metropolis [93] framework supports detailed application and platform modeling. It considers analysis and synthesis of the design (e.g., architecture configuration parameters) as one of its main design activity. Nevertheless, it only provides syntactic and semantic mechanisms for the user to plug in tools to perform the required design task. In its successor Metropolis II [94], automatic DSE is identified as one of the major goals.

DOL is an MDD approach close to our work [95]. It focuses on streaming applications on multiprocessor systems. A complete design flow is supported, including modeling, DSE and code generation. Although the DSE implementation is also generic and multi-criterion, reliability is not considered as one of the key non-functional properties.

Table 3.2 gives a qualitative comparison of relate work in the area of MDD. The first column summarizes if the modeling language provides a fine-grained description of the system. A fine-grained model typically contains more detailed information and enables more comprehensive

Approach	platform and application modeling	application to platform mapping	code generation
MARTE	fine-grained	manual	×
SysML	coarse-grained	manual	×
AADL	coarse-grained	manual	×
Ptolemy II	fine-grained	×	√
Metropolis	fine-grained	manual	√
Metropolis II	fine-grained	automatic DSE	√
DOL	coarse-grained	automatic DSE	√
Proposed	fine-grained	automatic DSE	√

Table 3.2: Qualitative Comparison of Related Work in the Area of MDD

analysis and code generation in the later phases. The next column considers the deployment process, i.e., if the application to platform mapping is performed manually or with tool support. As mentioned in [94] automatic mapping is a highly desirable feature for an MDD tool framework, especially because modern platforms are becoming more complex. Finally, the last column illustrates if (complete or partial) executable code can be generated from the model. The approach proposed in this thesis implements all three important features mentioned above and is already shipped with necessary tools to support a reliability-aware MDD design flow.

Chapter 4

Reliability Aware Design Space Exploration

This chapter presents details of our reliability-aware design space exploration approach. We start with the tree-based analysis algorithm for evaluation of the system reliability in the presence of FTMs. Afterwards, we show how the analysis is integrated into a multi-objective optimization process to guide the search in design space. The focus here is the encoding technique that transforms the DSE problem into an optimization instance. Finally, experimental results are presented.

4.1 Reliability Analysis

In the proposed framework, the reliability analysis focuses on computing the system-level reliability of a given design under impact of transient faults. Permanent faults are taken into account using an encoding technique in the optimization process (see Section 4.4). Computing the system reliability is a very difficult problem, especially in the presence of fault-tolerant mechanisms such as active redundancy. Recent work [77] especially analyzes the complexity of reliability analysis. The authors distinguish two types of schedules, namely *strict schedules* and *general schedules*. The strict schedules obey a rule that if a task t has a data dependency on task t', all replicas of t' must be completed before any replicas of t start. With this restriction, the execution results (success or faulty) of predecessor tasks will have no influence on the start time of successor tasks. In this way, the tasks can be considered independently in reliability

analysis and a closed form formula can be derived [24]:

$$Pr(\mathcal{S}, J) = \prod_{t \in J} (1 - (1 - Pr(t))^{num(t)}) \tag{4.1}$$

where $Pr(\mathcal{S}, J)$ is the reliability of job J achieved by schedule $\mathcal{S}$, $Pr(t)$ is the reliability of task t and $num(t)$ is the number of replicas that task t features. As for the general schedules, the authors prove that the problem is at least as hard as NP-Complete problems [77].

The reliability analysis for *TT-SP* and *TT-FS* schedules is even more difficult than the ordinary general schedules. First of all, a larger set of FTMs are utilized. In the current work [24, 77], the replicas of a task are always mapped to different processors to implement spatial redundancy. For *TT-SP* and *TT-FS*, concurrent spatial and temporal redundancy has to be considered. Second, shared time slots must be supported. In [24, 77], each slot is used for exactly one task. At the beginning of each slot, the scheduler automatically knows which task is to be executed. For *TT-SP* and *TT-FS* schedules, slots can be shared by multiple tasks. The actual utilization of slots depends on the execution history of previous slots. Figure 4.1 depicts an example, in which the utilization of slack slots S_2 and S_3 depends on the execution results (success or faulty) of previous slots S_0 and S_1. In particular, the execution results on one processor might also influence the execute sequence on other processors. In the same figure, if the instance t_1 succeeds in slot S_1 on processor p_2, the message will transfer the correct result to processor p_1 so that the slot S_2 can be left for t_2 (4.1c). Otherwise, the slot S_2 has to be used for re-executing t_1 (4.1b). The analysis algorithm must also maintain the execution history to select the correct task for the next slot. New analysis techniques are needed to conquer the extra complexity. In principle, to obtain the system-level reliability for *TT-SP* and *TT-FS* schedules, we need to carefully investigate which combinations of faults are tolerable by a certain schedule and which combinations are not. In the next sections, we propose a binary tree based approach.

We describe a combination of faults occurring in a system by a *fault scenario*:

Definition 1 (Fault Scenario)**.** *A fault scenario is a vector* $\boldsymbol{x} = \{x_0, x_1, ..., x_n\}$*, which contains for each scheduling slot* s_i [1] *a variable* $x_i \in \{1, 0, NA\}$*. It encodes the execution result of* s_i*:* x_i *is* 1 *if the slot executes some task successfully and* 0 *if the execution fails;* x_i *is* NA *if the slot* s_i *is not used, i.e., each task in* $s_i.T$ [2] *is either not ready or finished earlier and no task is actually executed in* s_i*.*

For the given job J, a fault scenario $\mathbf{x}$ is *tolerable* by a schedule $\mathcal{S}$ if J is still executed correctly in presence of faults specified in $\mathbf{x}$. The entire set of fault scenarios that are tolerable

[1] Recall that, for TT-FS schedules, a scheduling slot is 4-tuples as introduced in Section 2.5.
[2] The notation s.X denotes the element X in the tuple s in the entire paper.

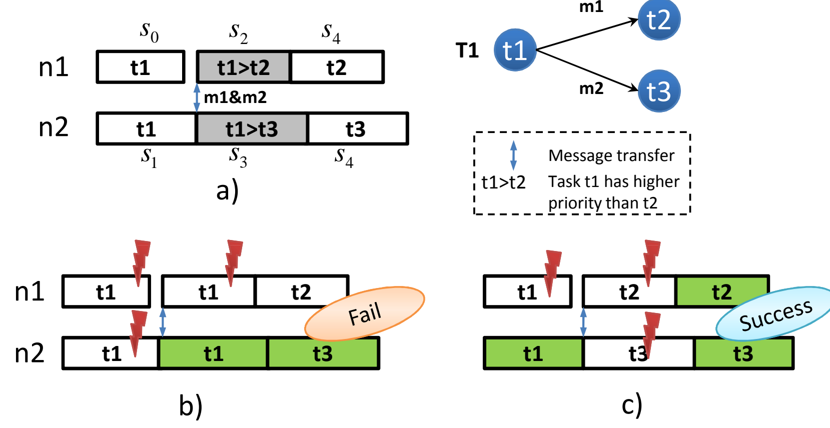


Figure 4.1: Example Execution Scenarios for *TT-SP*

by schedule $\mathcal{S}$ is called the *working set* of J, denoted as $W(\mathcal{S}, J)$. The overall probability that J is correct can be obtained by summarizing the occurrence probability of all fault scenarios in the working set:

$$Pr(\mathcal{S}, J) = \sum_{\mathbf{x} \in W(\mathcal{S}, J)} Pr(\mathbf{x}) \tag{4.2}$$

Before presenting the calculation of the working set, we first introduce some intermediate notations. Let $S(t_j)$ represent the set of slots to which task t_j is assigned, i.e., $S(t_j) = \{s \in \mathcal{S} | t_j \in s.T\}$. The boolean *request* variable $r_{i,j}$ evaluates to *true* if the task t_j *requests* to execute in slot s_i and $false$ otherwise. The boolean *utilization* variable $u_{i,j}$ is *true* if the slot s_i is actually used to execute task t_j and $false$ otherwise. For the case of static priority scheduling, $u_{i,j}$ computes to:

$$u_{i,j} = r_{i,j} \wedge (\bigwedge_{\substack{t_l \in s_i.T \wedge \\ priority(t_l) > priority(t_j)}} \neg r_{i,l}) \tag{4.3}$$

that is, s_i is utilized by task t_j only if t_j has the highest priority among all tasks requesting the slot. An execution request is sent only if the following conditions are fulfilled:

$$r_{i,j} = \; isReady \; \wedge notPrev \; \wedge \; notOther \tag{4.4}$$

The first term *isReady* requires the task t_j to be ready, i.e., all predecessor tasks have been finished successfully. The other two terms check the necessity of executing t_j. The term *notPrev* is computed as:

$$notPrev = \bigwedge_{\substack{s_k \in S(t_j) \wedge s_k.p = s_i.p \\ \wedge s_k.f \leq s_i.b}} \neg(u_{k,j} \wedge x_k = 1) \tag{4.5}$$

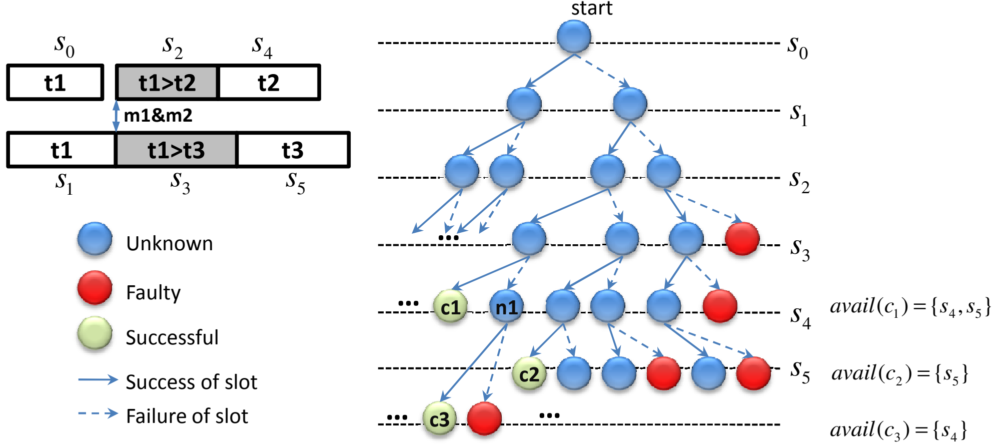


Figure 4.2: An Example of Binary Tree Analysis

It is true if t_j has not been successfully finished on the same processor. The term *notOther* checks if the task has been executed successfully on other processors and a message is scheduled to convey the result to the local processor:

$$notOther = \bigwedge_{\substack{s_k \in S(t_j) \wedge s_k.p \neq s_i.p \\ \wedge s_k.f \leq s_i.b}} \neg((u_{k,j} \wedge x_k = 1) \wedge (\exists m \in \mathcal{M} : m.t_{src} = t_j \wedge m.f \leq s_i.b \wedge m.b \geq s_k.f))$$

The values of variables $r_{i,j}$ and $u_{i,j}$ can be calculated in an iterative manner. Starting from the earliest scheduling slot, we iteratively consider each $s \in \mathcal{S}$. For a specific slot, we compute the variables from the task with highest priority to the task with lowest priority.

As mentioned before, we assume perfect fault detector at this point. Hence, a task is successful if at least one instance of it is executed without faults:

$$success(t_j, \mathbf{x}) = \bigvee_{s_k \in S(t_j)} (u_{k,j} \wedge x_k = 1)$$

For a given schedule, we can construct a function $\varphi_J : \{0,1\}^{|\mathbf{x}|} \rightarrow \{0,1\}$, which takes a fault scenario $\mathbf{x}$ and returns 1 if the job J is still correct under impact of $\mathbf{x}$ and 0 otherwise. Since the entire job is correct only if all of its tasks are correct, the function is given as:

$$\varphi_J(\mathbf{x}) = \bigwedge_{t_j \in \mathcal{T}} success(t_j, \mathbf{x}) \tag{4.6}$$

With the help of function φ, the working set $W(\mathcal{S}, J) = \{\mathbf{x} | \varphi_J(\mathbf{x}) = 1\}$ can be obtained by a Binary Tree Analysis (BTA). The procedure is demonstrated using an example shown

in Figure 4.2. We consider the scheduling slots according to the order of occurrence, i.e., the slots with earlier starting time are selected first (e.g., from S_0 to S_5 in Figure 4.2). Slots with equal start time can be considered in arbitrary order. The ith level in the tree is associated with the ith slot and the edges leaving a node in the ith level represent the execution result of that slot. Left branches (solid lines in Figure 4.2) represent the case that the slot executes some task correctly. Right branches (dashed lines in Figure 4.2) represent a slot with failed execution. Note that a slot might be unused when all tasks in $s.T$ are either not ready or finished earlier. In this case we skip this level and spawn children in the next level (see node n_1 in Figure 4.2). By constructing the tree in this way, each node will have a unique path to the start node representing a unique fault scenario. A node at depth m represents a fault scenario in which the first m variables are determined and the rest are considered to be NA. The total depth D of the tree equals to the number of scheduling slots: $D = |\mathbf{x}| = |\mathcal{S}|$.

Algorithm 1 analysis(n): binary tree analysis with starting node n.

```
//compute request and utilization variable using 4.3 and 4.4
computeRUVariables(n);
l ← createLeftBranch(n)
if checkLeftBranch()=successful then
  addToWorkingSet(l)
else
  analysis(l)
end if
r ← createRightBranch(n)
if checkRightBranch()≠ faulty then
  analysis(r)
end if
```

Algorithm 2 BinaryTreeAnalysis(S): top-level routine of BTA for schedule S

```
setScheduleToBeAnalyzed(S);
n0 ← createStartNode();
analysis(n0);
```

Each node in the tree is associated with its own request/ utilization variables. For a specific node n, we compute those variables using (4.3) to (4.4) based on the values of request/utilization variables associated with the nodes on the path from n to the start node. This procedure actually computes which task is going to be executed in a slot based on the execution history of previous slots.

With the request/utilization variables, a fault scenario $\mathbf{x}$ can be evaluated using (4.6), and the corresponding node is assigned to one of the states: *unknown*, *faulty* or *successful*. A node is *faulty* iff, given the current faults specified in $\mathbf{x}$, there exists no possibility to execute the job successfully in the remaining slots. A node is *successful* iff the entire job is already finished using the successful slots specified in $\mathbf{x}$, i.e., the remaining slots are not needed. The *faulty* and *successful* nodes will not spawn further branches. If a node is neither identified as *faulty* nor *successful*, the analysis continues with its children. The tree analysis is complete if all nodes at the maximum depth D have been visited or no more *unknown* node exists. In the end, the set of *successful* nodes constitute the working set. The analysis process above can be implemented recursively as outlined in Algorithms 1.

The occurrence probability of a *successful* node $\mathbf{x}$ can be computed as:

$$Pr(\mathbf{x}) = \prod_{x_i \in \mathbf{x} \wedge x_i = 1} Pr(s_i) \cdot \prod_{x_i \in \mathbf{x} \wedge x_i = 0} (1 - Pr(s_i)) \tag{4.7}$$

where $Pr(s_i)$ is the success probability of the task executed in slot s_i. The task-level error probability $Pr(s_i)$ are computed using fault model, e.g., the Poisson model described in Section 2.4. With the task-level reliability and the working set, we can obtain the system reliability using equation (4.2).

4.1.1 Complexity and Approximation

The complexity of processing a node during BTA is linear with respect to the number of tasks assigned to the corresponding slot (variables r and u need to be computed for each task). However, this number is typically very small and does not grow significantly when the system becomes more complex. We therefore assume the complexity of visiting a node to be constant. In this case, the complexity of the entire analysis is determined by the number of nodes visited. The worst case scenario occurs when all the nodes in depth smaller than $|\mathcal{S}|$ are in the *unknown* state. The complexity is in $\mathcal{O}(2^{|\mathcal{S}|+1})$ in this case.

As the analysis has a worst case exponential complexity, it is important to find approximations that improve the scalability. An observation from equation (4.7) is that the fault scenarios that specify more faulty slots have much lower occurrence probability, because the failure rate of a task is typically very low. Moreover, a fault scenario that specifies more faults is more likely to be a *faulty* node. Hence, an approximation of the system reliability would be to visit only nodes with at most d faulty slots and to assume all nodes specifying more than d faults to be non-tolerable. Since the reliability is obtained using (4.2), ignoring possibly tolerable nodes

is a safe underestimation of system reliability (see proof below). From the tree point of view, this corresponds to eliminating all nodes with more than d right branches on their paths to the root node.

With the above estimation, the total amount of visited nodes can be computed as follows. We divide the tree into two parts. For the first d levels of the tree, all nodes should be visited, i.e. in total $2^{d+1}-1$ nodes. For the rest, recall that the set of nodes in level l is a complete enumeration of all possible assignments of the first l variables in a fault scenario. Hence, the number of assignments with maximum d zeros is $\sum_{x=0}^{d}\binom{l}{x}$. The total amount of nodes is then:

$$T(|\mathcal{S}|) = 2^{d+1} - 1 + \sum_{l=d}^{|\mathcal{S}|}\sum_{x=0}^{d}\binom{l}{x} \tag{4.8}$$

By applying a simple upper bound for the sum of binomial coefficients $\sum_{x=0}^{d}\binom{l}{x} \leq (l+1)^d$, the complexity of the algorithm computes to:

$$\mathcal{O}(T(|\mathcal{S}|)) = \mathcal{O}(\sum_{l=d}^{|\mathcal{S}|}\sum_{x=0}^{d}\binom{l}{x}) \subseteq \mathcal{O}(\sum_{l=d}^{|\mathcal{S}|}(l+1)^d) \tag{4.9}$$

The expression above can be further overestimated as:

$$\mathcal{O}(T(|\mathcal{S}|)) \subseteq \mathcal{O}(|\mathcal{S}| \cdot (|\mathcal{S}|+1)^d) = \mathcal{O}(|\mathcal{S}|^{d+1}) \tag{4.10}$$

As it can be seen, the complexity of BTA is reduced to be polynomial in $|\mathcal{S}|$ by bounding the maximum number of faults by a constant d. Note that what we analyze here is the worst-case complexity of BTA. During our experiments, we observe that the portion of terminating nodes (mostly *faulty* nodes) increases significantly with higher d and the actual number of visited nodes is much smaller. As an example of runtime, the average execution time of BTA on the *mpeg2* application ($|\mathcal{S}| \approx 35$, measured on a 3GHz CPU) is $754ms$ for $d=3$ and $3405ms$ for $d=5$. Thus the runtime of BTA is acceptable for an offline optimization process.

Correctness Proof of the Approximation in BTA. We prove the correctness of the approximation in BTA by showing that the approximation is pessimistic underestimation of system reliability. The design decisions made based on the BTA result are therefore safe.

Lemma 1. *Eliminating nodes during binary tree analysis is a safe underestimation of system reliability.*

Proof. The system reliability is computed by accumulating the occurrence probability of all nodes in the working set (see equation 4.2). By eliminating a node during BTA, we consider

the node as *faulty* without checking, even if it is possibly *successful*. The occurrence probability of this node will not be added to the system reliability. In this case, the computed system reliability will be less or equal than the true value. Hence, the BTA result is pessimistic and safe. □

Theorem 1. *Visiting only nodes with at most N ($N > 0$) faulty slots during BTA is a safe approximation of system reliability.*

Proof. We prove it by induction. *Start node.* The fault scenario represented by a node is determined by the path from the node itself to the start node. Since the start node has an empty path, it represents a dummy fault scenario that is not considered during BTA. *Nodes in the first level.* For a node in level l, there is exactly l branches along the path to the start node. Since $l \leq N$, the two nodes in the first level are never eliminated by approximation. If they represent tolerable fault scenarios, the according probabilities will be accumulated.

Induction. Assume the BTA is visiting a node A in level L. The number of faults specified by scenario A (denoted by $faults(A)$) must be less than or equal to N, otherwise it is already eliminated. The two child nodes of A is expanded only if A is an *unknown* node, i.e., this branch terminates if A is identified as a *successful* or a *faulty* node (c.f. Algorithm 1). The branch to the left child B specifies a successful execution of the slot associated with the current level. Hence, $faults(B) = faults(A) \leq N$ and B will not be eliminated. No approximation is made at this point. The branch to right child C specifies a new fault occurring at the current level and $faults(C) = faults(A) + 1$. If $faults(C) \leq N$, the BTA continues normally with C and no approximation is performed. If $faults(C) > N$, node C is directly considered as *faulty* and the BTA terminates at this branch even though the children of C could possibly be successful. Nevertheless, eliminating possible *successful* node is a safe underestimation of the system reliability according to lemma 1. Hence, for either of the children of the current node A, the approximation, if it takes place, is safe. □

4.2 Static Priority Slack Sharing for Multiple Jobs

Many multiprocessor systems are designed for co-hosting multiple functionalities concurrently. In particular, there is an increasing trend towards implementing jobs with mixed criticality on a single shared computing platform [96]. It is likely that jobs with different criticality have highly distinct reliability requirements. For highly critical tasks, a significant amount of temporal redundancy is needed to meet their reliability requirements. However, the probability that the software slack is actually used is typically very low. In this case, implementing each job in a step-wise manner without a global view may result in sub-optimal system design (see example below). To cope with this problem, we propose a Static Priority Slack Sharing (SPSS) scheme. The idea is to introduce global re-execution slots and enable sharing of those slots among multiple jobs using a *job-level* static priority approach based on the criticality.

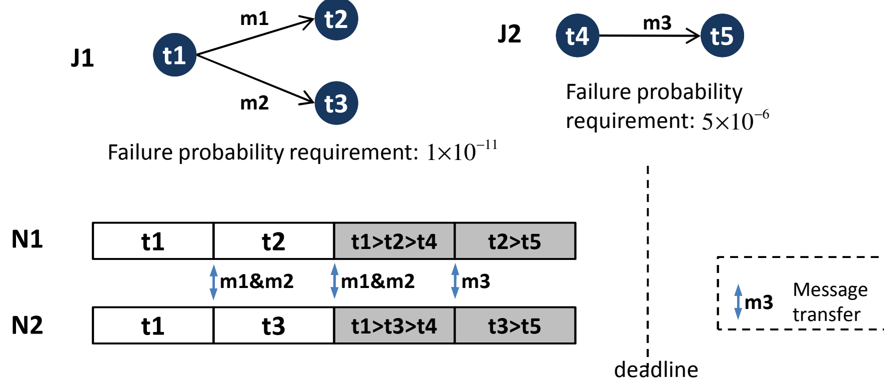


Figure 4.3: Example of Static Priority Slack Sharing

Figure 4.3 shows an example schedule of two jobs using the SPSS technique. A high-criticality job J_1 and a low-criticality job J_2 are allocated on two processors. Four global slack slots are scheduled, in which J_1 is assigned a higher priority. In this case, assigning J_2 in the slack slots has no influence on the execution of J_1. Assume the failure rate of each task is 10^{-5} and the period is $360ms$, we find that the reliability requirement of J_1 is met. For the low priority task J_2, a re-execution slot is granted only if J_1 finishes successfully without using that slot. Due to the fact that the task failure rate is low, such a setup already fulfills the requirement of J_2. Thus, two processors are sufficient to execute both jobs. Without using SPSS, a third processor would be necessary for J_2, since the remaining resources on N_1 and N_2 are not enough.

The BTA is able to analyze a global schedule with multiple jobs and compute the reliability of each job. A minor adaptation is that a node is now considered as *unknown* unless the concrete results (either *successful* or *faulty*) of all jobs are available. As discussed in section 4.1.1, the complexity grows rapidly with the total number of scheduling slots. To cope with this problem, we present an extended algorithm that computes the reliability of each job iteratively, a sketch of which is shown in Algorithm 3.

In the extended algorithm, we iterate over each job with decreasing priority. For a specific job J, we perform the BTA and obtain the set of *successful* nodes (working set). For each node $\widetilde{n} \in W(\mathcal{S}, J)$, there is an *availability scenario* $\widetilde{g}$ associated. It denotes which shared re-execution slots are used and which are not. Let S_G denote the set of shared re-execution slots in schedule $\mathcal{S}$, $\widetilde{g}$ is a subset of S_G that computes to:

$$\widetilde{g} = \{s_i \in S_G | \widetilde{\mathbf{x}}.x_i = NA\} \tag{4.11}$$

Where $\widetilde{\mathbf{x}}.x_i$ refers to the value of variable x_i in fault scenario $\widetilde{\mathbf{x}}$ associated with node $\widetilde{n}$. An example is given in Figure 4.2. The availability is $\{S_4, S_5\}$ for the *successful* node c_1, $\{S_5\}$ for node c_2 and $\{S_4\}$ for node c_3. Note that multiple *successful* nodes may result in the same availability scenario. Hence, the occurrence probability of a specific availability scenario g is:

$$Pr(g) = \sum_{\widetilde{n} \in W(\mathcal{S},J) \wedge \widetilde{g}=g} Pr(\widetilde{\mathbf{x}}) \tag{4.12}$$

For the analysis of next job J', we iterate over each availability scenario (line 4 in Algorithm 3). For a specific availability scenario g, the remaining slack slots are combined with the slots dedicated for J' to obtain the total schedule $\hat{S}$ (line 5 in Algorithm 3). The $\hat{S}$ is then used for the BTA of J' (line 6). In a certain availability scenario g, the occurrence probability of a fault scenario is

$$Pr(\mathbf{x}, g) = Pr(g)Pr(\mathbf{x}|\hat{S}) \tag{4.13}$$

Where $Pr(\mathbf{x}|\hat{S})$ is the occurrence probability of $\mathbf{x}$ using the schedule $\hat{S}$ associated with g. The probabilities of tolerable fault scenarios found with each availability scenario are summarized using equation (4.2) to obtain the system reliability. The BTA of J' computes again the availability scenario for further jobs (line 7 and 9 in Algorithm 3).

Algorithm 3 IterativeTreeAnalysis(): iterative tree analysis for multiple tasks. AS_{old}: the set of availability scenarios from previous job. AS_{new}: the set of availability scenarios for next job. $S(J)$: the set of slots dedicated for job J. The function *combine* computes the overall occurrence probabilities of availability scenarios using (4.12).

```
1: AS_old ← initAvailability();
2: AS_new ← initAvailability();
3: for all J ∈ A with decreasing priority do
4:     for all a ∈ AS_old do
5:         Ŝ = S(J) ∪ a
6:         avail ← BinaryTreeAnalysis(Ŝ)
7:         combine(AS_new,avail)
8:     end for
9:     AS_old ← AS_new
10: end for
```

Complexity. Let $|\mathcal{A}|$ be the number of jobs and $S(J)$ be the set of scheduling slots dedicated to job J, the total number of slots of schedule $\mathcal{S}$ can be represented as $|\mathcal{S}| = \sum_{J \in \mathcal{A}} |S(J)| + |S_G|$. Consider the case that we assume maximum d faults in each job, the maximum number of faults in the entire system is $|\mathcal{A}|\, d$ and complexity of the analysis is in $\mathcal{O}(|\mathcal{S}|^{|\mathcal{A}|d+1})$ according to

equation 4.10. Using the iterative approach, the worst-case complexity of BTA for a single job J is in $\mathcal{O}((|S(J)|+|S_G|)^{d+1})$. The BTA needs to be done for each availability scenario. Assume J is the xth job that we consider, then the previous jobs may encounter up to $(x-1)d$ faults. Those faults may consume shared slack slots and thus result in different availability scenarios. In the worst case, each combination of faults has a different availability scenario, so the total number of BTAs to be performed is $\sum_{i=0}^{(x-1)d} \binom{|S_G|}{i} \leq (|S_G|+1)^{(x-1)d}$. It can be easily seen that the complexity is significantly reduced. For further complexity reduction, we can apply the same idea as in section 4.1.1 on availability scenarios by considering the availability scenarios with occurrence probabilities lower than a threshold value as faulty. This is obviously also a safe underestimation of reliability.

4.3 Optimization Procedure

Guided by analysis results, we considered how to find the optimal task schedule. We adopt the Multi-Objective Evolutionary Algorithm (MOEA) as the optimization engine. To use MOEA, the candidate solutions must be encoded into a special data structure called *chromosome*. The set of chromosome maintained by the optimizer is called the population. In each iteration, the optimizer selects a subset of solutions from the population and uses them to produce offspring (new solutions). This procedure is done by applying crossover and/or mutation operators. The new solutions are evaluated by fitness functions and high quality solutions will replace low quality ones in the population. This process repeats until a candidate with sufficient quality is found or a maximum number of iterations is reached. Figure 4.4 shows an overview of MOEA.

To describe *TT-SP* or *TT-FS* schedules, the information about each slot is needed, including the start/finish time and the set of tasks allocated. A direct encoding of such schedules generates very large chromosome, resulting in a huge search space and low optimization efficiency. To cope with this problem, we utilize a two-step encoding process inspired from [97]. The main idea is, instead of encoding the complete schedule, we only put partial information, namely the mapping and FTM configuration, into the chromosome. A scheduler is used to rebuild the schedule from the chromosome, which is then used for fitness evaluation, e.g., reliability analysis. For the rest of this section, we present the encoding technique for *TT-SP* schedules. The section 4.3.2 discusses necessary changes to support *TT-FS* schedules.

Using this approach, the chromosome contains one gene per task. Each gene is a pair $g = (i, L)$, where i is the integer index of the task and L is a list of integer values denoting

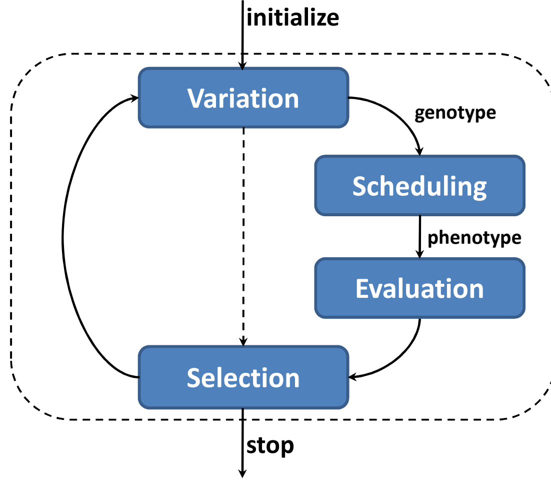


Figure 4.4: Workflow of EA-based Optimization

the set of PEs task i is mapped to. An example is shown in Figure 4.5. Multiple mappings of the same task onto the same PE are interpreted as temporal redundancy (task 2 and 3 in Figure 4.5); multiple mappings of the same task onto different PEs are interpreted as spatial redundancy (task 1 in Figure 4.5).

4.3.1 Schedule Reconstruction

Reconstruction of the schedule from the chromosome is the same problem as scheduling the tasks with known mapping and FTM configuration. The reconstructed schedule is sent to the BTA to evaluate the reliability for current solution. The selection of scheduler is a user decision and has no influence on the correctness of analysis. For example, the user may implement a scheduler that only generates strict schedules or another one that also generates general schedules. The BTA is generic and supports both types.

The scheduling procedure that we propose consists of three main steps. First, for each mapping entry of a task t, we instantiate a scheduling slot with length equal to the execution time of t. The set of slots is scheduled using a list scheduler. The priority is computed based on two criteria: 1) a task belonging to a job with earlier deadline has higher priority (job-level EDF); 2) for tasks in the same job, the one that has a longer critical path to the sink is assigned a higher priority. Using such an approach, data dependencies are automatically regarded. Second, bus scheduling is performed for each message (Figure 4.5b). In this paper, we adopt the *transparent recovery* approach [81], which requires that a fault occurring on one PE is masked to other PEs. This approach has several advantages such as fault-containment

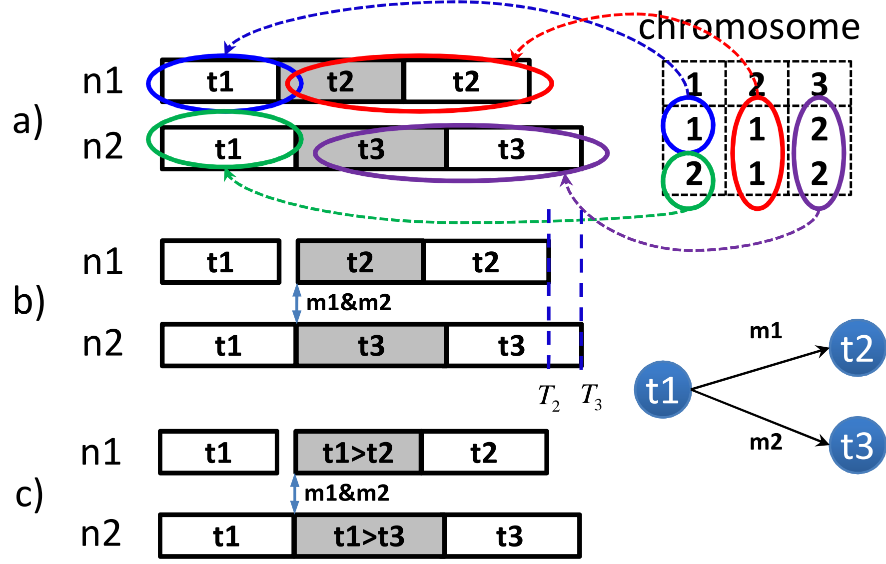


Figure 4.5: Encoding and Reconstruction of Schedule

and improved traceability. According to transparent recovery, the message should be scheduled after possible re-executions so that faults occurring at the sender are not visible to the receiver, e.g., if the task t_2 in Figure 4.5 sends a message to other tasks, the message should be placed at time T_2. Tasks may be postponed due to dependency on messages. In the last step, we perform slack sharing (Figure 4.5c) using a greedy approach. A slot is shared with all tasks that 1) may become ready before the start time of this slot; 2) has an execution time no greater than the slot size.

An advantage of the two-step encoding is that many application specific constraints can be easily translated into rules on the chromosome. The MOEA can be customized to generate only chromosome that fulfill these rules. For example, safety-critical applications often have separation constraints, such as task t_1 and t_2 must be strictly isolated in space. We could guarantee this constraint by making sure that the mapping entries of the two tasks are mutually exclusive.

4.3.2 Encoding of *TT-FS* Schedules

The encoding scheme needs slight modifications to handle *TT-FS* schedules. We use the integer 0 to denote slack slots and integers larger than 0 to denote regular slots. Figure 4.6 shows an example. Two tasks t_1 and t_2 are allocated on processor 1 and one slack slot is scheduled. To reconstruct the schedule from chromosome, the same list scheduler described in section 4.3.1

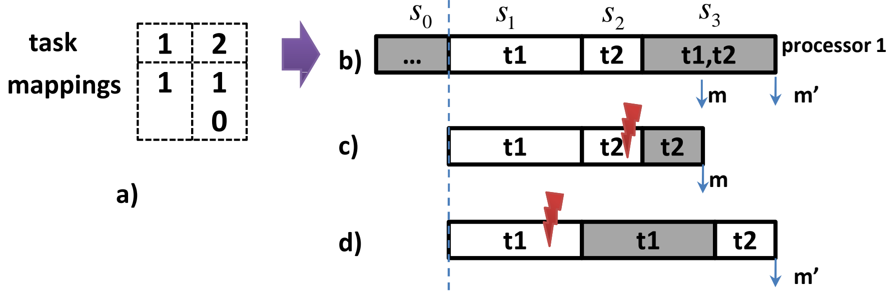


Figure 4.6: Example of Message Placement for *TT-FS* Schedules

can be used to generate an initial schedule. The slack slots are placed right after the regular slots of the same task, e.g., S_3 is located directly after S_2. We introduce a greedy slack sharing approach that works as follows. First, the schedule is divided into several segments. The segments are separated by one or several consecutive slack slots. Then each slack slot is allowed to be shared by all tasks in the segment just before itself. As an example, the regular slot S_1 and S_2 in Figure 4.6 belong to the segment separated by slack slots S_0 and S_3. Hence S_3 is shared by task t_1 and t_2. The size of slack slots is set to the largest execution time of all tasks sharing the slot.

For the *TT-FS* Schedules, special care is needed in the message placement step. This is because normal slots might be delayed due to out-of-order execution of slack slots. To achieve transparent recovery, the faults occurred on one processor must be masked to other processors even if the task is delayed. This can be explained using an example. Assume t_2 in Figure 4.6b is going to send some message to other processors and we want to mask a single fault that occurs on processor 1. Figure 4.6c shows the execution scenario that t_2 encounters a fault. Based on the idea of transparent recovery, the message m originated from t_2 should be placed no sooner than the second instance of t_2. However, the message should be scheduled at an even later time (m' in the figure), since the worst-case scenario happens when t_1 encounters a fault as shown in Figure 4.6d. In other words, if the message is placed at m, a single fault on t_1 cannot be tolerated and the system reliability decreases. Thus, our scheduler always analyzes the worst-case scenario and places the messages accordingly.

4.3.3 Crossover and Mutation

In order to improve the performance of the MOEA, we implement some crossover and mutation operators that add problem-specific knowledge to the optimizer [98]. We present those

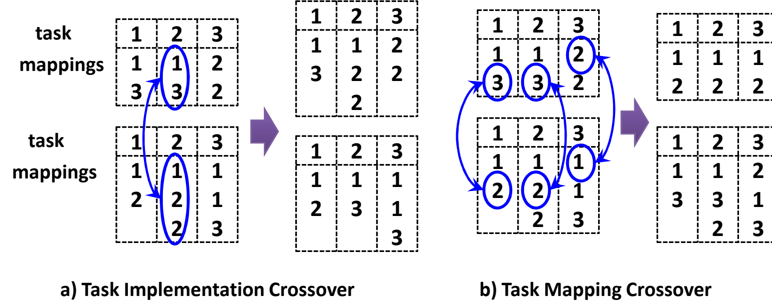


Figure 4.7: Task Implementation Crossover Example

operators in the following:

Task Implementation Crossover: This operator randomly selects a set of tasks and swaps the entire implementation of these tasks across two chromosome, including the amount of spatial/temporal redundancy and mapping. The rest of the chromosome remains unchanged. Figure 4.7a shows an example in which task 2 is selected for crossover.

Task Mapping Crossover: This operator performs crossover on the implementation of a specific task. Given two chromosome, the mapping entries for the selected task are randomly swapped. Figure 4.7b shows an example in which 3 mapping entries are swapped.

Increment Redundancy: This mutation operator inserts a new mapping entry for a randomly selected task. Insertion of the new mapping x to task t might result in: 1) a temporal replica, if the chromosome already contains a mapping of t to x, or 2) a spatial replica, if t has not been mapped to x.

Decrement Redundancy: The counterpart of *Increment Redundancy*, removes one mapping entry from a random task. At least one mapping entry must remain for each task.

Re-Mapping: This mutation operator randomly changes the mapping entries. The result might be: 1) re-mapping of the tasks to other PEs or 2) transformation of a temporal replica to a spatial replica or vice versa.

4.4 Tolerating Permanent Faults using Virtual Mapping

Safety-related systems must tolerate both permanent and transient faults. In most existing work, they are considered separately using dedicated techniques. However, it is particularly important to consider both types of faults in a **unified** manner, in order to achieve the most efficient and reliable design. We explain this point using the following example. Consider the system in Figure 4.8 consisting of two jobs J_1 and J_2 to be executed on two processors p_1 and p_2.

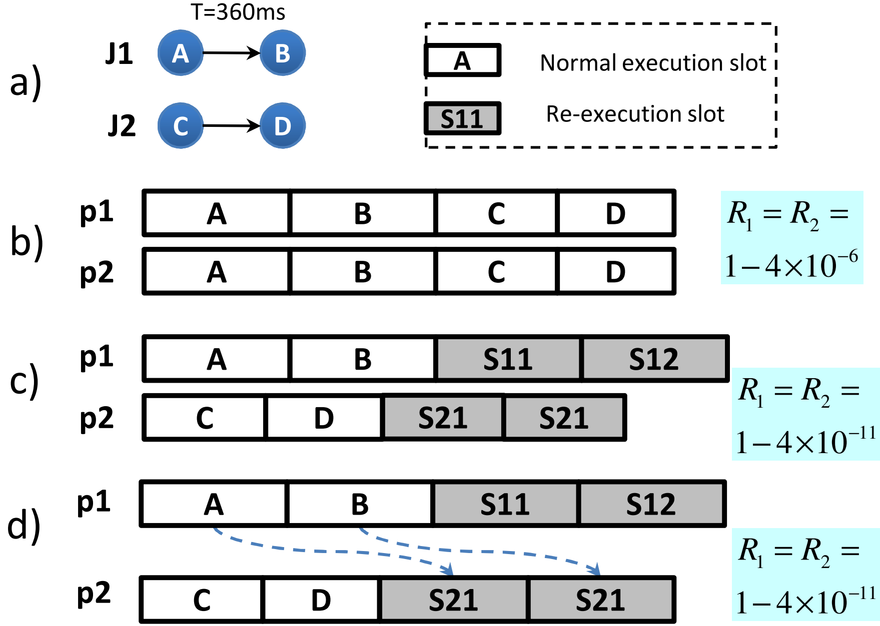


Figure 4.8: Motivating Example for Considering Transient and Permanent Faults Together

It has the requirement to tolerate a single defect on any of the two processors. A straightforward design considering only the permanent faults could be done using pure hardware redundancy as shown in Figure 4.8b. However, such a setup has very limited capability of tolerating transient faults. Assume transient fault probability is 1×10^{-5} for each task and the period of both jobs are $360ms$, the failure rate per hour of both J_1 and J_2 can be easily computed as 4×10^{-6}. Using the same amount of resources, the software re-execution technique can achieve much better tolerance to transient faults. In Figure 4.8c, two re-execution slots (or slack slots) are scheduled on each processor, which can be used to re-execute any previous task that is misbehaving due to transient faults. Using the analysis presented later in section 4.1, the failure probability of both applications is computed to 4×10^{-11}. However, the schedule in Figure 4.8c is not capable of tolerating permanent faults on p_1, since the slack slots S_{21} and S_{22} are not large enough to accommodate A and B. Actually, when we schedule the re-execution slots to tolerate transient faults, we can keep the requirement on permanent faults in mind and intentionally increase the sizes of slot S_{21} and S_{22} to fit task A and B (Figure 4.8d). In this way, permanent defects can also be handled since migration of task A and B is now possible. The schedule 4.8d therefore has the same tolerance to permanent faults as schedule 4.8b and achieves much higher tolerance to transient faults.

The analysis and optimization approach presented so far focuses only on transient faults.

In this section, we present an extension to add the consideration of permanent faults. Recall that, to tolerate a permanent defect of some processor p, we need to guarantee that each task mapped to p either has another running instance (spatial replication) or can be migrated to a slack slot on other processors. Thus, a straitforward way is to ensure that each task has at least one replica by adding constraints on the chromosome. However, spatial redundancy comes with high hardware cost and is less efficient to tolerate transient faults.

A more cost-efficient alternative to handle permanent faults is task migration. To design such a system, one of the most important goals is to minimize the overhead of migration. The ideal case is that the system recovers from faults with only minor re-configuration. Since attaining the optimal task migration decision is a highly complex task, recent work [9] proposes to compute the task re-mappings statically offline and store them in tables. The pre-computed configurations are applied at runtime if a permanent fault is detected. We adopt a similar approach and synthesize static schedules that can be adapted with minor changes to handle failure of processors. For static time-triggered scheduling that we are focusing on, the migration cost is highly influenced by the data dependencies. Consider the example depicted in Figure 4.10, where the task X is to be migrated to one of the possible locations S_1 to S_3. The tasks A and B are communicating with X via messages. If X is re-mapped to S_1, which is earlier than the original message M_1, the predecessor task A and the message M_1 need to be shifted forward due to data dependency. In consequence, other tasks communicating with A need further adaptation and overall migration cost could be much higher. A similar situation occurs if X is migrated to S_3, which is later than message M_2. In this case the successor tasks need to be shifted backwards. Instead, if X is moved to S_2, the rest of the schedule does not need to change and the migration can be performed with low overhead. An indication from this example is that, while building schedules to tolerate transient faults, we should take permanent faults into consideration and try to make such low-overhead migrations feasible. For the given example, we could try to schedule a slack slot between t_1 and t_2.

We propose a virtual mapping technique for this purpose. The general idea is to trace potential locations for task migrations already at the time when the schedule is constructed from the chromosome. A virtual mapping of task t to p is represented in our encoding scheme using a negative integer $-p$, which implies that p is the target of migration of task t. For example, the chromosome shown in Figure 4.9a specifies two entries 1 and -2 for task A, which means A is executed on processor p_1 during normal execution and it is to be migrated to p_2 if p_1 fails. When constructing the schedule, we instantiate a slot also for a virtual mapping, with

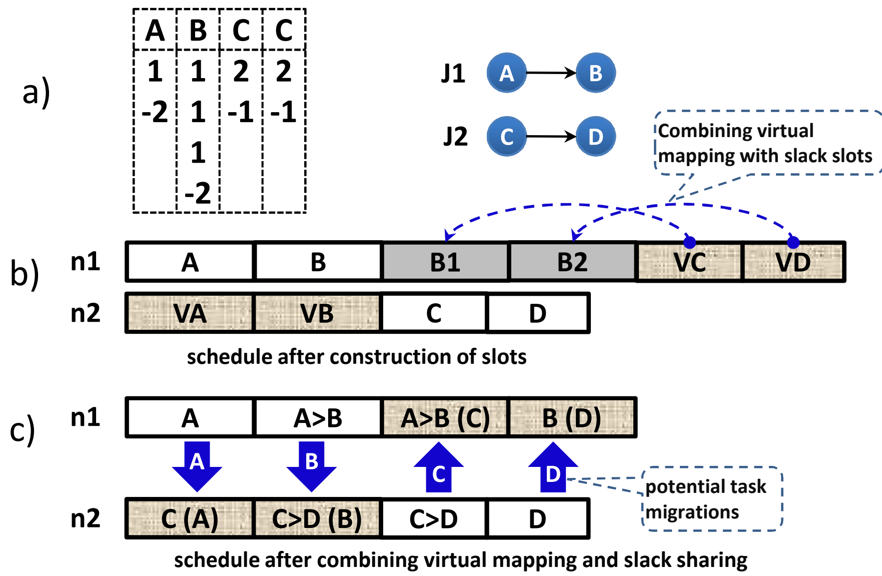


Figure 4.9: Example of Virtual Mapping

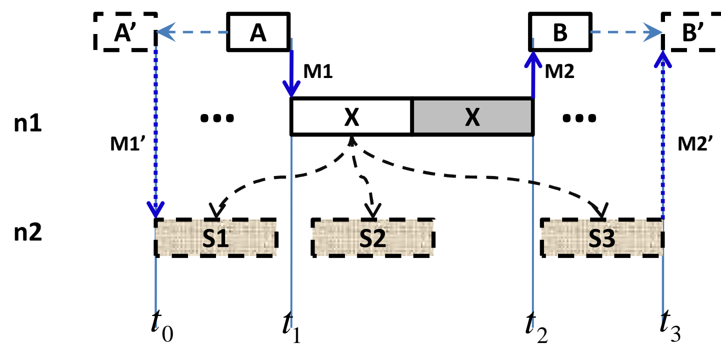


Figure 4.10: Influence of Data Dependency on Task Migration

the size of slot being the execution time of A (slot VA in Figure 4.9b). This slot is the location where A will be migrated to. Note that the virtual mapping slots are also scheduled using the same heuristic presented in Section 4.3 so that data dependencies are also regarded. This is essential to achieve a low-overhead task migration as shown in Figure 4.10. Nevertheless, during normal execution, this slot is not left empty but used as a slack slot for other tasks mapped onto the same processor. For example, in Figure 4.9c, the slot VA is actually used for task C. This technique reclaims the time reserved for task migration and uses it to improve the transient fault tolerance in normal execution. The efficiency of resource utilization is therefore improved. Note that virtual mapping slots may be combined with other slack slots scheduled on the same processor to reduce the length of the schedule. For example the slot VC is combined with B_1 and slot VD is combined with B_2. The combination is only possible if two rules are obeyed: 1) the normal slack slot is no smaller than the virtual mapping slot; 2) no data dependency is violated. These two rules guarantee that the task migration is still valid after combination. Afterwards, the corresponding slack slots are marked as new migration targets (Figure 4.9c).

Note that we assume a use scenario that task migration is only considered as an emergency response for permanent faults. The goal is to guarantees continuous service of the system, possibly with degraded quality due to lack of resources, before a maintenance (e.g., replace the failed hardware) can be carried out. In this case, we do not consider the reliability concerning transient faults after the migration.

There are two main advantages of using virtual mapping. The first is easy implementation, since the optimization process remains unchanged and no further objective is necessary. Tolerance of permanent faults is achieved by adding simple constraints to the chromosome. For example, if it is required to tolerate a defect of processor p, we just need to add the constraint that tasks that are mapped only to p must have a virtual mapping. The second advantage is low migration overhead. Using the proposed approach, the locations for task migration are statically computed and the required resources are allocated. To carry out the task migration, the scheduling slots do not need to change. Only a simple update of the priority table of virtual mapping slots needs to be done, e.g., task A can already be mapped to VA set to lowest priority to allow other tasks to acquire the slot during normal execution. When migration is needed, we just set A to the highest priority in the slot. Since the binary of task A is already loaded to the target processor, timely recovery can be achieved.

4.5 Simulation Results

We implement the analysis and optimization algorithms in JAVA using the opt4j library [99]. We assume that the target platform consists of two types of PEs, namely a RISC processor and a DSP. The failure probability of each task on a certain PE is randomly generated between 1×10^{-5} and 1×10^{-7} (the failure probabilities are in the typical value range for soft error rates [3]). We restrict each task to have at most 2 spatial replicas and 2 temporal replicas. For the metric of reliability, we use the System Failure Probability (SFP) per hour in logarithmic scale in the experiments, i.e., the lower the value, the higher the reliability is. We use two sets of Task Graphs (TGs) as the benchmark. The first is a set of random TGs with 5 to 15 nodes generated synthetically using TGFF[1]. The execution time of each task on the RISC/DSP is generated randomly between 100 and 1000. The second is an *mpeg2* decoder example from [95] that consists of 13 tasks.

The goal of the first set of experiments is to evaluate the accuracy and runtime of the reliability analysis. Several instances of the BTA with different approximation factors are evaluated. We use the approximation technique introduced in section 4.1.1, which bounds the maximum number of faults (MF) considered in BTA. Figure 4.11 presents the results averaged over 100 test runs, each round with a random TG and a random schedule. As it can be seen, the execution time increases rapidly with larger *MF*. The run time of BTA with $MF = 5$ is around $16x$ higher than the case with $MF = 3$. For the accuracy, all analysis with MF larger than 1 bounds the average relative error to less than 10%. The BTA with $MF = 3$ achieves a very good tradeoff between runtime (around 3 seconds) and accuracy (99.3%) and is considered as a good option in practice. Actually, the BTA should be used in most cases with relatively small MF. Since the occurrence probabilities of transient faults are typically very low (e.g., at the magnitude of 10^{-5} [100]), fault scenarios with a large number of faults happen very rarely. The scheduler should focus on covering all the fault scenarios with high probabilities instead of tolerating rare cases. For example, if a single-fault scenario with probability 10^{-5} is not tolerable, it becomes the system bottleneck. Even if all other fault scenarios can be tolerated, the maximum achievable reliability is limited to $1 - 10^{-5}$.

Note that using a lower MF value results in more pessimistic estimation of system reliability, since a larger number of nodes are automatically considered as non-tolerable (see Section 4.1). The solution found using a coarse analysis is safe due to pessimism. However, it may happen

[1] TGFF `http://ziyang.eecs.umich.edu/~dickrp/tgff/`

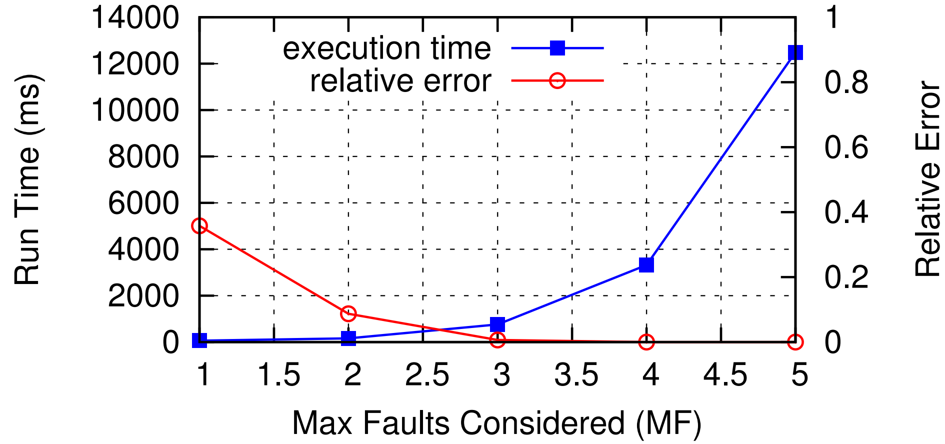


Figure 4.11: Evaluation of BTA with Approximation

that no feasible solution is found even if some exists. The approximation factor should be selected in a way that the accuracy of analysis is sufficient to reach the same accuracy of applications' reliability goal. The accuracy of the analysis can be roughly estimated as follows. Assume the transient fault probability of tasks are at the magnitude of 10^{-5}, the occurrence probabilities of single-fault scenarios are at the magnitude of 10^{-5} and the probability of 2-fault scenarios are at the magnitude of 10^{-10}. If we consider $MF = 2$, the analysis will drop all fault scenario with more than 2 faults, and the reliability is determined by the portion of tolerable single-fault and 2-fault scenarios. Since the BTA accumulates the occurrence probabilities of tolerable scenarios to obtain the system-level reliability, the accuracy of analysis is also at 10^{-10}. Hence, if the reliability goal is at a higher accuracy, e.g., 10^{-11}, a high MF should be considered. Also, using different approximation factors in the design process might be helpful. E.g., the coarse analysis can be used to obtain some fast results and more accurate analysis can be used for final decision making and verification.

4.5.1 Architecture Exploration Case Study

A challenging task in embedded system design is architecture exploration. The designer has to address problems such as what is the amount and the type of PEs needed to meet all application requirements. To illustrate how this can be supported by our approach, we consider the *mpeg2* application and run the DSE engine with several platform configurations consisting of 2 to 6 PEs. The execution time of tasks is specified according to [95]. The deadline of the application is set to two times the critical path of the TG to allow some slack for reliability improvement. Only

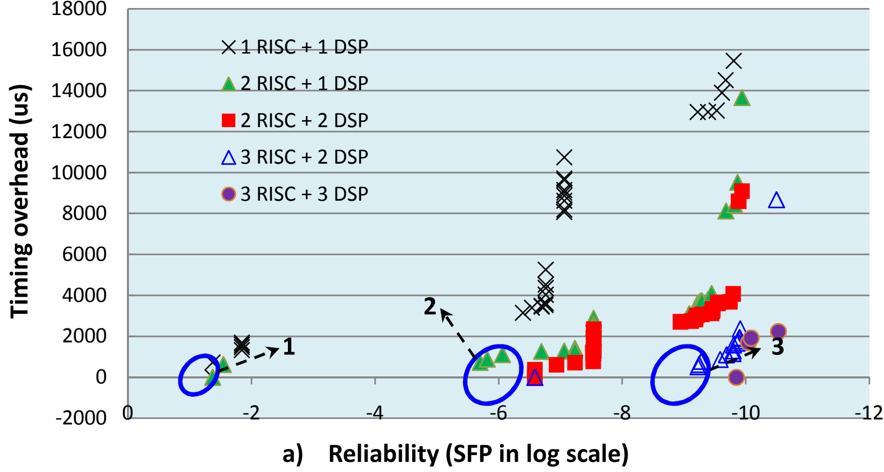


Figure 4.12: Pareto Optimal Solutions under Different Platform Configurations

transient faults are considered for the moment. The MOEA is configured with two objectives. The first one is timing overhead. It is defined as:

$$penalty(S) = \begin{cases} -1 & \text{iff } l \leq d \\ l - d & \text{otherwise} \end{cases} \tag{4.14}$$

where l is the finish time of the job in schedule S and d is the deadline. The idea is that, if the deadline is met, we set the penalty to a constant -1 and if not, we set the penalty to the difference between the finish time and the deadline. In this way the optimizer will prefer solutions that meet the timing constraints and optimize other objectives. The second objective is reliability using the SFP as a metric. Figure 4.12 shows the Pareto optimal solutions found by the optimization. It can be seen that the Pareto fronts obtained with more PEs dominate those obtained with less PEs in most cases, i.e., with more hardware resources, the application can be finished with shorter time and higher reliability. This is due to the increased opportunity for spatial redundancy.

For each platform, we are interested in the solution that achieves maximum reliability while meeting the deadline. These solutions are marked with 1 to 3 in Figure 4.12. As it can be seen, the $2RISC{+}2DSP$ platform is the minimal one to achieve SFP of 10^{-6} and the $3RISC{+}3DSP$ platform is necessary to achieve SFP of 10^{-9}. An important observation from Figure 4.12 is that, for the $2RISC + 1DSP$ platform, several solutions with SFP around 10^{-6} are very close to meeting the deadline. The same is observed for the platform $3RISC + 2DSP$, where several solutions are close to achieving SFP of 10^{-9}. This implies that, these two platforms might already be sufficient to reach respectively 10^{-6} and 10^{-9} if they can be made a bit faster. We

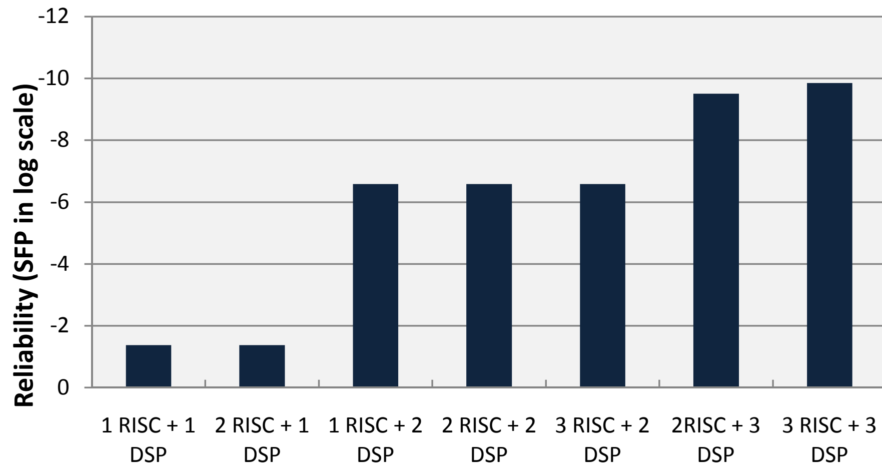


Figure 4.13: Achievable Reliability Comparison

therefore test two additional platforms with *1 RISC + 2 DSP* and *2 RISC + 3 DSP* (the DSP is faster for the *mpeg2* application). Figure 4.13 shows the best solution under deadline constraint for each platform. Clearly, the new platforms with *1 RISC + 2 DSP* and *2 RISC + 3 DSP* are the most cost-efficient solutions to achieve SFP of 10^{-6} and 10^{-9}, respectively.

4.5.2 Comparison of *TT-SP* and *TT-FS*

Qualitative Comparison. The *TT-SP* and *TT-FS* schemes can be compared in several aspects:

- Resource efficiency. Both *TT-SP* and *TT-FS* allow sharing of time slots by multiple tasks and are therefore more efficient than traditional techniques with dedicated redundancy for each task, e.g., [24]. For example, the schedule in Figure 2.6a and 2.6e are able to tolerate a single fault of any of the tasks t_1, t_2, t_3. Without using the slot-sharing technique, we would have to replicate all three tasks once, in order to achieve the same level of fault-tolerance. However, much more resources are needed in this case. If we compare *TT-SP* and *TT-FS*, the later achieves generally higher resource efficiency. This is because only the slack slots are shared and must have their sizes set to the largest WCET of all tasks assigned to them. Again in same example, the schedule in Figure 2.6a has the length $|t_1| + max(|t_1| + |t_2|) + max(|t_2| + |t_3|) + |t_3|$, whereas the schedule in Figure 2.6e has the length $|t_2| + |t_2| + |t_3| + max(|t_1| + |t_2| + |t_3|)$. It can be easily verified that the length of 2.6e is no larger than that of 2.6a. Nevertheless, the *TT-SP* scheme can also outperform *TT-FS* in certain circumstances (see the discussion of Figure 4.15).

- Predictability. The *TT-SP* scheme exhibits higher predictability in sense the start time of all slots are fixed and known. In Figure 2.6, the scheduling points, where the runtime scheduler has to be called, are marked. As it can be seen, the scheduling points of *TT-SP* are fixed in time without dependency on the faults occurred. In contrast, the scheduling points vary for different fault scenarios for *TT-FS* schedules.

- Scheduler Complexity and Overhead. The implementation of a *TT-SP* scheduler is straightforward. At each scheduling point, we just pick the pending task that has the highest priority for execution. The implementation is more complicated for *TT-FS*. Depending on the faults that occurred previously, the schedule has to be adapted at runtime and the new scheduling points need to be determined. In Figure 2.6f for example, the slot S_5 and S_6 have to be delayed due to the re-execution of t_1 using S_7. In this case, the complexity in scheduler implementation is higher, resulting in generally higher scheduling overhead.

Experimental Comparison. To evaluate the performance of *TT-SP* and *TT-FS*, we extend the optimizer with an additional optimization objective, namely resource consumption. We vary the reliability goal of the *mpeg2* application from SFP 10^{-5} up to SFP 10^{-9} and check the minimum amount of resources to achieve the desired reliability level. The timing constraints remain the same. The $2RISC + 2DSP$ platform is considered as the target architecture. For the fitness of timing and reliability, the same technique as in equation 4.14 is applied, i.e., the penalty is set to -1 if the timing/reliability requirements are fulfilled and a positive value otherwise. The resource utilization is the total processor time a schedule occupies. Clearly, all objectives need to be minimized.

Three approaches are compared, namely the *TT-SP* scheme, the *TT-FS* scheme and the traditional approach without slack sharing (*NoSharing*). *NoSharing* is a similar approach as the existing work [24]. However, the results are obtained using our optimization framework with three objectives instead of the bicriteria heuristic proposed in [24]. Figure 4.14 compares the minimum resources needed to meet both timing and reliability requirements. Clearly, *TT-SP* and *TT-FS* out-perform *NoSharing* significantly by allowing temporal redundancy to be shared by multiple tasks. As the reliability requirement becomes higher, more redundancy needs to be added and the benefit of slack sharing also increases. When the SFP goal is 10^{-8}, 25.7% more resources are comsumed by *NoSharing*. Moreover, *NoSharing* fails to provide any feasible

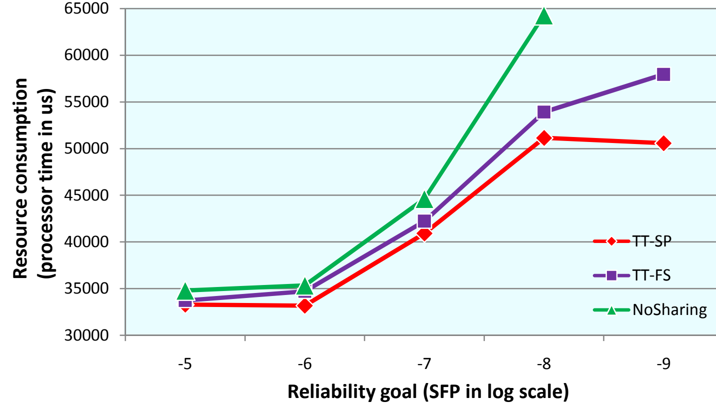


Figure 4.14: Comparing *TT-SP* and *TT-FS* using *mpeg2*

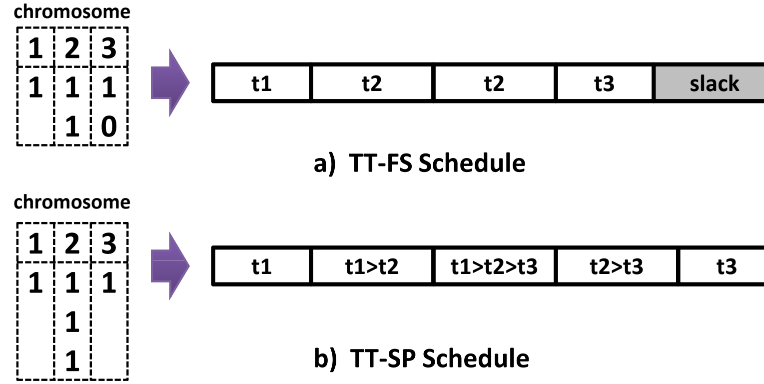


Figure 4.15: An Example for Comparing *TT-SP* and *TT-FS*

solution [1] when the SFP goal is set to 10^{-9}. What is a bit unexpected is that *TT-SP* exhibits better performance than *TT-FS* for the *mpeg2* application. After a detailed analysis of the schedules, we find out that this is because the tasks of *mpeg2* have a large variation on failure probabilities. This implies that more replicas should be scheduled for tasks with high failure rates. However, the slack slots in *TT-FS* implicitly treat all tasks in the same way. We explain this issue using a simple example depicted in Figure 4.15.

Consider that three tasks t_1 to t_3 are allocated on processor 1 and task t_2 has a higher failure rate. *TT-FS* allocates one shared slack slot for all three tasks and an extra replica for t_2 (Figure 4.15a). In this way, the schedule tolerates any single fault on t_1 to t_3 and also two consecutive faults on t_2. Since t_2 has the largest execution time, the size of the slack slot is set to $|t_2|$. The overall length of schedule is then $|t_1| + 3 * |t_2| + |t_3|$. Figure 4.15b shows the optimal *TT-SP* schedule that utilizes the same amount of resources. Three replicas are allocated for t_2

[1] When the approach fails to find a solution, the corresponding point is missing in the figure.

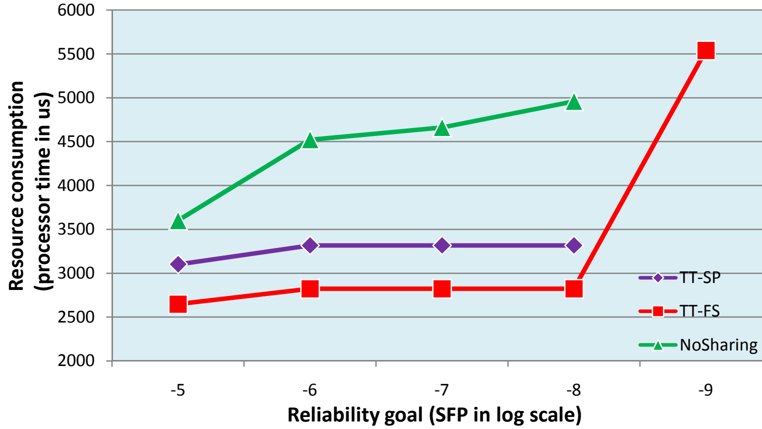


Figure 4.16: Comparing *TT-SP* and *TT-FS* using Random TG

and then shared with t_1 and t_3. Thanks to the sharing of three slots, the schedule in Figure 4.15b tolerates a larger set of fault scenarios (actually, it tolerates two faults on any of the tasks) and therefore achieves higher system-level reliability. In other word, the *TT-SP* schedule shows better performance in this scenario.

We did the same experiment on a randomly generated Task Graph (TG), whose tasks have small variation on failure rates. Figure 4.16 summarizes the results. As it can be seen, the *TT-FS* approach consumes less resources to reach the same reliability level. For this example, the resource saving archived by slack sharing is larger than the case of *mpeg2*, e.g., 75.7% for SFP 10^{-8}. This is because the deadline of the TG is relatively loose and the possibility to use re-execution slots is higher.

4.5.3 The Case with Permanent Faults

In the next step, the consideration of permanent faults is added and two approaches are compared:

- The step-wise approach in which permanent faults are handled first using spatial replications and then, on top of that, transient faults are handled using temporal and spatial redundancy.
- The proposed unified approach, in which permanent faults and transient faults are considered together using the virtual mapping technique.

As a reference, we also compare them with a case in which only requirements on transient fault tolerance are added (No-PF). We are interested in how much overhead is needed to fulfill

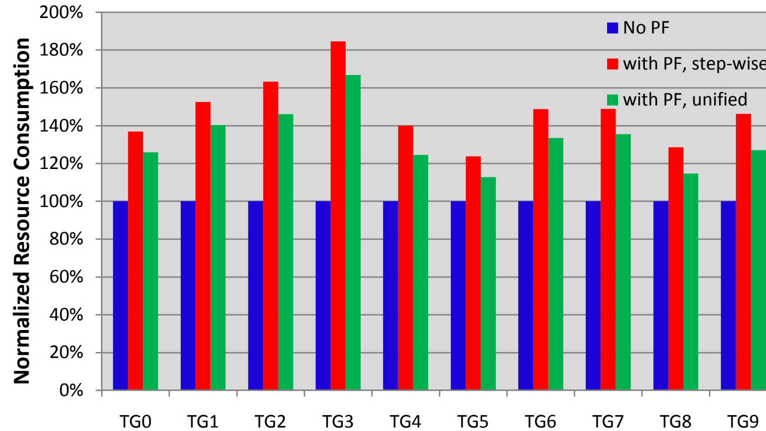


Figure 4.17: Performance Comparison of Step-wise and Unified Approaches

the additional requirements on permanent faults. We again use three optimization objectives, namely schedule length, reliability and resource utilization. We assume that it is required to tolerate a single defect on any of the processors. Figure 4.17 compares the solution that meets both timing and reliability requirements with minimum resources tested on 10 random TGs. The resource consumption is normalized with respect to the reference (No-PF). For the step-wise approach, 47% more resources are needed on average to handle the permanent faults. The unified approach reduces the resource overhead to 33%, i.e., 14% resource saving is achieved. Figure 4.18 gives a closer view of the Pareto optimal results for one example TG. As it can be seen, the solutions found using the unified approach dominate those found by the step-wise approach. One observation is that, some jobs such as $TG3$, need more resources to tolerate permanent faults than other jobs. The reason is, these jobs exhibit limited parallelism and the optimizer tends to schedule a large part of the job onto the same processor, so that transient faults can be handled efficiently using temporal redundancy. In this case, a large part of the job needs to be replicated/migrated if a defect occurs. As the opposite case, the *mpeg2* application is easy parallelizable and has a relatively tight deadline, which guides the optimizer to a distributed implementation even if permanent faults are not considered. In this case, the additional resources needed are marginal (2% using the unified approach), since only some minor modifications are needed to guarantee feasibility of task migrations.

4.5.4 Comparing Slack Sharing Schemes

We proceed with experiments with multiple jobs running concurrently. The focus of this set of experiments is on evaluation of the slack sharing schemes. We compare three configurations:

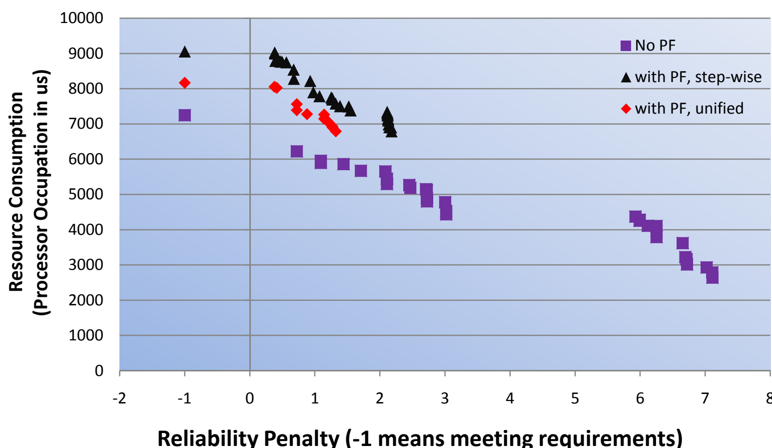


Figure 4.18: Example of Pareto Optimal Results

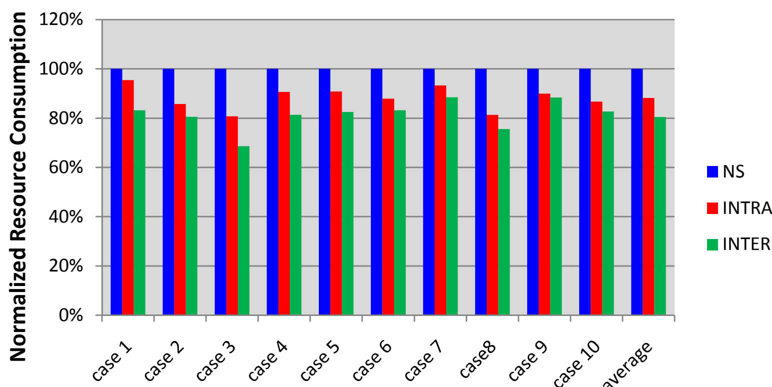


Figure 4.19: Comparison of Slack Sharing Schemes

in the first one, no slack sharing is enabled (NSS), i.e. each task has its dedicated replicas and slack slots; in the second one, intra-job slack sharing is used (INTRA), i.e. job-level global slack slots are scheduled and shared amongst all tasks belonging to the same job; and in the third configuration, the proposed SPSS scheme is used (INTER), i.e. global slack slots are shared by all jobs using a static priority based approach. We generate 10 random applications with 2 to 3 jobs running concurrently. Figure 4.19 compares the solution that fulfills deadline and reliability requirements of all jobs with minimum resource consumption. The resource consumption is normalized with respect to the NSS approach. As it can be seen, significant resource saving can be achieved using slack sharing. On average, $INTRA$ and $INTER$ save 12% and 20% resources, respectively.

Chapter 5

Fault Tolerant System Design using Imperfect Fault Detectors

The reliability analysis techniques presented so far are based on a common assumption that all transient faults are detected by embedded fault detectors. However, fault detection is often imperfect in reality. Certain faults may escape the detector and propagate to subsequent tasks. Hence, the perfect fault detection assumption may cause several practical concerns. In this thesis, we put special emphasis on developing an approach that supports imperfect fault detection.

This chapter starts with introducing extended system models to consider imperfect fault detection. Afterwards, we motivate our approach using an example. Section 5.4 and 5.5 describe the core analysis and optimization techniques. Finally, experimental results are discussed.

5.1 System Models

Taking imperfect fault detection into account, the execution of a task may result in three scenarios:

1. it executes successfully, denoted as SUC;
2. a transient fault occurs and is detected, denoted as Detected Unrecoverable Fault (DUF);
3. a transient fault occurs and is not detected, denoted as Silent Data Corruption (SDC).

We characterize the performance of a fault detector using a pair $d = \{c, o\}$, where c is the fault detection *coverage* in percentage and o is the timing overhead. The overhead is defined in

percentage with respect to the stand-alone WCET of the task. Let the stand-alone WCET of the task t_i on processor p_j without any fault detection be denoted using $w_{i,j}$. An instance of t_i that implements the fault detector indexed k has the WCET $w_{i,j}(1+o_k)$. We assume that a library of implementable fault detectors are available at design time for each task (denoted as D_i for task t_i).

For a specific instance $t_{i,l}$ in the set of replicas $R(t_i)$, the processor that $t_{i,l}$ is mapped to is denoted by $node(t_{i,l})$ and the ID of the fault detector it implements is denoted by $det(t_{i,l})$. The execution time and fault detection coverage of this instance are therefore $w_i^l = w_{i,node(t_{i,l})}(1+o_{det(t_{i,l})})$ and $c_{det(t_{i,l})}$, respectively. According to the Poisson fault model, the following formulas could be used to compute the probabilities that an instance is executed successfully (denoted by P_{SUC}) or it experiences detectable/undetectable faults (denoted by P_{DUF}/P_{SDC}):

$$
\begin{aligned}
P_{SUC}(t_{i,l}) &= e^{-\lambda_{node(t_{i,l})} w_i^l} \\
P_{DUF}(t_{i,l}) &= (1 - e^{-\lambda_{node(t_{i,l})} w_i^l}) c_{det(t_{i,l})} \\
P_{SDC}(t_{i,l}) &= (1 - e^{-\lambda_{node(t_{i,l})} w_i^l})(1 - c_{det(t_{i,l})})
\end{aligned}
$$

We again use the concept of *fault scenario* to describe the qualitative execution results of the replicas (i.e. if they deliver a correct output or not). Naturally, an element in the fault scenario has now three possible values.

Definition 2 (Updated Definition of Fault Scenario)**.** *A fault scenario is a vector $\boldsymbol{x} = \{x_1, ..., x_N\}$, which contains a variable $x_l \in \{1, 0, -1\}$ for each instance of a task t_i, where x_l is 1 if $t_{i,l}$ produces a correct output (i.e., SUC); x_l is 0 if $t_{i,l}$ encounters a fault that is detected (i.e. DUF) and x_l is -1 if $t_{i,l}$ encounters a fault that is not detected (i.e., SDC).*

We assume that a majority voter is implemented if redundancy is available. The voter collects results from all instances and produces a single output for the successor tasks. Each task instance tries to implement the fail-silent behavior, i.e., as long as the embedded fault detector reports a fault, this specific task instance will not produce any output. In this way, the voter considers only outputs from other instances and generates an output iff a dominating value (or a majority) is found. The overall execution of a task, considering all its instances, could again result in the 3 scenarios:

1. SUC: the voter successfully corrects all faults (if any). The faults that are corrected are called Detected and Tolerable Faults (DTF);

2. DUF: the voter fails to find a dominating result and thus produces no output;

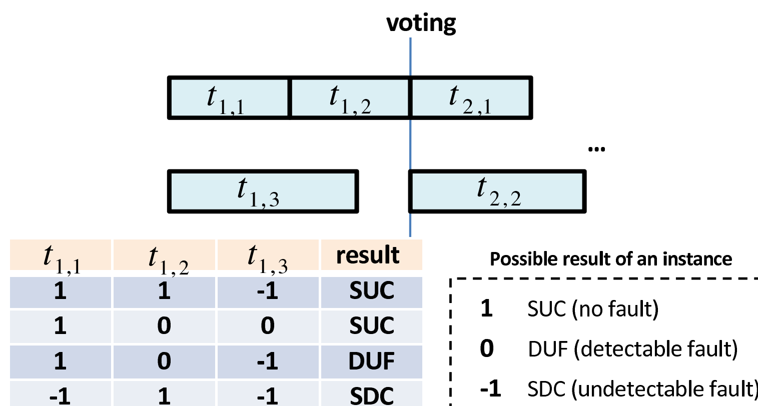


Figure 5.1: Example Fault Scenario

3. SDC: multiple faults occur and the incorrect outputs mask the correct one.

Both DUF and SDC are unwanted behavior that negatively influences the system reliability (see Section 5.2).

Figure 5.1 depicts an example of the voting scenario. If the fault scenario is $\mathbf{x} = \{1, 1, -1\}$, the incorrect output of $t_{1,3}$ is masked and the overall result is SUC. In the scenario $\mathbf{x} = \{1, 0, 0\}$, both $t_{1,2}$ and $t_{1,3}$ produce no result, and the only output from $t_{1,1}$ will be taken. Hence, the overall result is also SUC. In the scenario $\mathbf{x} = \{1, 0, -1\}$, a correct and an incorrect output are sent to the voter. However, the voter cannot identify the correct input since no majority is found. In this case, it generates no output and the overall result is DUF. In the last case $\mathbf{x} = \{-1, 1, -1\}$, two incorrect outputs are sent to the voter. Note that the fault scenarios model only the qualitative result (0,1, or -1), but the voting is performed based on the real value of the tasks' outputs. Hence, if two outputs are incorrect, two cases might happen: 1) the two incorrect outputs are equal and mask the single correct one, resulting in an SDC; 2) the two incorrect outputs are unequal and the voter does not see a dominating value, resulting in a DUF. To stay on the safe side, we have to assume the first case (SDC), because the probabilities of the two cases are very difficult to be quantified, even if possible[1].

5.2 Motivation

As discussed in Section 4.1, analyzing the system reliability in the presence of fault-tolerant mechanisms is a highly complex problem. To cope with the complexity, many state-of-the-art

[1]The probabilities are highly influenced by the application characteristic, the output data type, common caused errors, etc.

studies make simplifying assumptions on the fault models and modes. Perfect fail-silent behavior is one assumption that is often used in literature. It is assumed that all faults are detected within a certain time interval and the fault-detection overhead is contained in the tasks' Worst-Case Execution Times (WCETs), e.g., in fault-tolerant task scheduling [81, 12, 14, 24, 52, 98, 45], in reliability-aware energy management [88, 51, 26] and in error-aware system design [101, 46]. With this assumption, each task will produce either a correct output or no output at all. Although fail-silence is a highly desirable property, it is difficult to implement in practice. The prerequisite is the existence of a perfect fault detector that achieves 100% coverage under the given fault hypothesis.

The simplifying assumption of perfect fault detection is problematic. On the one hand, a perfect detector might not exist or is difficult to implement, making the algorithms developed under this assumption less useful in practice. On the other hand, even if implementable, perfect detectors typically come with high resource and timing overheads. In recent work [102, 47] it has been shown that the time needed for high-coverage fault detection may become much longer than the execution time of the task itself (e.g. the timing overhead could be 400% using techniques proposed in [102]). Hence, approaches under this assumption are very pessimistic, as the most expensive fault detector is selected for every task.

This problem can be viewed from a slightly different angle: choosing to implement the perfect fault detector is not only an **assumption** but also an important **design decision**. While making this assumption, all design alternatives with partial fault detectors are ignored without any justification. For example, when active redundancy is concerned, no analysis is performed to find out if it is more efficient to spend the available resources on applying better fault detection or a higher number of replications. Actually, our experimental results show that the answer is highly application and architecture dependent. This issue can be further explained using an example.

Consider a simple task running on a single processor system. We reuse the result of [47] and assume that the rate of undetectable faults decreases exponentially with linear fault detection effort. It is further assumed that the perfect fault detection (100% coverage) incurs 300% timing overhead (typical value in [47]). Figure 5.2a depicts the schedule using the perfect detector. By spending all resources on fault detection, SDCs are completely eliminated. Figure 5.2b is another possible schedule, in which the task is replicated twice and the remaining time (200% task execution time in this case) is used to implement two partial fault detectors (each 90% coverage using the 100% detection effort). Figures 5.2c and 5.2d show two similar schedules

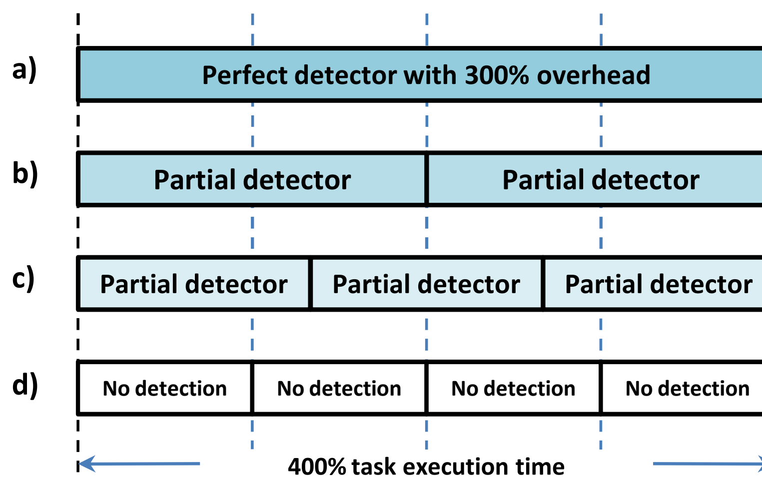


Figure 5.2: Example Scenario

with higher number of replications. When multiple replicas of the same task are available, the results from different instances can be compared to detect or even mask the faults. Figure 5.3 compares the probability of DUF and SDC for each schedule. For schedule a, although SDCs are avoided completely, the DUF probability is very high, since any transient fault occurring on the single task instance results in a DUF. With imperfect fault detectors (schedule b to d), SDC will not totally disappear but the probability of DUF can be significantly reduced. If both types of faults are considered together, the overall failure probability ($DUF + SDC$) of schedule c is almost six orders of magnitude lower than that of schedule a.

The selection of the best schedule depends on the reliability goal of the application. Many systems have specific requirements concerning DUF and/or SDC. For example, the IBM Power 4 processor-based systems target 10-25 years Mean Time Between Failures (MTBF) for DUF and 1000 years MTBF for SDC [103]. The schedule using perfect fault detectors may not meet the requirements of all applications. Moreover, the criticality of a certain type of faults is application-specific. For systems that require fail-operational behavior, DUFs and SDCs could be equally bad and schedule c is clearly a much better design choice. For other systems, SDCs might be more critical and schedule a or d are more preferable.

From the analysis above, it can be seen that the selection of appropriate fault detectors is critical. The decision has to be made jointly with other design parameters, e.g., task mapping and utilization of redundancy. However, the existing work assuming perfect fault detection prohibits the exploration of design alternatives using partial fault detectors. To tackle this problem, we need 1) a way to evaluate the system quality regarding both DUF and SDC; and

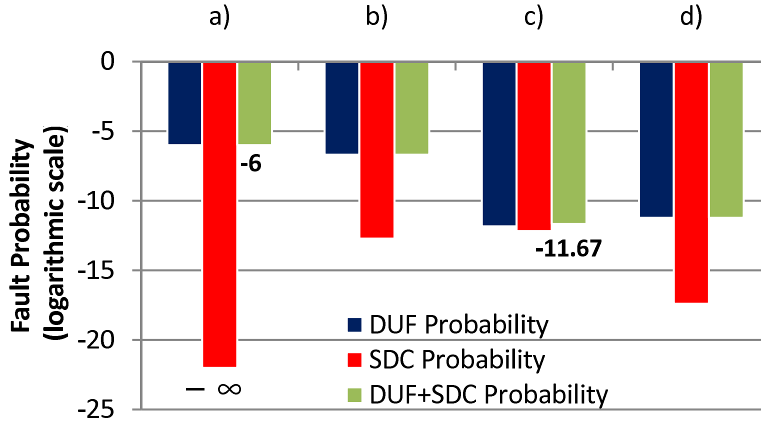


Figure 5.3: Reliability of the Example Schedules

2) an optimization approach for reliability-aware DSE with selection of fault detectors as design parameters. The following sections in this chapter present our approach to tackle these issues.

5.3 Experimental Analysis on the Impact of Imperfect Fault Detection on System Reliability

To understand the impact of imperfect fault detection on the system reliability, we carried out a set of experiments considering two scenarios. In the first one, we fix the amount of redundancy and analyze the influence of detection coverage on the system-level reliability. In the second one, we do it vice-versa, i.e., varying the number of replications with fixed fault detector. In general, we observe that the selection of fault detector and the utilization of redundancy show a tradeoff. In particular, when the system features only limited amount of resources or the application has tight timing constraints, inappropriate selection of fault detector might disallow certain options for redundancy due to the timing overhead.

Figure 5.4 to 5.6 summarizes the results of the first simulation. We increase the fault detection coverage from 1% to 100% with a step width of 1% while fixing the number of replications. Figure 5.4 shows the case that a single instance is scheduled. As expected, the probability of SDC decreases linearly with the detection coverage, since all detected faults are converted to $DUFs$. In Figure 5.5, two replicas are scheduled. The probabilities of both SDC and DUF decrease with higher coverage. The reason is that, if used together, the effects of redundancy and that of fault detection become *correlated*. As an example, assume the first instance generates a correct output whereas the second one encounters a fault. If the fault is

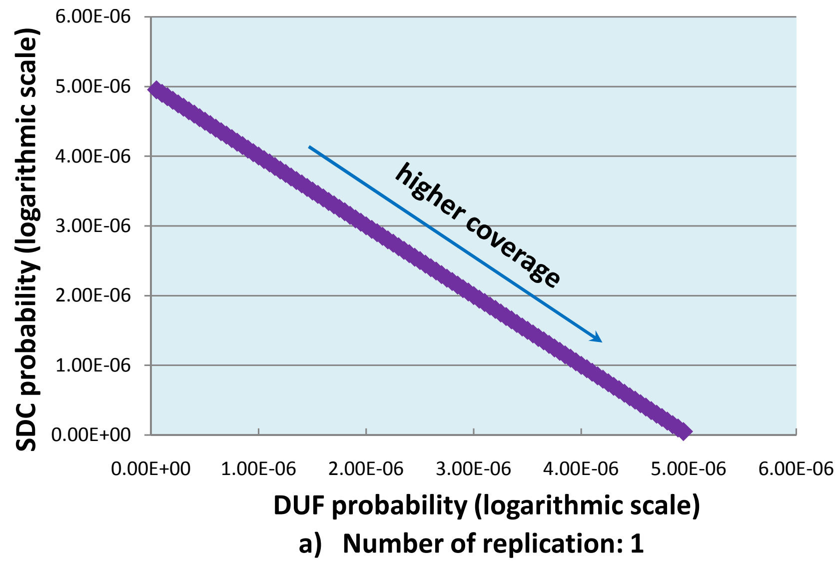


Figure 5.4: Effect of Fault Detection with Fixed Replication: replication = 1

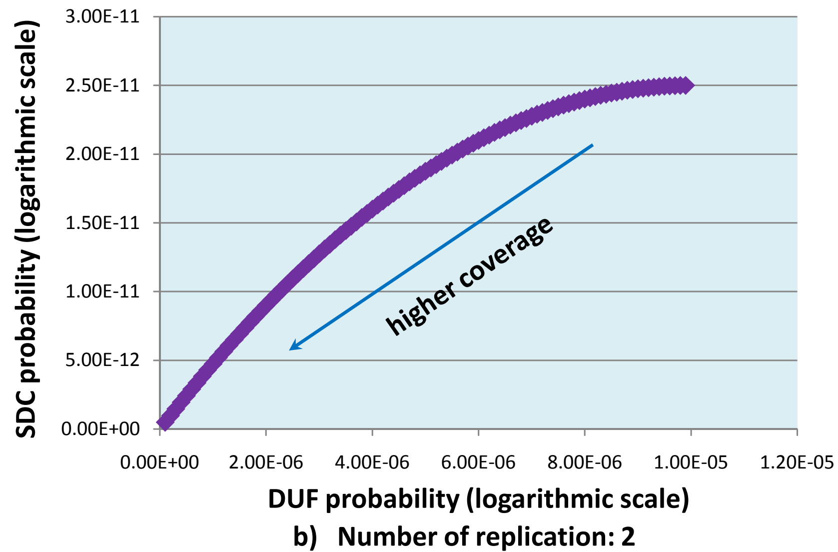


Figure 5.5: Effect of Fault Detection with Fixed Replication: replication = 2

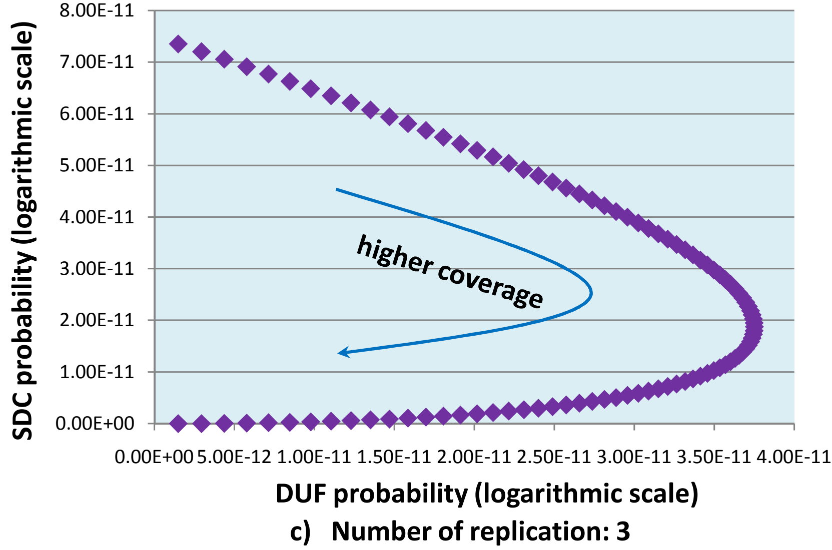


Figure 5.6: Effect of Fault Detection with Fixed Replication: replication = 3

undetected, the second one will produce a faulty output. Since the voter cannot distinguish the correct result from the two inputs, the system results in a DUF. As the counterpart, if the fault is detected, the faulty instance can fail-silent and the only (and correct) output from the first instance is taken, resulting in a SUC scenario. Hence, besides converting $SDCs$ to $DUFs$, fault detection can also convert $DUFs$ to $DTFs$ if voting is available. For this reason, probabilities of both DUF and SDC decease.

If three replicas are utilized (Figure 5.6), the DUF probability first increases and then decreases with higher coverage, whereas the probability of SDC decreases constantly. The reason is that, the effect of SDC-to-DUF dominates when the coverage is still low (upper part of the Figure 5.6), and the effect of DUF-to-DTF dominates when the coverage is relatively high. An observation from this set of simulations is that, higher fault detection coverage reduces the amount of SDCs but not necessarily reduces the amount of DUFs.

In the second simulation, we increase the number of replications while fixing the detector implementation. Similar as the motivating example, we again use the result of [47] and assume that the rate of undetectable faults decreases exponentially with linear fault detection effort. Several fault detectors with timing overhead ranging from 0% to 300% (corresponds to detection coverage from 0% to 100%) are tested. Figure 5.7 summarizes the results[1]. As can be seen, when the detection coverage is low, the probability curve shows a zigzag behavior with increasing

[1]The figure excludes the case of 0% and 300% by intension, because some of the probabilities are 0 and hard to be visualized in logarithmic scale.

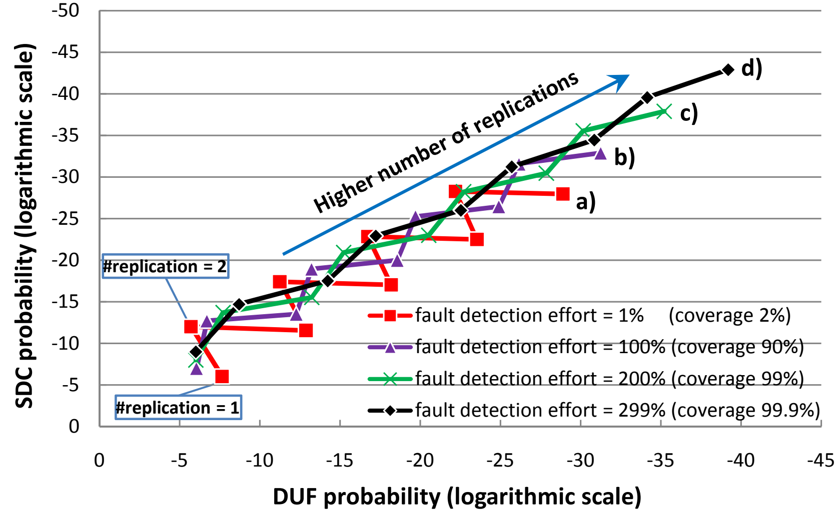


Figure 5.7: Effect of Replication with Fixed Fault Detection Coverage

number of replications (e.g., curve a). This is because the task instances themselves have only poor fault detection and the system relies mainly on the voter to discover the faults. On the one hand, the voter detects a fault when the number of correct and incorrect results breaks even. Hence, when we increment the number of replicas from an odd number and make it even (e.g., from 1 to 2), the fault detection capability of the voter is enhanced, resulting in a reduction of undetected faults (SDC probability drops). On the other hand, the voter recovers a fault when correct results dominate. Hence, when a new instance is added to an even number of replicas, the amount of recoverable faults increases (DUF probability drops).

As the counterpart, if the task instances have already good fault detectors (e.g., in the case of curve d), the system reliability will be improved more smoothly by inserting extra redundancy, i.e., both $DUFs$ and $SDCs$ can be eliminated at the same time. In other words, the effect of active redundancy could be amplified by good fault detection.

These experimental results reveal the correlation between fault detector implementation and active redundancy configuration. Due to such correlation, the configuration of both fault-tolerant mechanisms must be considered jointly. In our approach, we extend the optimization technique presented in Section 4.3 and consider fault detector implementation as an additional design freedom during DSE.

5.4 Reliability Analysis

Using the voting setup introduced in Section 2.3, the schedule generated by our algorithm falls into the category of *strict schedules* [24, 77]. Strict schedules obey the rule that if a task t has a data dependency on task t', all replicas of t' should be completed before any replica of t starts. With this restriction, all tasks use exclusively the voter output and the tasks of a TG can be considered independently in the reliability analysis.

For a task t_i, a fault scenario x is *tolerable* if the voter can produce a correct output in the presence of the faults specified in x. This condition can be computed by the following binary function *tolerable()*, which evaluates to *true* if the correct outputs are able to dominate.

$$tolerable(x) = ((\sum_{t_{i,l} \in R(t_i)} x_l) > 0) \tag{5.1}$$

Where $R(t_i)$ denotes the set of replicas of task t_i and $x_l \in \{1, 0, -1\}$ is the execution result of task $t_{i,l}$. Similarly, the fault scenario x is *silent* if the voter cannot distinguish a dominating result and x is $faulty$ if the incorrect results are majority.

$$silent(x) = ((\sum_{t_{i,l} \in R(t_i)} x_l) = 0) \tag{5.2}$$

$$faulty(x) = ((\sum_{t_{i,l} \in R(t_i)} x_l) < 0) \tag{5.3}$$

The probability that a task is executed successfully can be computed by summarizing the occurrence probability of all tolerable fault scenarios:

$$P_{SUC}(t_i) = (\sum_{\forall x: tolerable(x) = true} P(t_i, x)) \tag{5.4}$$

where $Pr(t_i, x)$ is the probability that the fault scenario x happens. As x specifies the qualitative execution result ($SUC/DUF/SDC$) of each instance of task t_i, the probability $Pr(t_i, x)$ can be computed as a product of occurrence probability of each task instance:

$$P(t_i, x) = \prod_{\substack{t_{i,l} \in R(t_i) \\ \wedge x_l = 1}} P_{SUC}(t_{i,l}) \prod_{\substack{t_{i,l} \in R(t_i) \\ \wedge x_l = 0}} P_{DUF}(t_{i,l}) \prod_{\substack{t_{i,l} \in R(t_i) \\ \wedge x_l = -1}} P_{SDC}(t_{i,l})$$

The instance-level probabilities $P_{SUC}(t_{i,l})$, $P_{DUF}(t_{i,l})$ and $P_{SDC}(t_{i,l})$ are computed from the fault model introduced in Section 5.1. In a similar way as in equation 5.4, the probability that

a task results in a fail-silence ($P_{DUF}(t_i)$) or it produces a faulty output ($P_{SDC}(t_i)$) can be computed:

$$P_{DUF}(t_i) = (\sum_{\forall x: silent(x)=true} P(t_i, x)) \tag{5.5}$$

$$P_{SDC}(t_i) = (\sum_{\forall x: faulty(x)=true} P(t_i, x)) \tag{5.6}$$

The complete set of tolerable (or silent or faulty) scenarios can be obtained by systematically enumerating all fault scenarios. Since each task instance has three possible results (1,0, or -1), the overall number of combinations is 3^N, where N is the number of replicas. Although this enumeration has exponential complexity, it is still acceptable in practice since the number of replicas for a task is typically very small, e.g., more than 3 replicas for a task is rarely used in practice. The above step is performed for all tasks in the application so that the task-level probabilities $P_{SUC}(t)$, $P_{DUF}(t)$ and $P_{SDC}(t)$ are obtained. Then, we proceed with analyzing the reliability of the entire application. Naturally, an application consisting of tasks $\mathcal{T}$ is successful (i.e. SUC) only if all of its tasks are successful:

$$P_{SUC}(\mathcal{T}) = (\prod_{t_i \in \mathcal{T}} P_{SUC}(t_i)) \tag{5.7}$$

The application is silent (i.e. DUF) if at least one of its tasks is silent, because if any task fails to produce an output, the successor tasks cannot proceed due to data dependency and the entire application has to start over. This probability is denoted by $P_{DUF}(\mathcal{T})$. The application is faulty (i.e. SDC, the corresponding probability is denoted by $P_{SDC}(\mathcal{T})$), if none of its tasks is silent and at least one of its tasks is faulty. Assume t_0 is the first task in $\mathcal{T}$, the application is faulty if t_0 is faulty and the remaining tasks are non-silent (denoted by $P_{\overline{DUF}}(\mathcal{T}\backslash t_0)$), or t_0 is successful and the remaining tasks are faulty.

$$P_{SDC}(\mathcal{T}) = P_{SDC}(t_0) P_{\overline{DUF}}(\mathcal{T}\backslash t_0) + P_{SUC}(t_0) P_{SDC}(\mathcal{T}\backslash t_0)$$

Since $P_{\overline{DUF}}(\mathcal{T}\backslash t_0)$ is the sum of $P_{SUC}(\mathcal{T}\backslash t_0)$ and $P_{SDC}(\mathcal{T}\backslash t_0)$, the above formula can be rewritten as:

$$\begin{aligned} P_{SDC}(\mathcal{T}) &= P_{SDC}(t_0)(P_{SUC}(\mathcal{T}\backslash t_0) + P_{SDC}(\mathcal{T}\backslash t_0)) + P_{SUC}(t_0) P_{SDC}(\mathcal{T}\backslash t_0) \\ &= P_{SDC}(t_0) P_{SUC}(\mathcal{T}\backslash t_0) + (P_{SDC}(t_0) + P_{SUC}(t_0)) P_{SDC}(\mathcal{T}\backslash t_0) \end{aligned} \tag{5.8}$$

In the above formulas $P_{SDC}(t_0)$ and $P_{SDC}(t_0)$ are computed using equation 5.4 and $P_{SUC}(\mathcal{T}\backslash t_0)$ is computed using equation 5.7. The only unknown term is $P_{SDC}(\mathcal{T}\backslash t_0)$. However, calculating

$P_{SDC}(\mathcal{T} \backslash t_0)$ is a reduced problem since it is the SDC probability of the application without considering the first task t_0. Hence, the calculation can be implemented using recursion. The complexity is linear with the number of tasks. The DUF probability can then be computed by:

$$P_{DUF}(\mathcal{T}) = 1 - P_{SUC}(\mathcal{T}) - P_{SDC}(\mathcal{T}) \tag{5.9}$$

5.5 Optimization Procedure

After having the reliability analysis, the next step is to develop an optimization approach to search for high-quality designs. We identify two major scenarios that the designers may encounter. In the first one, the system is intended to execute a single application, so the design goal is to maximize the reliability while meeting the deadline. We show that this problem can be transformed into a deadline assignment problem that can be solved using Integer Linear Programming (ILP). Section 5.5.1 details the transformation and ILP formulation. In the second scenario, multiple applications may be executed on the same platform. We add an additional optimization objective that the resource consumption is to be minimized so that more space can be reserved for future applications. A Multi-Objective Evolutionary Algorithm (MOEA) based optimization approach is presented for this problem.

5.5.1 ILP Based Optimization for Single-Objective Case

This section presents an ILP based solution to handle the design scenario of maximizing the reliability of a single application. A real-time application typically has an end-to-end deadline that represents the time budget B for the entire application. The total budget can be distributed to individual tasks so that each task t_i has a local deadline b_i. The maximum reliability that can be archived by a task is constrained by the available local time budget. To describe this relationship, we define a Reliability Function (RF) $U_i(b)$, which is a monotonic function that models the achievable reliability of task t_i with given time budget b. Figure 5.8 depicts an example RF. The metric for reliability is Failure In Time (FIT). To capture both $DUFs$ and $SDCs$, we define $U_i(b)$ as a weighted sum of the FIT of both fault classes, i.e.:

$$U_i(b_i) = \alpha FIT_{DUF}(b_i) + \beta FIT_{SDC}(b_i) \tag{5.10}$$

The weighting factors represent the criticality of the type of fault for the application. The RF for a task t_i can be obtained as follows. The possible time budget b_i assigned to t_i is lower-bounded by its execution time and upper-bounded by the available system slack time,

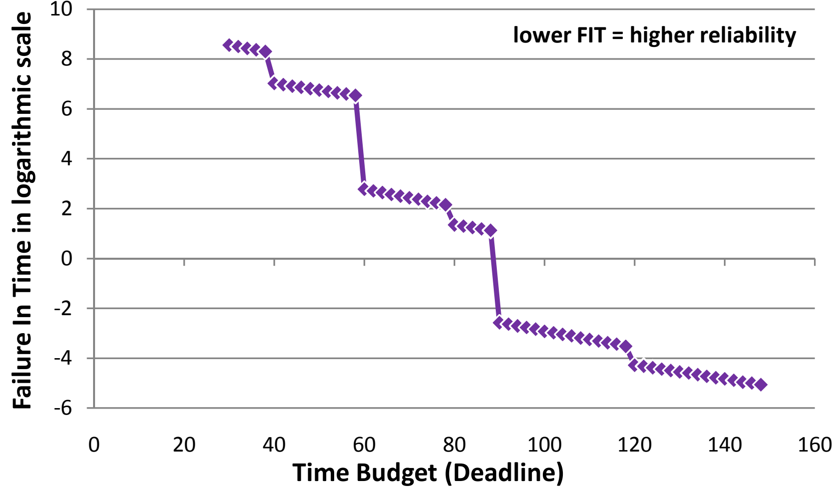


Figure 5.8: Example of Reliability Function

i.e., $b_i \in [C_i, C_i + B - \sum_{\forall j} C_j]$. We sample this range with a fixed step width. For each sample value b, we investigate all design alternatives that fit into b, i.e., we try different numbers of replications and all implementable fault detectors[1]. For each design, the DUF and SDC probabilities are analyzed using equation 5.1-5.3 and the reliability is evaluated by equation 5.10. We assign the value of the $U(b)$ to be highest achievable reliability under the budget constraint.

With the reliability function for all tasks, we can now proceed with calculation of the system reliability. The system-level SDC probability can be computed using equation 5.8. Since the success probabilities $P_{SUC}(\mathfrak{T} \backslash t_0)$ and $P_{SUC}(t_0)$ are typically very close to 1, we approximate equation 5.8 as follows:

$$P_{SDC}(\mathfrak{T}) < P_{SDC}(t_0) + P_{SDC}(\mathfrak{T} \backslash t_0) < ...$$
$$= \sum_i P_{SDC}(t_i)$$

As can be seen, the system-level SDC probability can be overestimated by summarizing the SDC probabilities of all tasks. It can easily be verified that the system FIT can also be computed in an additive manner from the tasks' FITs. Similar approximation exists for the DUF probability. Let $\vec{\mathbf{b}}$ be a vector that contains the timing budget for each task. The system

[1]This procedure is durable since the number of alternatives is very limited. On the one hand, the number of replications for a single task is typically very small. On the other hand, since fault coverage increases monotonically with detection effort, we can simply choose the best detector that fits into the budget. When the complexity is still too high, methods like Monte Carlo simulation can be used to approximate the RF.

reliability can be approximated as $U_{sys}(\vec{\mathbf{b}}) = \sum_{t_i \in \mathfrak{T}} U_i(b_i)$. The optimization problem becomes a deadline assignment problem stated as follows:

$$\begin{aligned} &Minimize : U_{sys}(\vec{\mathbf{b}}) = \sum_{t_i \in \mathfrak{T}} U_i(b_i), \\ &Subject\ to : \sum_{b_i \in \vec{\mathbf{b}}} b_i \leq B \end{aligned} \tag{5.11}$$

By restricting the local time budget of each task to be a set of discrete values (as what is done to sample the RF), the above problem can be transformed into an integer linear programming problem and solved using standard solvers. Assume that M samples in the RF are considered for each local deadline value, i.e. $b_i \in \{b_{i,1}, ..., b_{i,M}\}$. We define a set of binary variables to describe the assignment of b_i:

$$x_{i,m} = \begin{cases} 1 & \text{iff } b_i \text{ is assigned to the } m\text{th sample } b_{i,m} \\ 0 & \text{otherwise} \end{cases}$$

Obviously, b_i can only be assigned to exactly one sampling value:

$$\sum_{m \in [1,M]} x_{i,m} = 1, \ \forall t_i \in \mathfrak{T}.$$

The actual value of b_i can then be denoted as:

$$b_i = \sum_{m \in [1,M]} x_{i,m} b_{i,m}.$$

The actual reliability of the task i is:

$$u_i = \sum_{m \in [1,M]} x_{i,m} U_i(b_{i,m}).$$

The ILP problem can be stated as:

$$\begin{aligned} &Minimize : \sum_{t_i \in \mathfrak{T}} u_i, \\ &Subject\ to : \sum_{t_i \in \mathfrak{T}} b_i \leq B \end{aligned} \tag{5.12}$$

The ILP formulation consists of $M\,|\mathfrak{T}|$ binary variables (the x variables) and $2\,|\mathfrak{T}|$ integer variables (for the b and u variables).

5.5.2 Multi-Objective Optimization

The multi-objective optimization problem is again solved using MOEA. To take imperfect fault detection into account, the coding techniques presented in Section 4.3 is extended. For each

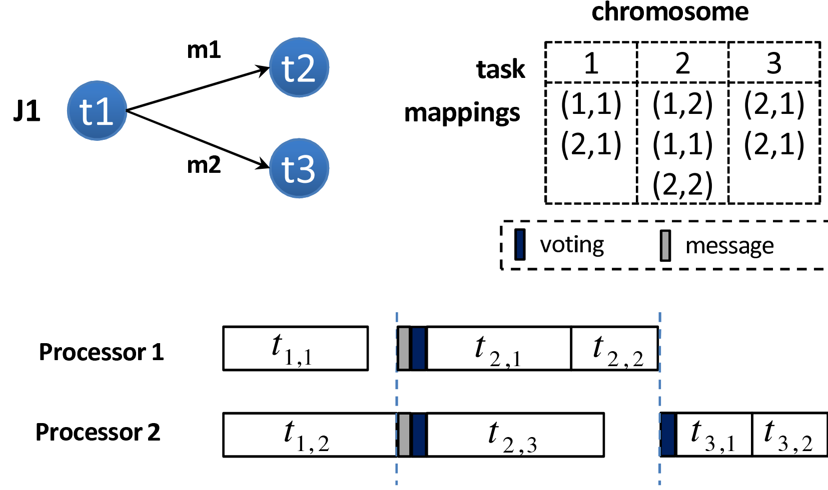


Figure 5.9: Example of Encoding Scheme

task instance, an additional attribute is added to indicate which fault detector is selected for it. The updated encoding maintains a gene (i, M) for each task, where i is the integer index of the task and M is a list of mapping entries, each representing a replica of task i. The mapping entry is a pair (p, d), where p is the processor that the task instance is executed on and d is the index of the fault detector it implements. Figure 5.9 illustrates an example, in which task 1 and 3 are replicated 2 times and task 2 is replicated 3 times. The lower part of the figure depicts the corresponding schedule that the chromosome represents. Since we target on generating strict schedules [24, 77], the reconstruction of the schedule from the chromosome can be done using a simple greedy heuristic. We consider all tasks in the TG in topological order. For each task, the replicas specified in the chromosome are instantiated and scheduled greedily at the earliest possible time. Output messages are scheduled at the end of execution. If the current task has data dependency on previous tasks, a voter is inserted. The failure rate of the voter is added to the failure rate of the current task.

We consider three optimization objectives. The first two are the reliability objectives, one for DUF and one for SDC. The metric is Failure In Time (FIT). One unit FIT specifies one failure in a billion hours. The conversion from failure probabilities computed in section 5.4 to FIT is as follows:

$$FIT = 10^9 * 3600 * (\frac{1}{p}) * Pr \tag{5.13}$$

where p is period of the application in second and Pr is the failure probability of the application, i.e. $P_{DUF}(\mathcal{T})$ or $P_{SDC}(\mathcal{T})$. In the third objective, we intend to encode the design goal

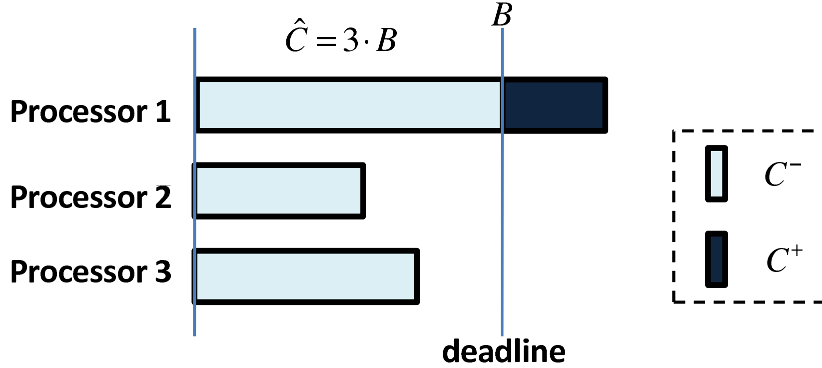


Figure 5.10: The Resource Consumption Objective

of minimizing resource consumption while meeting the deadline. The resource consumption (denoted by C) is defined as the overall processor time that a schedule occupies. Let B be the deadline of the application and N be the number of processors available in the execution platform. The available time budget within the deadline is $\hat{C} = NB$. For a given schedule S, we use C^- to denote the fraction of resource consumption within the deadline and C^+ for the part above the deadline. Figure 5.10 depicts an example. The objective function is defined as follows:

$$penalty = \begin{cases} C & \text{iff } C^+ = 0 \\ \hat{C} + C^+ & \text{otherwise} \end{cases} \tag{5.14}$$

By constructing the objective function as above, each schedule that violates the deadline ($C^+ > 0$) has a higher penalty value than any schedule that meets the deadline. For two schedules that meet the deadline, the one that has less resource consumption will be preferred. Clearly, all three objectives are to be minimized.

5.6 Experiments

The analysis and optimization algorithms are implemented in JAVA and integrated to the modeling framework. We use a similar experiment setup as in Chapter 4. The target platform consists of two types of Processing Elements (PEs), namely a RISC processor and a DSP. The failure probability of each task on a certain PE is randomly generated between 1×10^{-5} and 1×10^{-7} . Random fault detectors are generated based on the exponential model in [47], i.e., the undetectable faults reduce exponentially with linear fault detection effort.

The proposed approach is applied on an mpeg2 decoder example [95]. We compare the performance of two approaches: 1) the proposed approach that explores the optimal utilization

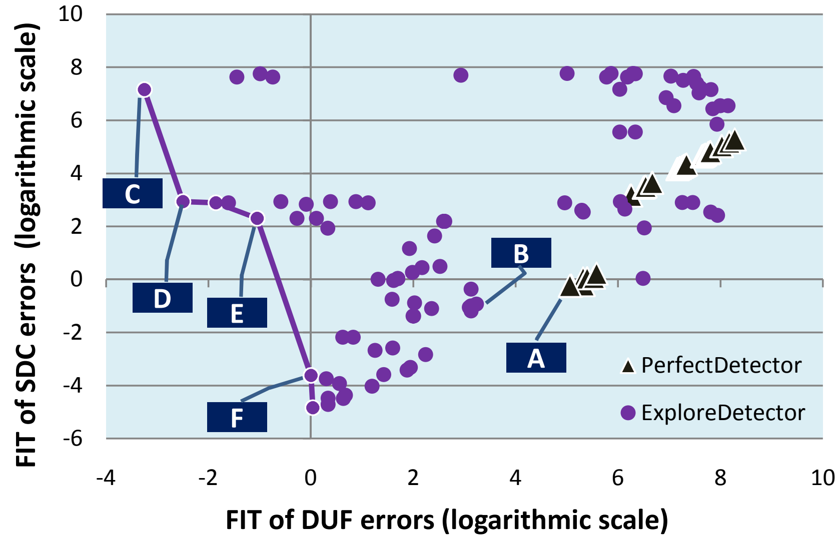


Figure 5.11: 2D Projection of Optimization Results

of fault detectors (*ExploreDetector*); 2) existing approaches that utilize the perfect fault detector[1] for all tasks (*PerfectDetector*). We use the MOEA optimizer to compute the Pareto optimal solutions considering the three objectives introduced in section 5.5.2. Figure 5.11 shows the results on a platform that consists of 2 *RISC*s and 2 *DSP*s. The dots in the figure show the solutions projected into a 2D plane, with the vertical axis being the FIT of *SDC* and the horizontal axis being the FIT of *DUF*. The Pareto front considering only the two reliability objectives is marked using a solid line. The triangle symbols in Figure 5.11 show the results of the *PerfectDetector* approach.

It can be seen that the solutions found by *ExploreDetector* is of much higher quality than those found by *PerfectDetector*. The difference in terms of FIT is up to several orders of magnitude. For the *PerfectDetector* approach, the FIT of *SDC* can still be kept relatively low due to good detection coverage but the FIT of *DUF* is the major issue (always beyond 10^5 in this experiment). Another advantage of the *ExploreDetector* approach is that it provides a much wider spectrum of solutions, from the one that achieves very low FIT of *DUF* (*C* in Figure 5.11) to the one that achieves very low FIT of *SDC* (*F* in Figure 5.11). This allows the designer to carefully evaluate the tradeoff between the two classes of faults and select the implementation that fits the application requirements.

[1] For better visualization in the logarithmic scale, the results we presents are using the detector with 99.9% coverage. If the perfect fault detector is used, the probability of *SDC* can be reduced to 0, but the probability of *DUF* remains almost the same.

solution	DUF FIT (log.)	SDC FIT (log.)	avg. rep.	avg. cov.(%)	resource (time unit)
A	5.06	-0.23	3.25	99.9	114.0
B	3.24	-0.93	3.67	63.0	74.2
C	-3.25	7.15	3.50	84.4	55.9
D	-2.49	2.93	3.83	74.9	65.5
E	-1.04	2.30	3.92	83.0	150.6
F	0	-3.62	3.92	89.3	189.5

Table 5.1: Comparing Representative Implementation Alternatives

We mark some representative implementation alternatives in Figure 5.11. A is the best solution in terms of reliability found by the $PerfectDetector$ approach; B is a solution found by $ExploreDetector$ which is close to and dominates A; C to F belong to the Pareto optimal solutions found by $ExploreDetector$. Table 5.1 compares these implementations in several aspects, e.g., the average number of replications for each task, the average fault detection coverage over all task instances and the resource consumption. It can be seen that implementation A has the lowest number of replications, since a lot of resources are already consumed by fault detection. The solution B has higher quality than A concerning all three objectives. Using fault detectors with average coverage of 63%, it achieves much higher reliability than A and saves 35% resources. The implementation F achieves higher reliability than A as well. By spending 65% more resources, it reduces the FIT of DUF by 5 orders of magnitude and the FIT of SDC by more than 3 orders of magnitude. It is also worth noticing that, since most of the solutions found by $PerfectDetector$ implement 2 replicas, the curve formed by those solutions has similar shape as the curve in Figure 5.5.

The optimization results from MOEA can also be viewed from different angles. In Figure 5.12, the results are projected to in a 2D plane considering the FIT of DUF and resource consumption. Similarly, Figure 5.13 focuses on the FIT of SDC and resource consumption. Clearly, for both cases, the solutions found by $ExploreDetector$ have better quality than those found by $PerfectDetector$. Concerning SDC, the performance of $PerfectDetector$ is relatively close to that of $ExploreDetector$. Nevertheless, the performance gap is significant for DUF. In this sense, the main drawback of the $PerfectDetector$ approach is that the design objective is biased. It fails to take application-specific reliability requirements into account. Instead, the focus is always on reducing the SDCs. For many applications (e.g., those requires fail-operational behavior), this is certainly suboptimal.

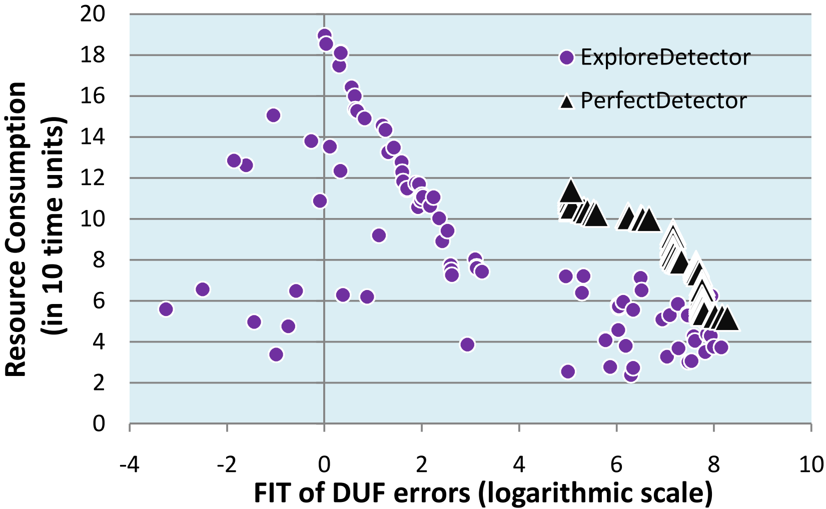


Figure 5.12: 2D Projection of Optimization Results: FIT of DUF vs Resource Consumption

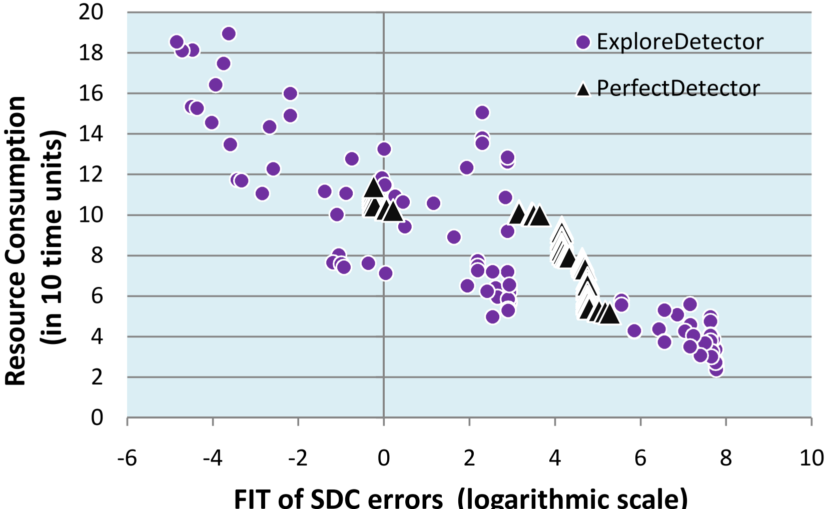


Figure 5.13: 2D Projection of Optimization Results: FIT of SDC vs Resource Consumption

Application (num. tasks)	200 round	500 round	1000 round	1500 round
mpeg2(13 tasks)	29.0	76.4	120.8	198.3
TG1(50 tasks)	78.3	179.0	395.1	583.0
TG2 (100 tasks)	195.9	442.2	777.0	1692.0

Table 5.2: Execution Time of Optimization Approach

We measure the execution time (in seconds) of our approach on a Windows machine with 3GHz CPU. The MOEA is configured to run for 200, 500, 1000 and 1500 rounds. Table 5.2 presents the results. For a small TG (e.g., mpeg2), the analysis and optimization procedure takes only a few minutes to execute for 1500 iterations. As expected, the execution time grows linearly with the number of iterations. It is also worth mentioning that the execution time also increases roughly linearly with the size of TG. This is because the reliability analysis, as most computational intensive operation, has linear complexity in the number of tasks. For a syntactic TG[1] with 100 tasks, the 1000-iteration EA takes about 13 minutes. In general, the runtime is acceptable for an off-line design space exploration procedure.

The propose approach can be used to perform reliability-aware architecture space exploration. To do this, we just need to apply the optimization approach on the candidate platforms. In Figure 5.14, we compare the maximum achievable reliability using three platforms consisting of 2 to 4 processors. Clearly, the solutions found using a larger architecture dominate those obtained using a smaller architecture, due to the extra resources to implement more replications and/or better detectors. From these results, the designer may choose the best platform that meets the application requirement. For example, if the reliability goal is point A in Figure 5.14, the $2RISC + 1DSP$ platform is the cheapest one adhering to the requirement.

5.7 Summary

This Chapter presents our novel techniques to support imperfect fault detection in reliability analysis and optimization. It removes the unrealistic assumption and constitutes an essential step to enable practical use of the proposed design flow. Together with the previous chapter, we have presented our reliability-aware DSE approach. The DSE process takes an abstract model of the application and platform and computes the recommended design parameters, including mapping, scheduling and FTM configurations. The next step in the design flow is to bring

[1]generated using TGFF `http://ziyang.eecs.umich.edu/~dickrp/tgff/`

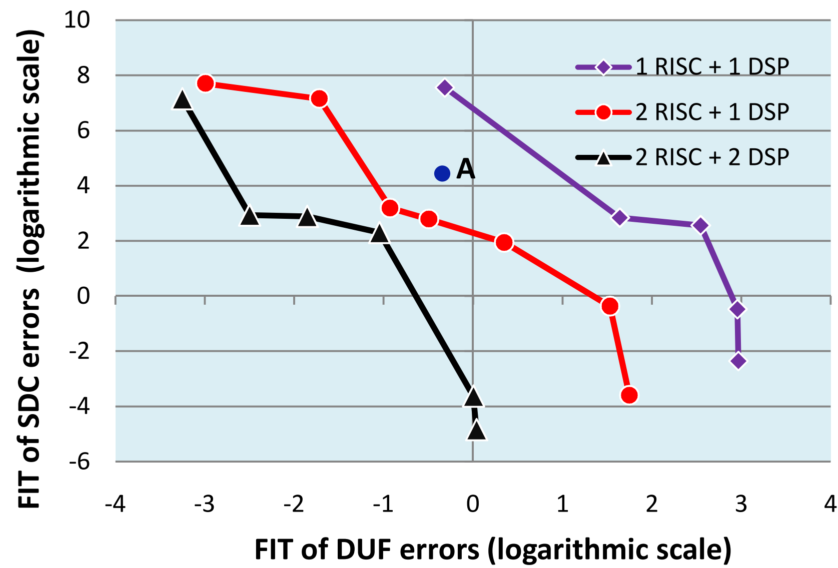


Figure 5.14: Comparing Results of three Architectures

the abstract design in to reality. The code/configuration generation backend is developed to facilitate this step.

Chapter 6

Automatic Code Generation and Platform Configuration

After the DSE phase introduced in previous two chapters, the final step in the design flow is to realize the abstract design in the target platform. Several challenging tasks are involved in this step, in particular the multi-processor implementation of the application software and the configuration of the platform. Using the design model as input, the back-end of our framework facilitates the implementation phase by automatic generation of implementation artifacts, including application source code and platform configuration files. This chapter discusses the design of the back-end in detail.

6.1 Template Based Code Generation

Code generation is a model-to-text transformation that derives part or all of the source code of the system from an abstract model. As a common module in model-driven engineering, code generation has several advantages:

- Productivity. Code generation abstracts away the implementation details. The design can be performed using a much more human-elaborated modeling language, from which software is produced instantly. It reduces the burden of programing complex architectures such as MPSoCs. The overall design process is therefore accelerated.
- Software Quality. Manuel implementation of the application software is error-prone, especially when the domain experts lack detailed knowledge of the hardware platform. Instead, the code generator is developed by platform experts and the generated code is thoroughly

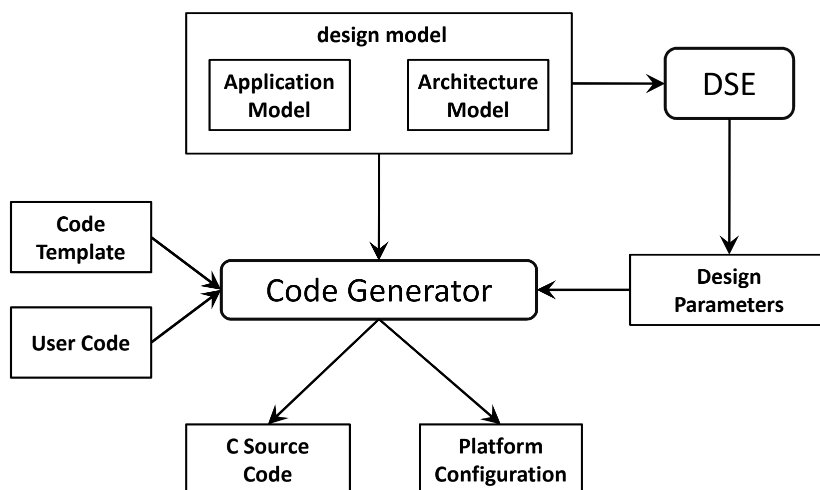


Figure 6.1: Code Generation Flow

verified during the development. Moreover, the code generator is typically used in many design projects and is therefore well tested in field. The potential implementation errors are expected to be reduced significantly.

- Retargetability. The abstract model is a generic representation that can be used to generate code for multiple platforms. An appropriately designed code generator will automatically adapt the generated code according to platform specification, allowing a central design to be retargeted easily.

6.1.1 Code Generation Strategy

In our framework, a template-based approach is adopted to generate text files from the models. The *code templates* specify a skeleton of the output files. It may contain both hard-coded static contents that are used directly, and dynamic contents. The latter contains references to elements in the input model, which are expanded during the code generation process. Figure 6.1 illustrates the code generation scenario. Besides the design model created by the user, the design parameters determined during the DSE are also considered, including the mapping of software tasks to cores, the scheduling parameters of tasks and messages, etc. In certain locations, also user-supplied C code can be integrated to the generated code. This allows for easy integration of legacy code to our framework. The user-supplied source files are directly associated to corresponding model elements using dedicated annotation objects. Here, the parts

```
1 «REM» entry point of code generation for a KPN component «ENDREM»
2 «DEFINE entryPoint FOR pn::PnComponent»
3   «REM» specify the output file «ENDREM»
4   «FILE "pn.c"»
5     //pn component's init function
6     int pn_«this.name»_init (void) {
7       «REM» call subroutine to generate init function «ENDREM»
8       «EXPAND generateInitFunction FOR this»
9     }
10    //pn component's read function
11    int pn_«this.name»_read(void) {
12      «REM» call subroutine to generate read function «ENDREM»
13      «EXPAND generateReadFunction FOR this»
15    }
16    ......
17  «ENDFILE»
18 «ENDDEFINE»
```

Figure 6.2: Example Code Template

to be extracted by the code generator is marked up using dedicated C comments (see Section 6.1.2).

The code templates are developed using the *Xpand* language provided by EMF. It is a template engine tailored for text file generation from EMF models. In *Xpand*, we can import meta-model packages developed using *Ecore* and reference the corresponding model elements in the statements. A snippet of *Xpand* code is shown in Figure 6.2. The code template for a specific class in the meta-model starts with a `DEFINE` statement. In the example, we develop a code template for `KPNComponent`. During code generation process, the template is executed for a concrete object of matching type. The input object can be referred to using the `this` pointer. This allows us to traverse the input model to obtain the information required. For example, in line 6, we access the `name` attribute of the KPN object by `this.name`. In *Xpand* language, statements in double brackets "≪" and "≫" specify commands to the code generator. Other text outside the double brackets is considered as static text going directly to the output file. The `"REM"`/`"ENDREM"` commands are used to add comments and `"FILE"`/`"ENDFILE"` are used to specify the output path. In the example, we generate a C source file *pn.c* for the `KPNComponent` object. *Xpand* supports subroutines in template specification. A subroutine is invoked using the `EXPAND` command as shown in line 8 of Figure 6.2.

One useful feature of *Xpand* is the extension point to Java. Static Java subroutines can be directly called in the code template. This allows us to delegate complex functions that are otherwise difficult to implement in *Xpand* to external Java code or tools. In our implementation, we develop a library of auxiliary functions for model analysis and transformation to support the code template development. Another example use case is the integration of TTNoC configuration tool introduced in Section 6.2. The configuration involves a complex scheduling problem solved using formal methods. It is implemented as a separate tool and integrated to the code generation process via Java extension point.

Algorithm 4 codeGeneration(M): code generation for input model M.

```
//compute the set of deployment units
C ← getAllDeploymentUnits(M)
for all c ∈ C do
  //compute the set of KPN components mapped to the deployment unit
  A ← getMappedKPNComponents(c)
  for all a ∈ A do
    generateKPNImplementation(a, c)
  end for
  generateMainFunction(c)
  generateBuildSystem(c)
end for
```

The overall code generation procedure is illustrated in Algorithm 4. First, we analyze the input model, in particular the mapping of the application to the platform, to compute the set of deployment units required for the design. The deployment unit could be a firmware image for a bare-metal core or an OS executable. Then, for each deployment unit, we traverse again the input model to obtain the set of software components (in our case `KPNComponent` objects) mapped to it. Next, we generate the implementation for each software component under consideration of the type of the deployment unit. This part of code generation depends on the MoC used in the application model, the details of which is presented in the following sections (6.1.2). Here, usually at least one source file is created for each individual software component. Additionally, a main file is generated for the deployment unit to implement technical aspects such as initialization, finalization and communication.

One important aspect that we consider in the code generator implementation is retargetability. As the modeling approach offers a flexible environment that allows modeling of a variety of platforms, the code generator must be easily configurable to support each individual platform.

To achieve this goal, we strictly separate platform-independent code and platform-dependent code in the code templates. The platform-dependent code is specified using dedicated code templates that are tightly coupled with the platform modeling objects. For example, for each type of communication port (e.g., local memory buffer, partition-to-partition, core-to-core), specific templates are associated to specify the syntax for reading/writing/initializing the port. Using this mechanism, the code generator is implicitly configured by the platform model. Since the platform-specific code templates share a common interface and *Xpand* supports polymorphism, the appropriate templates are transparently invoked by the code generator.

During the code generation process, also the build system required to compile the application images is automatically generated. Here, we synthesize CMake [104] scripts, which can be used to create cross-platform build environments, e.g., Visual Studio projects and Eclipse CDT projects. The generation of CMake files is also based on the *Xpand* language. We provide extension point in the generated build system to allow the user to insert additional rules, e.g. extra include and library path.

Besides the source code and build system, the code generator also synthesize platform configurations to support execution of the application. The platform configuration data can either be in plain text or XML format. For the generation of plain text files, the same (template-based) approach as the source code generation can be adopted. For XML files, EMF is used to transform the underlying schema to a corresponding meta-model. The configuration generation is then implemented as a model-to-model transformation. This solution is more efficient and reliable due to type-safety of model transformation and the tool support from EMF.

One advantage of our code generation back-end is guaranteed consistency between the application software and the platform configuration. This is particularly important for MPSoC based systems that have resources shared by multiple entities. There, the application can be executed correctly only if sufficient resources are allocated. Compared with manual approaches, the high degree of automation of our code generator guarantees the overall consistency and speeds up the development process. The details of platform configuration is presented in Section 6.2.

6.1.2 Code Generation for KPN

As introduced in Section 2.1, Kahn Process Network (KPN) is used as the primary MoC for application modeling. This section discusses code generation for KPN in detail. During the discussion, we will distinguish platform-independent (functional) and platform-specific (structural)

code. A simple producer-consumer application serves as the running example. The producer generates a random integer which is printed to the console by the consumer.

KPN is a coarse-grained model that focuses on modeling the structure of the application and the interaction between components. Each KPN component represents a computational task that communicates with other tasks exclusively via messages. Our code generator creates one C source and header file for each KPN component. It contains the following subroutines:

- `pn_init()`: initialization of the KPN component. It is used to initialize objects used in the subsequent execution phase before entering the main loop. For example, the random number generator of the *producer* component is initialized and the communication ports and channels are opened.
- `pn_read()`: this function executes a blocking read on all input ports of the KPN component and stores the data into local buffers.
- `pn_fire()`: computation kernel represented by the KPN component. It processes the input tokens and stores the results to local output buffer.
- `pn_write()`: this function transfers the results from local output buffer to the output ports.
- `pn_done()`: finalization of the KPN component.

The `main` function of the deployment unit serves as driver code to invoke the above functions. The implementation of the `main` function depends on the execution model. In a time-triggered system, the `main` function sequentially invokes `pn_read()`, `pn_fire()`, `pn_write()` and `pn_done()` functions in the time slot allocated to the KPN component. In an event-triggered system, each KPN component is implemented in a separate thread. The blocking read guarantees that the execution adheres to the KPN semantics.

In the above functions, `pn_read` and `pn_write` are pure structural code, whereas the `pn_init` and `pn_fire` contain functional code of the component. In the following sections, we discuss how both parts are generated.

6.1.2.1 Functional Code Generation

The KPN model is a coarse-grained model that views tasks as black-box components. It does not cover the functional aspect of the application. Thus, we need to enrich the model with a

behavioral specification to enable full code generation. Our framework provides two different alternatives:

- The modeling framework provides a set of fine-grained meta-models with pre-defined semantics. If also the corresponding code templates are provided, functional code can be directly generated. For example, the IEC61131 [105] meta-model included in our framework can be used to create behavioral specification in terms of data-flow and state-flow models. The fine-grained models are then associated to a specific component in the structural KPN model to describe its behavior.

- If the enhanced analyzability provided by a formal application model is not required, or if legacy code needs to be integrated, the behavior of a KPN component can be described using annotated C files. This is done by marking relevant parts in the source code using pre-defined C comments introduced below.

Annotating C source code. The designer may annotate a regular C source file with the following annotations to insert part of the code to the generated software.

- /* PN_USER_CODE_INIT_START */
  The code in between is inserted to the pn_init() function of the generated code
  /* PN_USER_CODE_INIT_END */

- /* PN_USER_CODE_FIRE_START */
  The code in between is inserted to the pn_fire() function of the generated code
  /* PN_USER_CODE_FIRE_END */

- /* PN_USER_CODE_INCLUDE_START */
  The code in between is inserted to the include section of the generated code
  /* PN_USER_CODE_INCLUDE_END */

Figure 6.3 depicts an example of integrating user-supplied C code. In `pn_init()` function of the `producer`, we use annotations to insert the code that initializes the random generator. Another code piece is added to the `pn_fire()` function to generate the integer and write it to the port “out”. The designer may also include multiple code sections, as shown in the `pn_fire()` function of the consumer component.

Interface between user code and generated code. From the user code viewpoint, the interface to the generated code is the local buffer allocated to the ports. A set of auxiliary functions

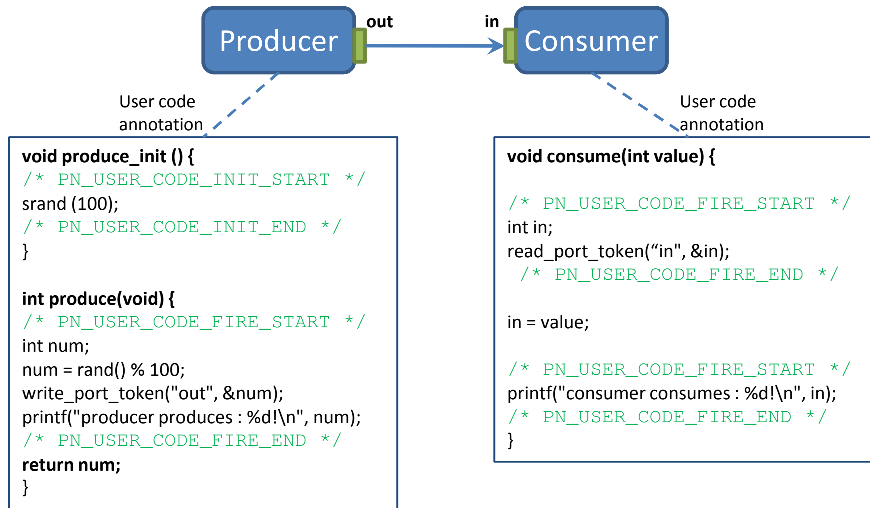


Figure 6.3: An Example for Including User Code for Code Generation

is generated to access these buffers, e.g., `read_port_token` and `write_port_token` functions used in Figure 6.3. The implementation of these functions is based information contained in the model, e.g., the data type and token size of the port. In this way, only a port name and a pointer need to be specified by the user. The auxiliary functions contribute to decouple the functional code from the actual buffer implementation.

6.1.2.2 Structural Code Generation

The structural code is platform-dependent glue code that links the functional code to constitute the complete software system. It is typically automatically derived from the design model. We illustrate the generation of structural code using inter-component communication as an example. The same producer-consumer example is used. Figure 6.4 depicts the mapping of the application to the platform model.

As it can be seen, the `ProcessingRequests` of the producer and consumer tasks are mapped to $Core_1$ and $Core_2$, respectively. The `EndpointRequests` generated from the communication ports are mapped to the `BusPort` objects. They inherit the `CommunicationEndpoint` class and are equipped with platform-specific code templates that specify the communication syntax. Finally, the `TransportRequest` is hosted by the `Bus` object. The platform objects implement a set of pre-defined API to expose the associated platform-specific code templates to the generic code generator. In this particular example, the `CommunicationEndpoint` class provides the following Java methods:

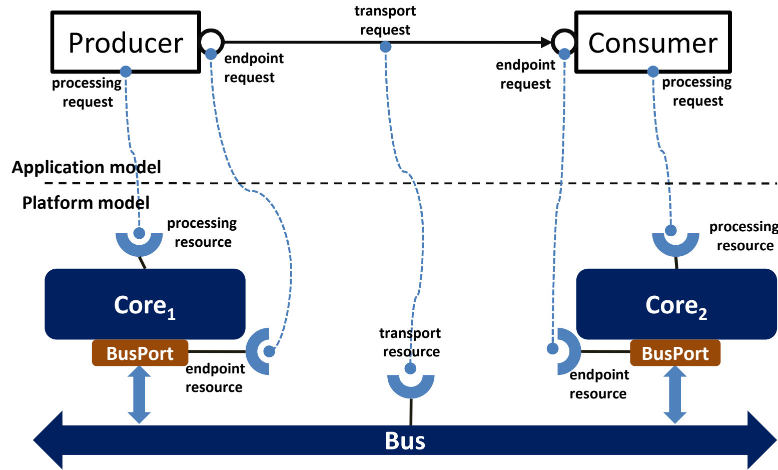


Figure 6.4: An Example Scenario for Structural Code Generation

- *generate_init()*: generates source code to initialize the port based on the platform-specific code template associated to the parent object.
- *generate_read()*: similar as above, generates code for reading a port.
- *generate_write()*: similar as above, generates code for writing a port.
- *generate_done()*: similar as above, generates code for finalizing a port.

Using the aforementioned extension mechanism provided by *Xpand*, these methods can be directly used in the code templates. As an example, Figure 6.5 shows a snippet of the code template for `KPNComponent`. In the `pn_read()` function, we iterate over all input ports of the `KPNcomponent` and generate code to read each port to the local buffer. The `getPortMapping` function (also implemented using Java extension) traces the mapping of an application port to obtain the platform object that offers the communication resource (here the `BusPort`). Afterwards, it triggers the `generate_read()` method of the platform object. At this point, the code generator delegates to the platform-specific code template. Next, the `generate_fire()` method offered by the consumer object is invoked to produce functional code for the task using approaches introduced previously. Finally, the `pn_write()` function iterates overall output ports of the component and implements the port writing.

From the above discussion, we see that the code generation for `KPNComponent` is guided by the mapping of the `Processing` and `CommunicationEndpoint` requests. The separation between platform-independent and platform-specific code templates is the key to achieve retargetability

```
«REM» code template for KPN component «ENDREM»
//read the input port
void pn_read() {
«FOREACH this.inputPorts AS port»
  «getPortMapping(this,port.name).generateRead()»
«ENDFOREACH»
}

//trigger the task
void pn_fire() {
  «this.generateFire()»
}

//write to output port
void pn_write() {
«FOREACH this.outputPorts AS port»
  «getPortMapping(this, port.name).generateWrite()»
«ENDFOREACH»
}
```

Figure 6.5: An Example for Structural Code Generation Procedure

of the code generator. In the example, based on the mapping of application ports to platform ports, the communication code is automatically adapted. One thing we have not discussed so far is the `CommunicationTransport` requests from the application channels that are mapped to the `Bus` object. This is because it is handled transparently by the framework. The transport request is typically used by the arbitrator of the resource object during platform configuration. In this particular example, the `Bus` arbitrator obtains the bandwidth demand of the channel through this relation and uses it in scheduling.

Implementation. Association of platform-specific code templates to the modeling objects is implemented using the *EOperation* framework provided by EMF. It provide mechanisms to add pre-defined methods to classes in the meta-model. Both the signature and body of the methods can be specified. Using *EOperation*, we define an API as the interface between code generator and the model. We define the interface methods in the base classes of the modeling framework and provide a default implementation. For example, the base class `CommunicationEndpoint` contains a default implementation of port access using the standard POSIX C API. Derived classes that require a platform-specific implementation, e.g. `BusPort`, can overload the interface functions to indicate their own communication syntax. Polymorphism of Java guarantees that the overloaded methods are correctly selected by the code generator. Such an implementation contributes to extensibility of the code generation framework. New target platform can be supported straightforwardly by implementing the pre-defined API in the according meta-model.

6.2 Platform Configuration

In MPSoC based systems, successful execution of the application requires matching configuration of the hardware platform. The configuration can be performed at runtime (dynamic configuration) or at design time (static configuration). For our target application domain, static configuration is preferred for the sake of predictability. Hence, the DSE framework presented in previous chapters focuses on static optimization of the configuration parameters. The back-end of our approach transforms the design parameters into concrete platform configuration files.

Besides others, the most import aspect of platform configuration is the arbitration of shared resources, including processor time, communication media, memory and I/O. The modeling framework provides a generic interface to integrate configuration tools. For each platform object that offers `CapabilityResources`, an arbitrator object can be specified. The arbitrator gathers requests from the application model and uses a pre-defined API to schedule and allocate resources to serve the requests. By implementing the interface API, platform configurators can be integrated to the `CapabilityResource` classes.

The implementation of platform configurator is highly specific to the target architecture. As an example, we present in this section the design of a configurator for the Time-Triggered Network-on-Chip (TTNoC). The TTNoC provides predictable inter-core communication and is the core of the ACROSS MPSoC. We start with presenting the scheduling algorithm for TTNoC, followed by simulation results. Afterwards, we discuss how the TTNoC configurator is integrated to the overall reliability-aware design approach.

6.2.1 Configuring a Time-Triggered Network-on-Chip

Time-Triggered (TT) communication is a natural and efficient way to provide reliable and predictable communication for safety-critical embedded systems. In TT networks, the communication entities are synchronized with each other. Traffic is injected strictly adhering to the predefined schedule and resource collision is avoided by design. Examples of time-triggered protocols include Flexray (the static segment) in the automotive industry, SAFEBus and TTP in the avionics domain, and TTEthernet being an extension of the classical Ethernet [106].

The traditional time-triggered protocols usually operate on bus-like systems. The shared communication media is organized in time slots and all messages are separated in the time domain. However, bus-based systems cannot meet the communication requirements of modern Multiprocessor System-on-Chip (MPSoC) platforms [107] due to the bandwidth limitation.

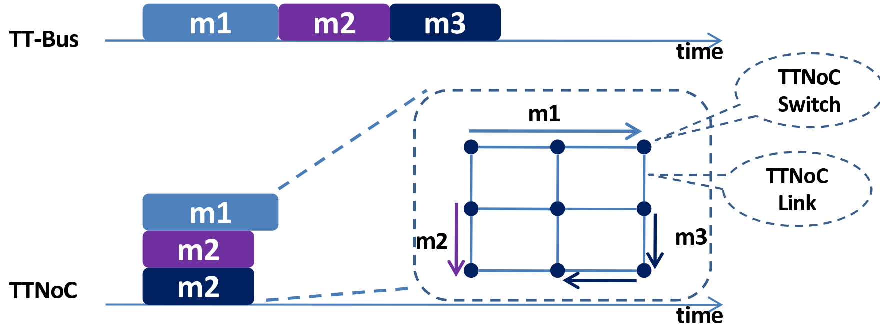


Figure 6.6: A TTNoC Scheduling Scenario Example

Researchers therefore investigated the integration of time-triggered communication in Networks-on-Chip (NoC) and proposed the TTNoC architecture [108]. TTNoC is based on a network of on-chip switches. Although the network is globally arbitrated in time, a major advantage of TTNoC is the possibility to separate messages in the space domain, i.e. messages can share the same time-slot as long as their routes are non-overlapping. An example is depicted in Figure 6.6.

The configuration TTNoC is a two-fold problem: message scheduling in the time domain and routing in the space domain. Our configuration approach is based on Satisfiability Modulo Theories (SMT) solving. An SMT solver accepts problems formulated in first-order logic and checks the feasibility of a solution with respect to the background theories. We first present a specification to formulate the complete problem as an SMT instance (Section 6.2.1.4). This approach always computes a feasible solution if one exists. Since the solving time may become unacceptable as the problem size grows, we develop an incremental algorithm to improve the scalability (Section 6.2.1.5).

6.2.1.1 Related Work

The scheduling problem specific to TTNoC has not been studied in existing literature. However, our work is closely related to the scheduling approaches on other time-triggered architectures. In [109] the authors present an approach for the scheduling of static segment of Flexray using Integer Linear Programming (ILP). *Lukasiewycz et al* propose a transformation of the Flexray scheduling into a bin-packing problem and solve it subsequently using ILP [110]. To increase the effective bandwidth, the concept of switched Flexray is proposed [111]. The corresponding scheduling problem is studied in [112, 113]. Their solutions are based on the branch-and-price algorithm [112] and graph-based heuristics [113]. The scheduling problem for time-triggered

multi-hop networks is consider in [114]. The author adopts a similar approach as ours, namely a pure SMT formulation followed by an incremental method to enhance the scalability. One major different between [114] and our work is that the routes of all messages are known in [114] and the author focuses only in the time domain.

6.2.1.2 Problem Definition and Transformation

The TTNoC architecture consists of a set of fragment switches (also called *nodes*). A switch offers four identical *ports* as depicted in Figure 6.7. Each port-to-port connection consists of one link per direction (full-duplex). A port can connect to another switch or a Processing Element (PE) via the Trusted Interface Sub-System (TISS). The unified interface of switches and TISSs allows the designer to implement different topologies with low effort. A set of routes may co-exist as long as no two routes use the same link, e.g., in Figure 6.7, the messages m_0, m_1 and m_2 can co-exist whereas m_3 collides with m_1. The switches are not aware of the communication schedule and just forward the message from the input port to the output port according to the routing information contained in the message header. The latency of forwarding is constant.

The payload of a message is decomposed into a set of fixed-size *flits*, which is the basic transmission unit in TTNoC. A flit is handled by a switch in one *system clock cycle*. The TTNoC is globally arbitrated using TDMA. The granularity of the TDMA slots is called a *macro tick*. A macro tick is a multiple of the system clock cycle, i.e. multiple flits can be sent in one slot. The duration of a macro tick is restricted to be a negative power of a physical second by design [108], e.g., $\frac{1}{2}$ or $\frac{1}{4}$ second. The time slots are allocated statically to each communication entity and the information is stored in each TISS. The TISS abstracts the details of communication away from the application side. The TISSs are synchronized in macro tick, i.e. all communication activities are aligned to the TDMA slots. In the remainder of the paper, macro tick is used as the basic unit of time.

We focus on periodic messages as they are typical in the target application domain. A message is described as a four-tuple (s, t, p, l), where s is the message source, t is the message sink, p is the period and l is length of the message. According to the timing specification of TTNoC, the period must be a positive power of two of macro ticks ($p \in \{2^n | n > 0\}$), i.e. the messages are **harmonic**. The length l is the number of TDMA slots needed to transmit the message. To guarantee collision freedom, all flits should reach their destination before the end of the allocated time interval. Let *payload* be the message payload in terms of flits, x be the number of hops in the route, d be the delay per hop and T be the number of flits per macro tick,

the total number of TDMA slots needed by a message can be computed as $l = \left\lceil \frac{payload+xd}{T} \right\rceil$. As can be seen, the message length depends on the routing. This dependency causes a correlation between the time and space domain in the scheduling problem and it increases the complexity significantly. To cope with this problem, we introduce a restriction on routing. Let the distance (in hops) between source and the target TISS of a message m be D_m. We restrict the routing algorithm to explore routes with a maximum of $D_m + \alpha$ hops, where $\alpha \geq 0$ specifies the *flexibility* of the routing. By doing so, l can be over-estimated by $l = \left\lceil \frac{payload+(D+\alpha)d}{T} \right\rceil$.

The TTNoC scheduling problem can be stated as follows. Given an architecture with a set of nodes N and links B, a set of communicating PEs C and a set of messages M, determine: 1) the PE-to-switch allocation π, i.e. the PE c joins the network via a port of the switch $\pi(c)$, 2) the timing offset (or *phase*) f of each message, 3) the path P for each message, such that each two messages are separated either in the time or in the space domain. A message with period p and phase f occupies the time intervals $[np + f, np + f + l]$ with $n \in \mathbb{N}_0$. The path P must be a continuous route from the source TISS s to target TISS t. Since the message periods are always positive powers of two, the hyper-period of any set of messages is the longest period of all messages. We denote the hyper-period using p_{max}. Without loss of generality, it is sufficient to schedule only the first hyper-period.

6.2.1.3 Problem Transformation

The problem of allocating messages into time intervals can be transformed into a 2D bin-packing problem. Here we adopt the transformation proposed in [110] and adapt it to our needs. A brief outline is given in the following.

Assume the shortest period of all messages in M is p_{min}. The periods of all messages can be represented as positive powers of two times p_{min}, i.e. for message m, $p_m = r_m p_{min}$, where $r_m \in \{2^n | 0 \leq n \leq p_{max}/p_{min}\}$ is the *repetition* factor. The time line of a hyper-period can be divided into *segments* of size p_{min} and viewed in a 2D fashion as shown in Figure 6.8a. Each message will appear in every r segments, e.g., m_1 appears in every segment and m_3 appears in every 4 segments. To transform message scheduling to bin-packing, each message is converted into a rectangle element. The size of the element can be computed by:

- $h_m = l_m$: the height of element is the length of message.
- $w_m = p_{max}/p_m$: the width of element is the number of appearances in a hyper-period.

Obviously, the widths of elements are always powers of two. The size of the bin is:

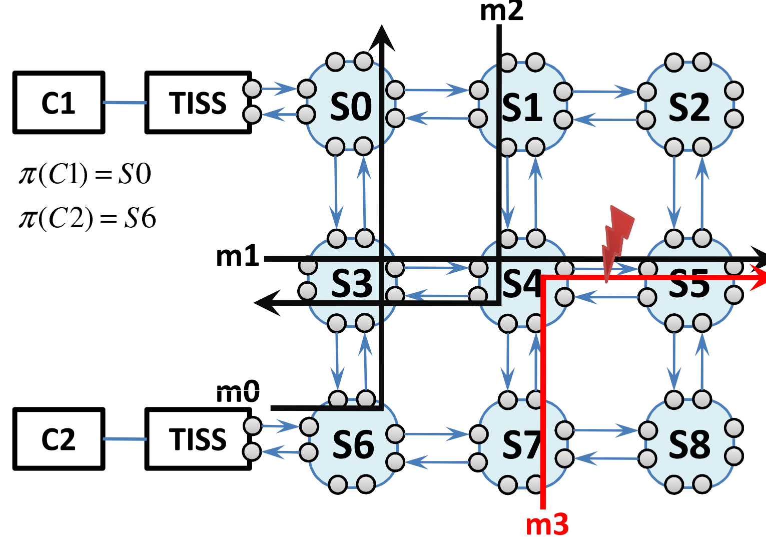


Figure 6.7: The TTNoC Architecture

- $H = p_{min}$: the height of the bin is p_{min} in macro ticks.
- $W = p_{max}/p_{min}$: the width of the bin is the number of segments of size p_{min} in one hyper-period.

Figure 6.8b depicts an example. Bin-packing is about placing the elements in appropriate locations inside the bin, such that no two elements intersect. The placement of an element is defined by offsets $x_m \in [0, W)$ and $y_m \in [0, H)$ in horizontal and vertical directions. Note that the horizontal offset x_m must be a multiple of the width, i.e. $x_m \in \{nw_m | n \in \mathbb{N}_0\}$. The placement of elements in the bin-packing problem has a one-to-one mapping to the allocation of messages in time intervals. The phase of a message m can be calculated from the position of the rectangle element as:

$$f_m = b_m p_{min} + y_m \tag{6.1}$$

$$b_m = t(\frac{x_m}{w_m}, \frac{W}{w_m}) \tag{6.2}$$

where t is the transformation function defined as:

$$t(x, y) = \begin{cases} 0 & \text{x=0} \\ t(\frac{x}{2}, \frac{y}{2}) & \text{x is even} \\ t(\frac{x-1}{2}, \frac{y}{2}) + \frac{y}{2} & \text{x is odd} \end{cases}$$

$$with \quad x \in \mathbb{N}_0, y \in \{2^n | n \in \mathbb{N}_0\}, 0 \leq x < y$$

Here b_m denotes the segment in which message m appears. The vertical position y_m denotes the offset within the segment, e.g., the offset of m_3 can be computed by $f_{m_3} = t(\frac{2}{2}, \frac{4}{1})p_{min} + h_{m_1} =$

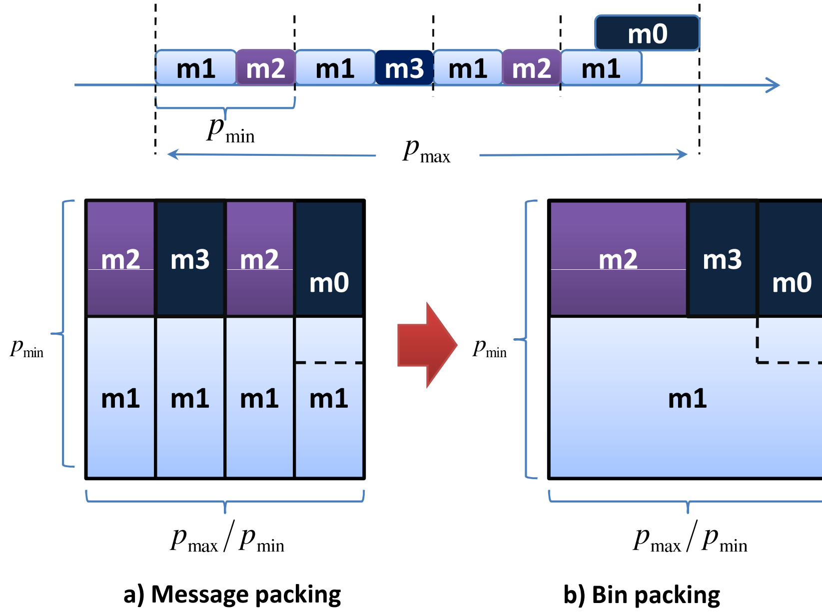


Figure 6.8: Message Scheduling to Bin Packing Transformation

$p_{min} + l_{m_1}$. Using the transformation above, a feasible message schedule exists if and only if a feasible bin-packing scheme exists [110]. Since the bins are of the height p_{min}, only messages shorter than p_{min} can fit inside. Thus, if a message is longer than p_{min}, it has to be broken into several pieces. Figure 6.9 illustrates an example, in which m_4 occupies three time segments of size p_{min}. Those pieces can be scheduled individually. Additional constraints are needed to make sure all pieces follow the same route and are continuous in time (see Section 6.2.1.4).

The bin-packing problem transformed from TTNoC scheduling is not a standard one. The major difference is that the intersection of objects is allowed, as long as the collision can be resolved in the space domain (e.g., m_0 and m_1 in Figure 6.8 can be assigned to non-overlapping routes and share the same time slot). New approach is needed the address this issue.

6.2.1.4 SMT Specification

This section describes the formulation of the TTNoC scheduling problem as an SMT specification. We first introduce the used variables and then proceed with the constraints that apply on the variables.

Variables. To describe the PE-to-switch allocation, we enumerate all available ports that a PE can attach to and place a *virtual component* on each port. The PEs are then mapped to

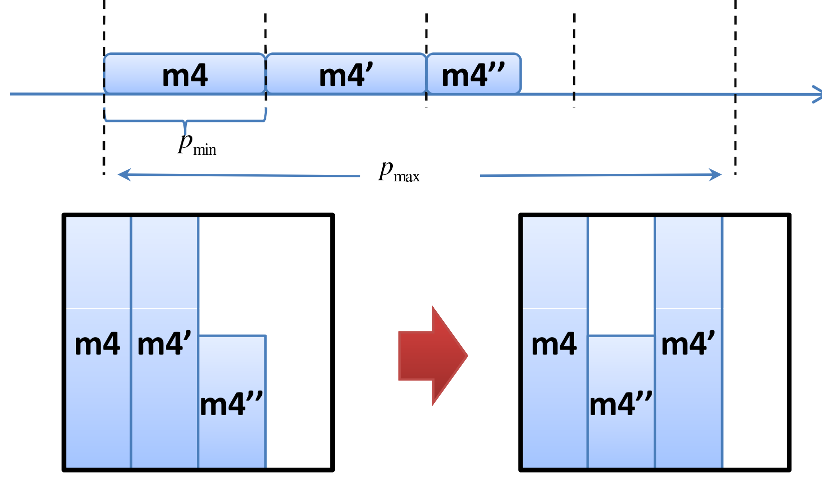


Figure 6.9: Segmentation of Long Messages

the virtual components. We use $node(v)$ to denote the switch that offers the port for virtual component v. A set of binary variables is defined:

- $\mathbf{a}_{c,v} \in \{0,1\}$ is 1 iff the PE c is mapped to virtual component v.

For each message m, two sets of binary variables are used to denote the route:

- $\mathbf{q}_{m,n} \in \{0,1\}$ is 1 iff switching node n is on the path of message m and 0 otherwise.
- $\mathbf{k}_{m,i,j} \in \{0,1\}$ is 1 iff link (i,j) is on the path of message m and 0 otherwise.

The following variables specify the location of the message in the bin:

- $\mathbf{x}_m \in \{nw_m | n \in \mathbb{N}_0, n < \frac{W}{w_m}\}$ is the horizontal offset of m.
- $\mathbf{y}_m \in \{y \in \mathbb{N}_0, y < H\}$ is the vertical offset of m.

Path Constraints. We introduce a set of constraints to make sure that the variables $\mathbf{q}$ and $\mathbf{k}$ denote a continuous, acyclic path from the source to the destination. If a node is on such a path, exactly one of its input links and exactly one of its output links should be used (see Figure 6.7 for example). Hence, the general path constraints are:

$$\forall n \in N : \mathbf{q}_{m,n} \rightarrow (one_in \wedge one_out) \quad \text{where}$$
$$one_in = (\sum_{i \in in(n)} \mathbf{k}_{m,i,n}) = 1$$
$$one_out = (\sum_{j \in out(n)} \mathbf{k}_{m,n,j}) = 1$$

Here $out(n)$ is the set of switches reachable from the output links of n, and $in(n)$ is the set of switches that the input links of n originate from. To guarantee that the path starts and ends at the correct nodes, the following constraints need to be enforced:

- The switch that attaches the message source s_m and the one that attaches the message target t_m must be in the path of the message.

$$\mathbf{a}_{s_m,v} = 1 \rightarrow \mathbf{q}_{m,node(v)} = 1$$
$$\mathbf{a}_{t_m,v} = 1 \rightarrow \mathbf{q}_{m,node(v)} = 1$$

- Also, the link that connects the source/target of a message to the network has to be used:

$$\mathbf{a}_{s_m,v} = 1 \rightarrow \mathbf{k}_{m,v,node(v)} = 1$$
$$\mathbf{a}_{t_m,v} = 1 \rightarrow \mathbf{k}_{m,node(v),v} = 1$$

If a link is on the path, the nodes on the two ends must also be on the path:

$$\forall (i,j) \in B : \mathbf{k}_{m,i,j} = 1 \rightarrow \mathbf{q}_{m,i} = 1 \wedge \mathbf{q}_{m,j} = 1$$

Dummy loops that go from node i to j and immediately back should be avoided:

$$\forall (i,j) \in B : \mathbf{k}_{m,i,j} = 1 \rightarrow \mathbf{k}_{m,j,i} = 0$$

The overall route length should be below the upper bound:

$$\begin{aligned} \forall v_1, v_2 : (\mathbf{a}_{s_m,v_1} = 1) \wedge (\mathbf{a}_{t_m,v_2} = 1) \rightarrow \sum_{(i,j)\in B} \mathbf{k}_{m,i,j} \\ \leq distance(node(v_1), node(v_2)) + flexibility \end{aligned}$$

Non-Overlapping constraints. If two messages intersect in the bin-packing, non-overlapping routes must be assigned to them. This constraint is denoted as following:

$$\begin{aligned} &\forall m_1 \in M, m_2 \in M, m_1 \neq m_2 : \\ &overlap(m_1, m_2) \rightarrow \bigwedge_{(i,j)\in B} \neg(\mathbf{k}_{m_1,i,j} = 1 \wedge \mathbf{k}_{m_2,i,j} = 1) \end{aligned}$$

where the overlap occurs if and only if:

$$\begin{aligned} &overlap(m_1,m_2) = \\ &\qquad (\mathbf{x}_{m_1} < \mathbf{x}_{m_2} + w_{m_2}) \wedge (\mathbf{x}_{m_2} < \mathbf{x}_{m_1} + w_{m_1}) \\ &\qquad \wedge (\mathbf{y}_{m_1} < \mathbf{y}_{m_2} + h_{m_2}) \wedge (\mathbf{y}_{m_2} < \mathbf{y}_{m_1} + h_{m_1}) \end{aligned}$$

Problem Specific Constraints. The SMT formulation can also incorporate problem specific constraints. For example, in the current specification of TTNoC, the TISS can only transmit or receive one message at a time. This means two messages with the same source or target must be separated in time. This constraint can be written as:

$$\forall m_1 \in M, m_2 \in M, m_1 \neq m_2 :$$
$$s_{m_1} = s_{m_2} \vee t_{m_1} = t_{m_2} \rightarrow \neg overlap(m_1, m_2)$$

As discussed in section 6.2.1.3, a long message may need to be broken into several pieces to fit into the bin. Those pieces must share the same path and be continuous in time. Let m' and m'' be two successive pieces of a message, then the following constraints must be enforced:

- the piece m'' appears one segment later than m' : $t(\frac{\mathbf{x}_{m'}}{w_{m'}}, \frac{W}{w_{m'}}) + 1 = t(\frac{\mathbf{x}_{m''}}{w_{m''}}, \frac{W}{w_{m''}})$,
- the offset within segment is 0 if it is not the first piece: $\mathbf{y}_{m''} = 0$,
- the same links are used: $\forall (i, j) \in B : \mathbf{k}_{m',i,j} = \mathbf{k}_{m'',i,j}$.

6.2.1.5 Heuristic Approach

In most cases, message scheduling is only one part of the design process and needs to be carried out multiple times. However, as the problem size increases, the long execution time of a purely SMT based approach might become a hurdle for the designer. To cope with this problem, we propose an incremental heuristic to improve the scalability. The algorithm proceeds in three steps as detailed in the next section: 1) PE-to-switch allocation, 2) classical strip packing, 3) level packing. The general idea of the heuristic is to reuse the existing bin-packing algorithms to place the objects (step 2) and rely on the SMT solver to handle the non-standard constraints, i.e. overlapping of objects (step 3).

1) PE-to-switch allocation. The goal of this step is to find a PE-to-switch allocation scheme π that minimizes the communication cost estimated by the distance between source and target nodes:

$$Minimize : cost = \sum_{m \in M} \frac{l_m}{p_m} * distance(\pi(s_m), \pi(t_m))$$

For that we adopt an Evolutionary Algorithm (EA) based optimization approach. The algorithm takes an architecture graph G_A and a communication graph G_C (Figure 6.10) as input. The architecture graph is a full-meshed graph, whose vertices correspond to virtual components

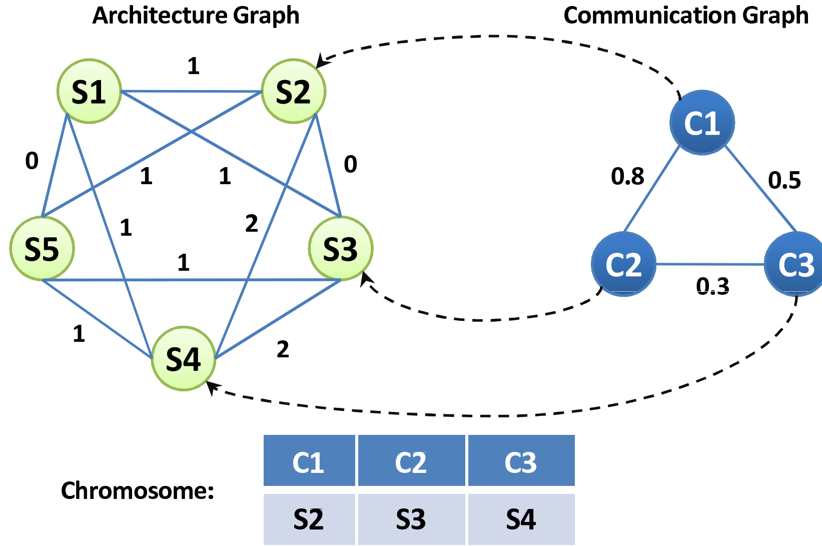


Figure 6.10: PE-to-Switch Allocation Optimization

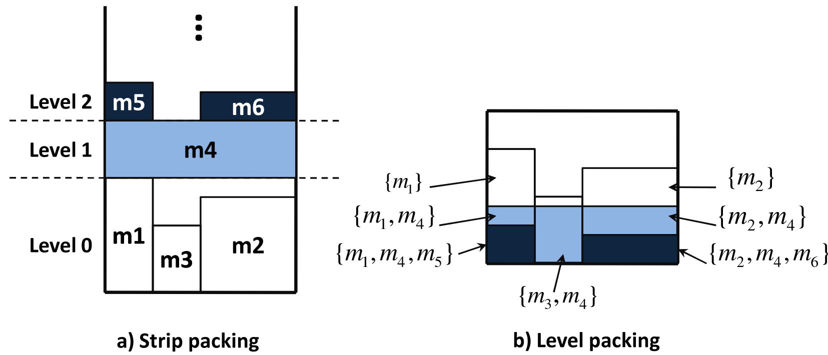


Figure 6.11: Example of Strip Packing and Level Packing

in the TTNoC and edges specify the distance between virtual components in hops. The communication graph is also a full-meshed graph. Its vertices represent PEs and its edges describe the communication requirements between any two PEs. The communication requirements are computed by:

$$R(C_1, C_2) = \sum_{m \in M \wedge ((s_m = C_1 \wedge t_m = C_2) \vee (s_m = C_2 \wedge t_m = C_1))} \frac{l_m}{p_m}$$

A PE-to-switch allocation maps each vertex of G_C to one vertex of G_A. This mapping can be encoded as a list of integers as depicted in Figure 6.10. Standard operators such as two-point crossover can be adopted during the EA-based optimization process.

2) Strip Packing. This step packs the objects into an imaginary strip with infinite height as

illustrated by Figure 6.11a. This problem has been extensively studied in the past (cf. [115]). Because of the good performance and simplicity in implementation, we adopt the First Fit Decreasing Height Decreasing Width (FFDHDW) algorithm in this work. In principle, any other existing algorithms could be used as well. The FFDHDW algorithm belongs to the category of *level algorithm*. These algorithms enforce the restriction that the objects are placed with the lower edge on certain horizontal levels [115]. The height of a level is determined by the height of the tallest object in that level. Using FFDHDW, the objects are pre-ordered in non-increasing height, and when height equals, non-increasing width. The objects are iteratively placed onto the lowest level with sufficient space and a new level is created on top of the current level if it does not fit any existing level. Recall that the objects transformed from the TTNoC messages are restricted to be placed into horizontal locations that are multiples of their width.

3) Level Packing. In this step, the levels are considered as one-dimensional objects with size equal to the height of the level and packed into bins (Figure 6.11b). As previously discussed, messages can overlap in time as long as a spatial separation is guaranteed. Thus, we try to overlay different levels to reduce the overall height of the strip. An outline of the level packing algorithm is presented in Algorithm 5. After strip packing, we iteratively place levels in decreasing height into the bin. The vector *locations* contains the possible vertical locations to place the level. We try to place the level in the lowest possible location (line 4). If it is successful, the position and routing of the messages contained in this level will be computed and fixed later on. The top of the current level is considered as a possible location for future levels (line 5). This procedure is demonstrated in Figure 6.12. If a level fails at all locations, the messages are added to the *failedMessage* set and we move on with the next level (line 10).

The feasibility of placing a level at a certain location is checked by the SMT solver. Since packing and routing of existing messages in the bin are fixed, the corresponding SMT variables are replaced by constants in the constraints to simplify the formulation. The constraints to check if level l can be placed in location x is:

$$\begin{aligned}&\forall m \in l, m' \in existingMessages(x):\\&overlap(m, m') \rightarrow \bigwedge_{(i,j)\in B} (\mathbf{k}_{m',i,j} = 1 \rightarrow \mathbf{k}_{m,i,j} = 0)\end{aligned}$$

The SMT solver can be granted the freedom to change the horizontal location of messages inside levels. In many cases, only certain combination of messages causes an unroutable case. It will be much more efficient to resolve these conflicts by moving the messages in the horizontal direction than by placing levels at different locations. For example, if messages m_5 and m_1 in

Algorithm 5 IncrementalMessagePacking(): iterative level based bin-packing with intersection of objects allowed. The function *place* uses SMT solver to check the feasibility. M: the set of messages.

```
1:  locations = {0};
2:  for all l ∈ levels with decreasing height do
3:    for all a ∈ locations in increasing order do
4:      if place(l,a)=success then
5:        locations = locations ∪ {a + height(l)}
6:        break;
7:      end if
8:    end for
9:    if l fails at all locations then
10:     addToFailedMessages(l)
11:   end if
12: end for
13: for all m ∈ failedMessages do
14:   place(m)
15: end for
```

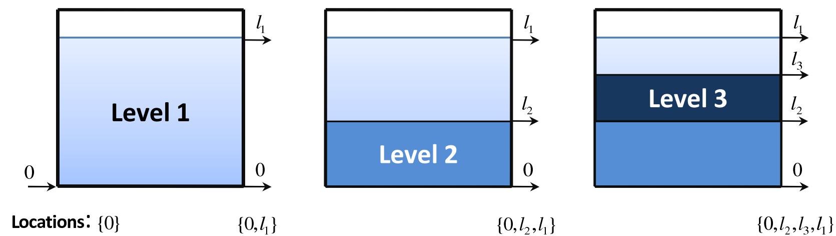


Figure 6.12: Example of Level Packing

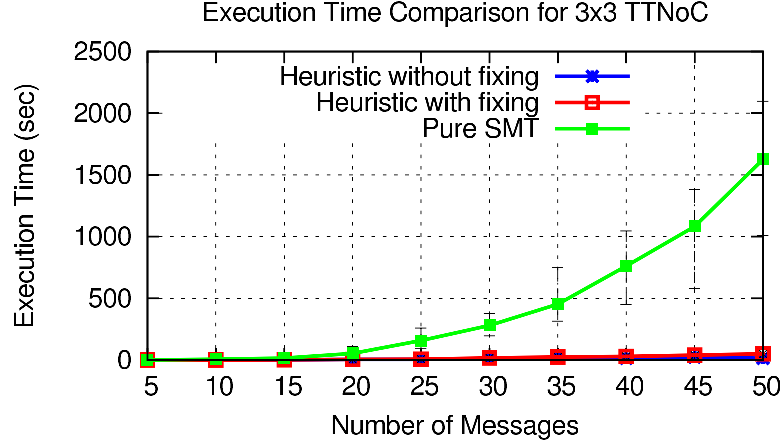


Figure 6.13: Execution Time Comparison: 3x3 NoC

Figure 6.11 cannot be routed together (e.g., they are from the same sending TISS), the area $\{m_1, m_4, m_5\}$ becomes unroutable and *level* 2 cannot be placed at location 0. However, this can be resolved by moving m_5 to the right, e.g., to the same location as m_3. After placing all levels, an additional fixing step can be introduced, which tries to allocate the failed messages (line 15 to 17 in Algorithm 5). We iteratively consider all messages and give the SMT solver the full freedom to change the location in both horizontal and vertical axis, i.e. messages are not restricted to any levels. Since only the variables associated with a single message need to be computed, such a problem is expected to be solved in a short time. Our experimental results verify this assumption.

6.2.1.6 Experiments

The SMT formulation and incremental heuristic are implemented in JAVA using the Z3 SMT solver [116]. The program is running on a Windows machine with 3GHz CPU and 4GB memory. We tested the scheduling algorithms on three architectures, namely a 3x3 TTNoC with 9 switching nodes, a 5x5 architecture with 25 nodes and a 7x7 architecture with 49 nodes. Three algorithms are compared:

- Pure SMT formulation (*SMT*).
- Incremental heuristic without fixing failed messages (*H-NoFix*).
- Incremental heuristic with the fixing step (*H-WithFix*).

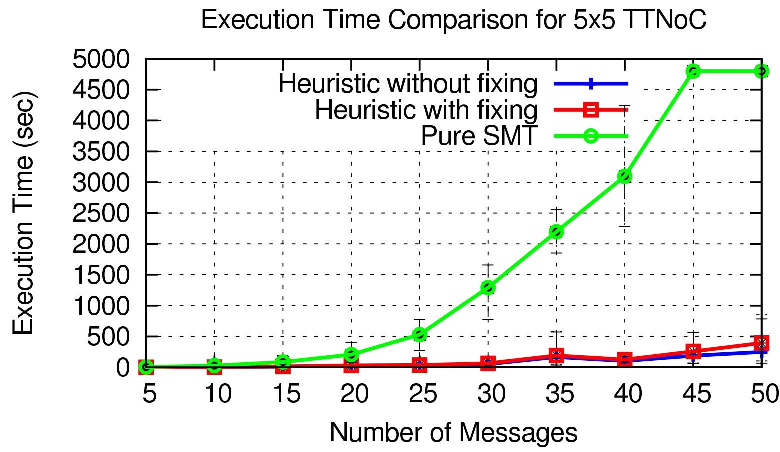


Figure 6.14: Execution Time Comparison: 5x5 NoC

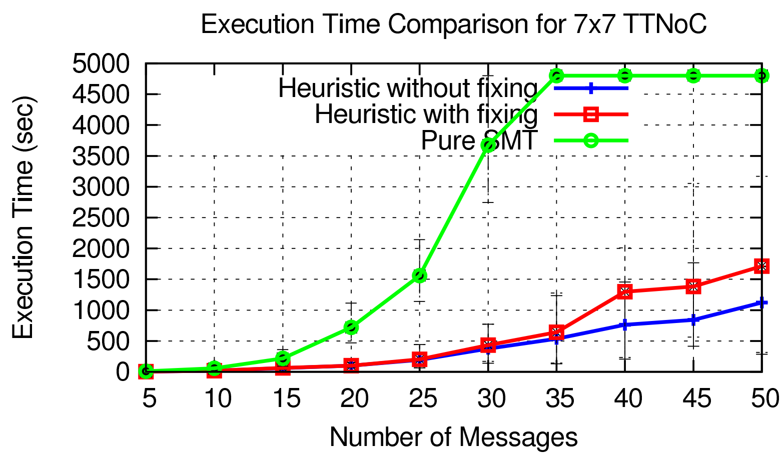


Figure 6.15: Execution Time Comparison: 7x7 NoC

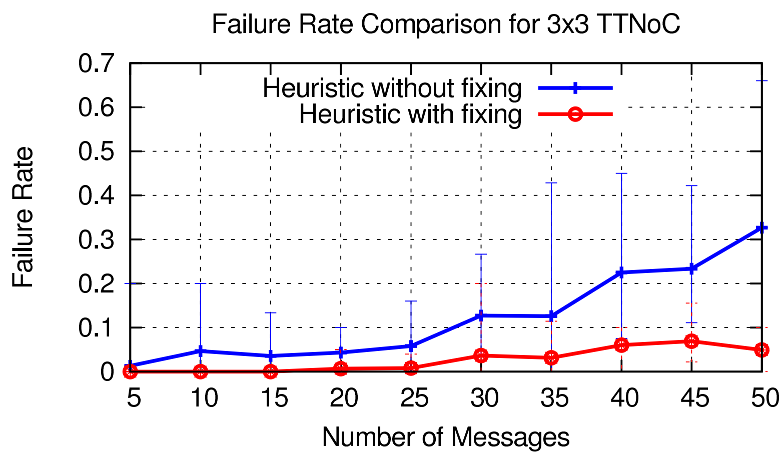


Figure 6.16: Error Rate of Heuristic Algorithms: 3x3 NoC

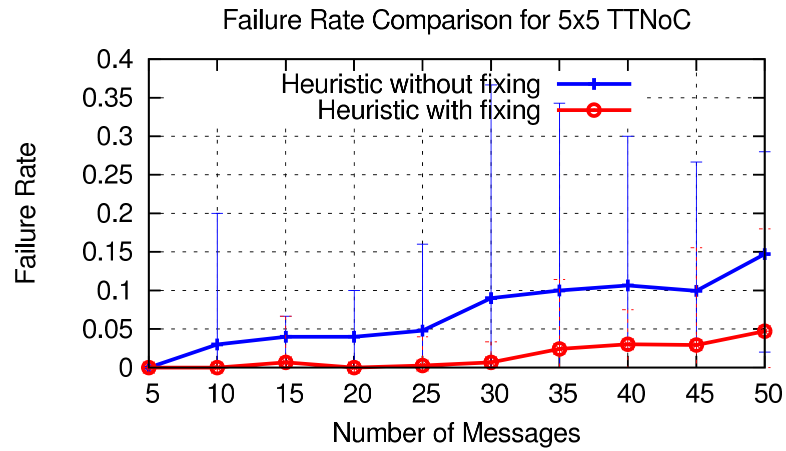


Figure 6.17: Error Rate of Heuristic Algorithms: 5x5 NoC

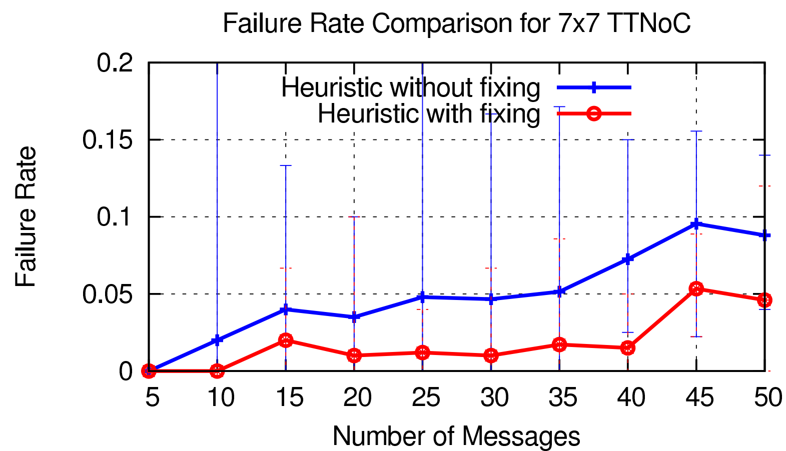


Figure 6.18: Error Rate of Heuristic Algorithms: 7x7 NoC

We generate messages with periods between $32\mu s$ and $32ms$ and random length. For each architecture, a random set of 5 to 50 messages is generated and allocated. We execute 15 such test cases and compute the average results. In each round, the execution is terminated if the runtime exceeds 1.5 hour. Figure 6.13 to 6.15 compares the runtime of the three algorithms. The error bars depict the best and worst results obtained. As can be seen, the heuristic algorithm scales far better than the pure SMT solution. In case of 40 messages, a speedup of $58x$ and $30x$ is achieved by *H-NoFix* for the 3x3 and 5x5 TTNoC, respectively. The pure SMT solution exceeds the 1.5 hour budget as the number of messages approaches 45 for the 5x5 architecture and 35 for the 7x7 architecture. The execution time of *H-WithFix* is very close to the *H-NoFix* algorithm. The extra time spent on fixing increases with the total number of messages. This is due to the fact that more messages fail as the problem becomes more difficult.

To evaluate the performance of the heuristic, we consider the percentage of messages failed to be allocated using the algorithm. Figure 6.16 to 6.18 presents the results. It can be seen that the failure rate generally decreases as the architecture size becomes larger. For the case of 50 messages, failure rates of 33%, 14% and 9% respectively are observed by *H-NoFix*. A very likely explanation is that mapping the same amount of messages to a larger architecture is generally a simpler problem due to more routing options. The *H-WithFix* algorithm further improves the performance. Again for the 50 message case, the failure rate is reduced to round 5% after the incremental fixing step. In Table 6.1, we compare the number of successful/failed/expired test runs for some representative setups. A test case is considered as successful if all messages are allocated and failed if at least one message cannot be allocated. For relatively simple test cases (e.g., tests with less than 25 messages), the success rate of the heuristic (*H-WithFix*) is comparable with that of *SMT*. For larger tests (e.g., those with 50 messages), the success rate of heuristic is relatively low. Nevertheless, only a small portion of messages (less than %5) cannot be allocated. The designer may manually map the remaining messages or explore a larger architecture. As the counterpart, the *SMT* approach fails to provide a result due to expiration of time budget.

6.2.2 Integration of TTNoC Configurator

The platform configuration tools are typically integrated as part of the back-end of the our framework and runs in parallel with the code generator. Following the waterfall design flow, they takes the model resulted from the DSE phase as input and synthesize configuration files. In other words, they have read-only access to the model. The TTNoC scheduler is different

		SMT			Heuristic			
arch	num. mess.	#succ. case	#fail. case	#exp. case	#succ. case	#fail. case	#exp. case	#failed mess.(%)
3x3	25	13	2	0	13	2	0	0.8
5x5	25	15	0	0	14	1	0	0.3
7x7	25	15	0	0	12	3	0	1.2
3x3	50	13	2	0	1	14	0	4.9
5x5	50	0	1	14	3	12	0	4.7
7x7	50	0	1	14	2	13	0	4.6

Table 6.1: Comparing the Number of Successful/Failed/Expired Tests

from a regular configuration tool, because it determines part of the design parameters, namely the scheduling/routing of messages. These parameters play an important role in the timing property of the application. For this reason, the TTNoC scheduler is not only integrated as a configuration back-end but also as part of the DSE. On the one hand, the redundancy scheme determines the amount of messages needs to be scheduled. On the other hand, the results of TTNoC scheduling are needed to evaluate the timing requirements such as end-to-end latency.

As presented in Section 4.3, our DSE framework involves a scheduler that generates time-triggered schedules for combined tasks and messages based on the partial design provided by the optimizer. To avoid misunderstanding, this scheduler is referred to as *task scheduler* in the rest of this section, although it considers communication as well. The task scheduler has to interact with TTNoC scheduler to achieve a complete design. The interaction depends on the scheduling model. In our approach, we consider two possible scenarios described as follows.

Asynchronous task and message scheduling. In this scheduling model, the processors and network do not operate on a common time base. Although they both may execution in a time-triggered manner, the phase of tasks and messages are not aligned by design. Scheduling of tasks and messages are typically performed independently using dedicate tools. Having the individual task and message schedule, analysis tools such as MPA and SYMTA/S can be adopted to evaluate the timing properties of the entire system, such as end-to-end latency and jitter. These tools typically provide the best-case and worst-case results of the system.

Synchronous task and message scheduling. This scheduling model requires the processors to synchronize with the network. Task and messages are scheduled jointly. Data dependency between tasks and messages is respected. The phase of tasks and messages can be

arranged carefully to optimize the timing property of the system, e.g., minimizing the end-to-end latency and jitter.

Asynchronous scheduling has the advantage of simplified implementation, since no complex synchronization layer between processors and the network is required. Moreover, it might be preferred by the industry due to organizational issues. Take automotive industry from example, the bus system such as Flexray is shared by many functions in the vehicle and serves as an integration point. The individual functions, such as Adaptive Cruise Controller or Driver Assistant, are developed independently by different suppliers. Asynchronous scheduling allows the suppliers to design their systems in a relatively stand-alone scenario, reducing the management overhead. The OEMs can integrate the functions by collecting the requirements of subsystems and performing scheduling of the shared resources.

As the counterpart, synchronous scheduling has its advantage mainly in performance. First of all, by optimizing the timing behavior of the system, the performance of the applications can be improved. For example, the work [117, 118] shows how control performance in automotive applications can be optimized by scheduling techniques. Second, synchronous scheduling achieves higher bus utilization. This is essential for extensibility and robustness of the system, since more space can be reserved for future applications. Last but not least, this model can guarantee timing predictability which is of utmost importance for safety-related systems. For these reasons, although synchronous scheduling requires higher implementation effort, it is believed to be one viable solution in future systems. The recent Flexray 3.0 standard adds the support for synchronous scheduling.

The TTNoC configuration can be integrated to support both scheduling models. In asynchronous scheduling, the TTNoC configuration can be used as it is. The message specification (period, length, source and target core, etc) is extracted from the design model and submitted to the TTNoC configurator. On the one hand, the results of TTNoC scheduling can be directly used to configure the network. On the other hand, the results are combined with the task schedule to build an analysis model to evaluate the timing and other extra-functional properties of the system.

In synchronous scheduling mode, the TTNoC configurator is integrated into the task scheduler to enable joint message and task scheduling. More precisely, it is used to check if it is valid to schedule a message at a certain point. Unlike bus-based systems, where messages must be simply separated in time, TTNoC allows messages to be scheduled into the same time slot as long as a non-overlapping path can be allocated for them. We show this process using an

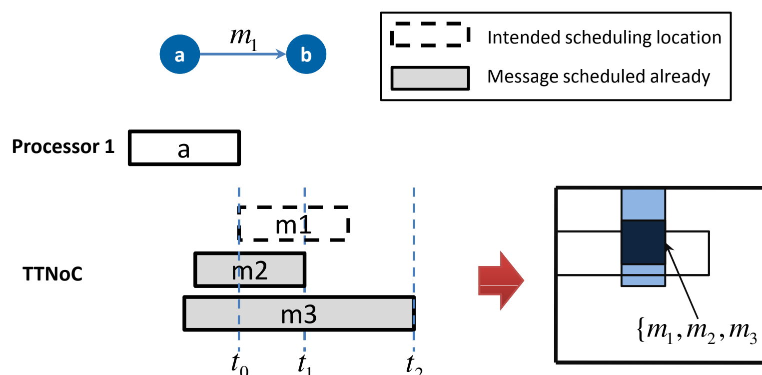


Figure 6.19: Feasibility Check Using TTNoC Configurator

example depicted in Figure 6.19. Assume the scheduler has just scheduled task a and is now considering message m_1. The intended location of m_1 is at t_0, right after the source task. At this point, other messages might have already been scheduled on the TTNoC, e.g., m_2 and m_3 in the figure. The TTNoC configurator is needed to check if m_1 can be scheduled at t_0. Since m_1 overlaps with existing messages in time, a non-conflicting route must exist.

The SMT formulation introduced in Section 6.2.1.4 can be used to solve this feasibility problem. Only messages that overlap with m_1 in time need to be considered. The offsets and routes for existing messages are considered as constant in the formulation. The only variables are the $\mathbf{q}$ and $\mathbf{k}$ variables needed to describe the route of m_1. Since the problem size and the number of variables are very small, such an SMT instance can be solved in an acceptable time, making it feasible to be integrated into the task scheduler. If such a route does not exist, m_1 has to be shifted to other time slots. In this particular case, the scheduler will subsequently consider t_1 and t_2, since one of the conflicting message finishes at this time instant. If m_1 fails to be scheduled to any time slot due to lack of resources, the task scheduler returns an error and the corresponding candidate solution will be dropped by the MOEA-based optimizer.

6.3 Summary

The code generation and platform configuration back-end is an essential part of our framework. By automatic transformation of an abstract design to a concrete implementation, it significantly increases the practical usability of the tool, overcoming a weak point of most existing work in reliability-aware design. We provide an extensible code generator that combines platform-specific and platform-independent code templates to produce executable application software

directly from the models. To illustrate platform configuration, we consider the TTNoC scheduling problem, which is one of the most important issues in the configuration of the ACROSS platform. It demonstrates how platform-specific configuration tools can be integrated to the framework.

Chapter 7

Case Study

The components of the proposed framework, including the DSE approach and TTNoC configurator, are validated using experiments presented in individual chapters. In this chapter, we proceed with case studies that demonstrate the complete design flow. While the unit tests focus on evaluating the performance of individual components, the case studies aims at demonstrating the practical usability of our framework. We cover all three design phases, namely modeling, DSE and code generation.

7.1 The Adaptive Cruise Control Case Study

For the first case study, we consider an Adaptive Cruise Control (ACC) application in the automotive domain. Its task graph consists of 10 tasks communicating via 16 channels (Figure 7.1). The goal of ACC is to maintain the cruise speed while keeping safe distance from objects ahead. We execute this application using test cases automatically generated from the AutoFocus tool [1], where simulated sensor values are stored as data arrays. The WCET of each task is annotated to the model using the mechanism introduced in 2.1.

The target architecture is the ACROSS MPSoC [30] running in an Altera Stratix IV FPGA [119]. The MPSoC implements in total 8 NoisII soft-cores from Altera, 3 out of which are for general purpose application tasks. The application processors run the PikeOS operating system from Sysgo [120]. Communication between processors is realized using a TTNoC architecture with $2x2$ configuration. A dedicated processor (called the I/O core) is used as the gateway to off-chip resources, including I/O pins, sensors, actuators and network interfaces. The application cores can access these peripherals via pre-defined messages to the I/O core.

[1] http://af3.fortiss.org/

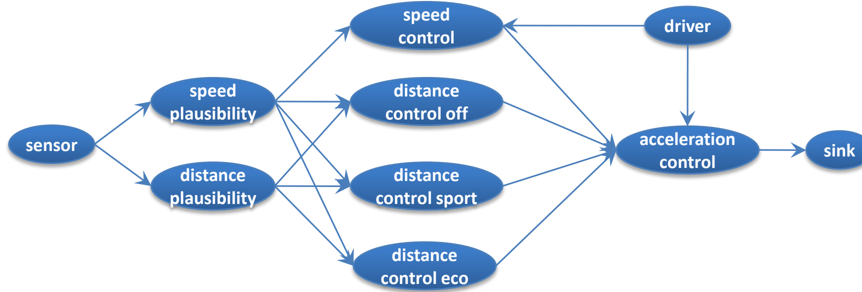


Figure 7.1: Task Graph of ACC

7.1.1 Application and Platform Modeling

The ACC application is modeled using the generic KPN meta-model provided by the framework. The software tasks are modeled as `KPNComponents` and channels as `KPNChannels`. Figure 7.2 is a snapshot of the application model (only part of the channels are shown for clarity). Each of the `KPNComponents` contains a processing request to be mapped to a compatible resource (modeled as a `PNProcessingRequest` object). The mapping of these requests is determined in the DSE phase and it indicates the core where task is executed. The input/output ports of the `KPNComponents` are explicitly modeled. The `PNPort` class superclasses `CapabilityRequest` and `CommunicationEndpoint` to represent the request to platform ports. Similarly, the `KPNChannel` class inherits the `CommunicationTransport` and represents the request to the transport media. Depending on where the source/target of the channel is located, the transport request can be mapped to a local memory channel, an inter-partition channel or an inter-core channel via the TTNoC.

Besides describing the structure of the application, we also annotate extra-functional information in the model. This is done by instantiating according annotation objects and add them as children of certain model elements. For example, the following annotations are added to the sensor task (the first model element in Figure 7.2):

- The WCET of the task is specified using a `TimeSpecificationDelay` object.
- The period of the task is specified using a `TimeSpecificationPeriodicity` object. According to the semantics of KPN, only the periods of the source tasks need to be annotated. The execution sequence of other tasks is determined by the availability of input tokens (recall that the KPN components implement a blocking read on input ports).

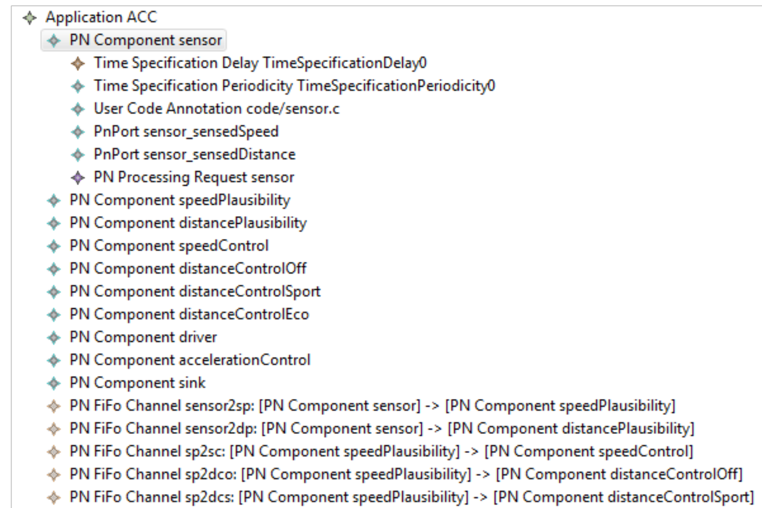


Figure 7.2: The Appliation Model for ACC

- The behavior of the sensor task is described using user-supplied C code. The path to the source file is specified using a `UserCodeAnnotation` object.

In contrast to the application model that can be described using a generic MoC, the execution platform consists of several ACROSS-specific components that are not covered by the generic platform meta-model. Thus, the modeling framework is *instantiated* towards an ACORSS tool-chain. Here, a set of refined meta-models is developed to describe the ACROSS-specific components in more detail. They are stored as a separate Eclipse plugin to maintain the modularity of the tool implementation.

Figure 7.3 depicts the platform model for the ARCOSS architecture. The top-level object is an `MPSoC`, which consists of 8 processors connected via a `TTNoC`. As it can be seen, dedicated objects offered by the ACROSS meta-models are used to distinguish the functionality of processors, e.g., the `ApplicationComponent` and `IOComponent`. The `Link` objects specify the physical connection between platform components, which is used later in the TTNoC scheduling step for routing calculation. Annotations can also be added to the platform model elements. For example, a `ReliabilityAnnotation` object is used to specify the intrinsic failure probability λ of the application processor `CORE1`, which is needed by the Poisson fault model introduced in Section 2.2. In addition, `CORE1` owns a `PikeOS` object that describes the configuration of the operating system. Here, the partitions offered by the OS, inter-partition communication ports and channels and other configuration parameters are covered. This object can be automatically translated into a compatible XML file to configure the PikeOS.

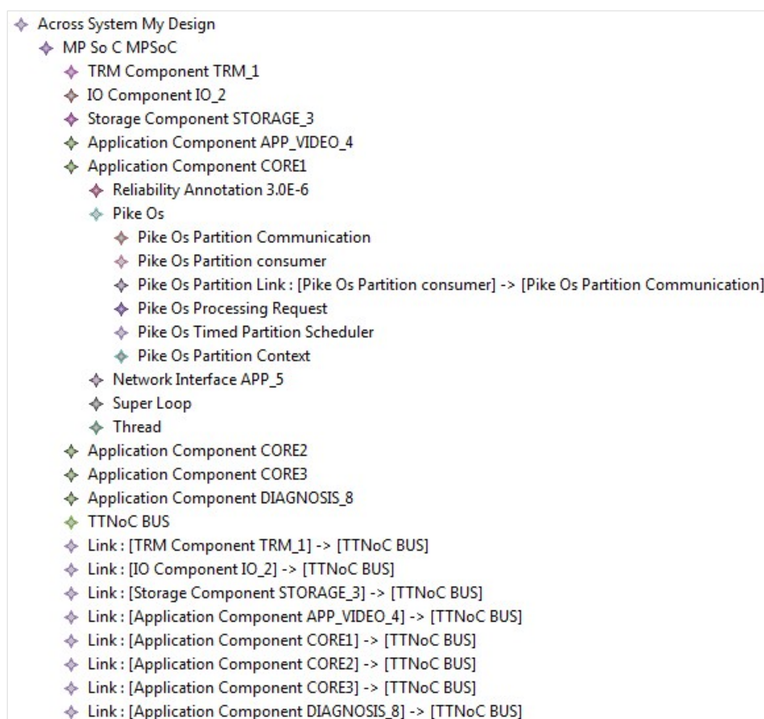


Figure 7.3: The Platform Model for ACROSS Architecture

7.1.2 Design Space Exploration

After construction of the design model, we use design space exploration to determine the mapping of the requests in the application model to the resources in the platform model. The DSE is guided by the design objectives and constraints specified by the user. Here, we consider the following design problem:

- Design objective: the reliability of the application is to be maximized. The application requires fail-operational behavior (in this case, a fault scenario that is only detectable but not correctable is considered as a failure). We use FIT as the metric of reliability, so the fitness value is to be minimized.

- Design objective: the energy consumption of the application is to be minimized. The energy consumption is computed using a simple energy model based on the CPU time occupied by the application.

- End-to-end deadline constraint: the maximum latency between the sensor and sink task must be smaller than the specified value.

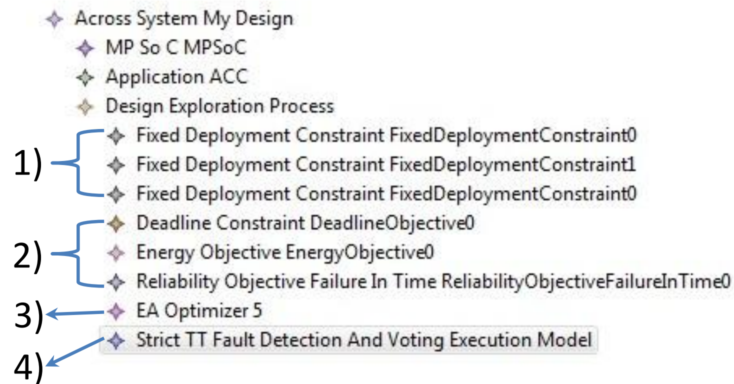


Figure 7.4: The DSE Configuration Model for ACC Case Study

- IO constraints: the sensor, driver and sink tasks must be mapped to the I/O core, which is the gateway to off-chip peripherals. The sensor task gathers information from the speed and distance sensor, the driver task collects the driver input and the sink task delivers the output to the actuator. Due to unique availability of physical devices, these tasks cannot be replicated.

The above DSE problem is specified using a configuration model, which resides in parallel with the application/platform models (see Section 2.1.4). The `FixedDeploymentConstraint` objects marked with 1) are used to force the mapping of sensor/sink tasks to the I/O core. The next three objects in the model describes the design objectives (marked with 2 in the figure). They may contain attributes to customize of a certain design objective. For example, the reliability objective object contains an attribute to specify if the system requires fail-safe or fail-operation behavior. The object marked with 3) is the configuration of the optimization engine. In this example, the MOEA optimizer is used with a population of 100 and a iteration of 500. Finally, the object 4) specifies the scheduling model, which summarizes the scheduling policy and assumptions used in the DSE. The user may select one of the supported scheduling models introduced in Section 2.5. The *TT-FDV* model is used in this particular case study.

The execution time of the DSE process is around 120 seconds on a Windows machine with a 3GHz CPU (single-thread). The results of the DSE are a set of recommended design parameters. Since the two optimization objectives are conflicting with each other (higher reliability requires more redundancy which consumes more energy), the result is not a single solution but a set of design alternatives (Pareto optimal solutions). The DSE tool provides a GUI to visualize the results in the solution space (see Figure 7.5). What is currently shown in the figure is the

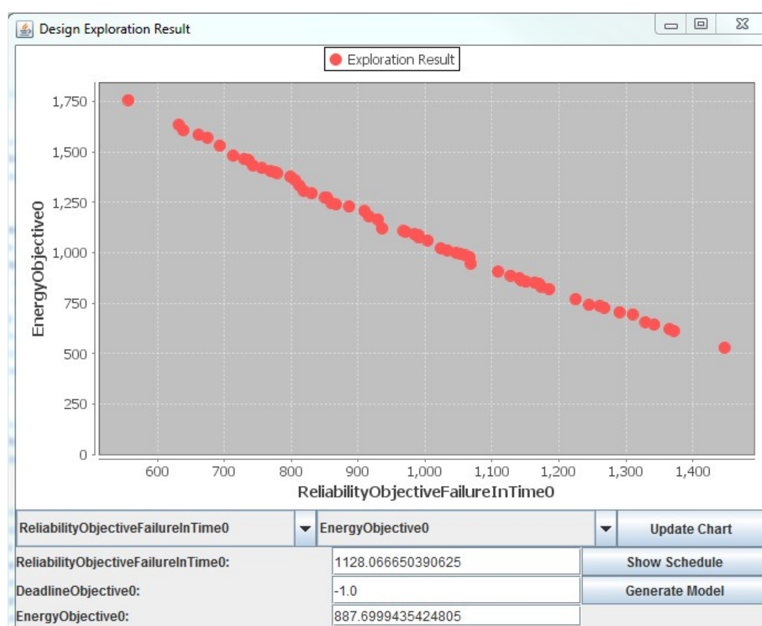


Figure 7.5: DSE Result of the ACC Case Study

tradeoff between the reliability and energy. Here, the solutions found by DSE are projected into a 2D plane, with the x-axis being the fitness value of the reliability objective and y-axis being the energy consumption. The user may select any other two objectives from the list and visualize their tradeoff in the similar way.

The designer can pick a solution in the figure to look into the implementation details. Since we are considering a time-triggered system in the example, the design is represented as a Gantt chart, which depicts the mapping and scheduling for the tasks and messages. A screen shot is shown in 7.6. Here, time slots in dark blue are allocated for application tasks and the time slots in light red are for voting components. As it can be seen, the tasks are selectively replicated and distributed to the three application cores. In particular, a Triple-Modulo-Redundancy (TMR) scheme is implemented for the *DistanceControlEco* task using temporal replication, whereas a TMR is implemented for the *AccelerationControl* task using spatial replication. Since *DistanceControlEco* provides input to the *AccelerationControl*, a voter is inserted at each replica of *AccelerationControl*. Also, the results of replicated *AccelerationControl* tasks are voted at the sink task.

Based on the tradeoff analysis of the DSE results, the designer selects the final implementation. An updated model can be automatically generated using the *Generate Model* but-

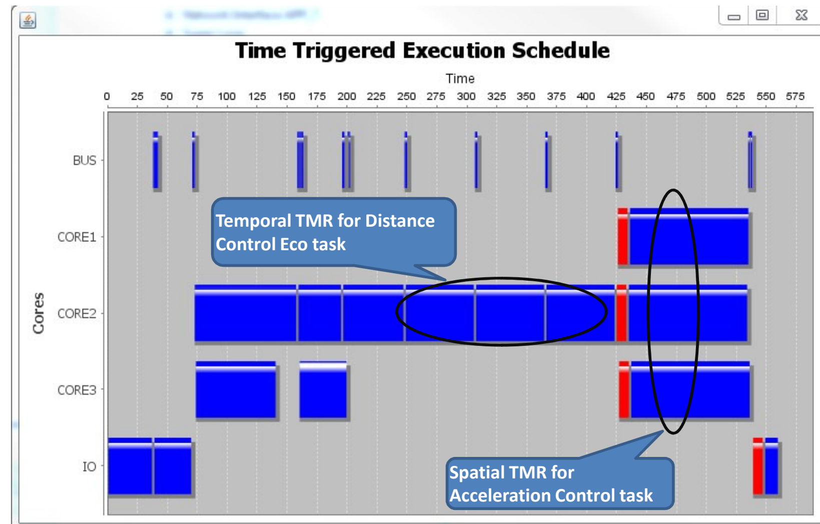


Figure 7.6: DSE Result of the ACC Case Study

ton from the GUI. The design parameters are automatically applied in the updated model. The `Processing` requests from software tasks, `CommunicationEndpoint`requests from the KPN ports and `CommunicationTransport` requests from the KPN channels are mapped to according resources in the platform model. In addition, the FTMs are explicitly instantiated, including the replicated tasks, voting components as well as the additional ports/channels required to send the results to the voter. The updated model is ready for the code generation and platform configuration phase.

7.1.3 Code Generation and Execution

We use the design alternative shown in Figure 7.6 as an example to demonstrate the code generation process. Table 7.1 summarizes the mapping of tasks. In our modeling approach, the task mapping is specified by allocating the `ProcessingRequests` owned by the tasks to the `ProcessingResources` in the platform model. In this particular case, the according resource is a `PikeOSPartition` and a fraction of the resource is allocated as a `PikeOSThread` to serve the request. In addition, the communication requests (both endpoint and transport) are allocated based on the mapping of source/target tasks. If the two communicating entities are mapped to the same `PikeOSPartition`, communication is implemented via local memory. Similarly, inter-partition communication is realized using PikeOS channels and inter-core communication is realized using TTNoC messages (see Table 7.2).

core	tasks
CORE1	accelerationControl (replica1)
CORE2	speedPlausibility, distancecontrolEco (replica1) distancecontrolEco (replica2), distancecontrolEco (replica3), distancecontrolSport, speedControl, accelerationControl (replica2)
CORE3	distancePlausibility, distanceControlOff, accelerationControl (replica3)
IO	sensor, driver, sink

Table 7.1: Task Mapping in the Example Design

capability request in application model	capability resource in platform model
processing request	PikeOs thread
communication endpoint (intra-partition)	local memory port
communication endpoint (inter-partition, inter-core)	PikeOS queuing/sampling port
communication transport (intra-partition)	memory block
communication transport (inter-partition)	PikeOS queuing/sampling channel
communication transport (intra-core)	TTNoC message

Table 7.2: Mapping of Capability Requests to Resources

Using techniques presented in Section 6, C source code can be directly generated from the design model. Here, an image is created for each application core [1]. The operating system abstracts away the underlying communication mechanism (here the TTNoC communication) and provides a user-space communication API. In this example, we select the POSIX personality of PikeOS, which provides the standard file access API (open, read, write, etc).

Besides the application source code, the code generator also produces the following platform configuration files:

- PikeOS Configuration. PikeOS is a predictable OS that requires design time static configuration. In particular, PikeOS ports and channels needed for communication across the partition boundary (inter-partition and inter-core) must be statically allocated. For each application core, the PikeOS configuration is stored in a custom XML file called Virtual Machine Initialization Table (VMIT). In our implementation, we transform the PikeOS XML schema into an ECore meta-model using utilities from EMF and implement the PikeOS configuration as a transformation from our design model to the target configuration model.

[1]In this example, all tasks mapped to an application core are implemented in a single PikeOS partition. Hence, only one image needs to be compiled. If a processor contains multiple PikeOS partitions, a separate image is created for each partition.

- TTNoC Configuration. Based on the task mapping, we gather all inter-core communication requests and submit them to the TTNoC scheduler described in Section 6.2. The scheduler computes the route and phase of messages. Afterwards, a low-level utility provided by the TTNoC vendor is used to synthesize binary data that can be loaded to the on-chip TTNoC configuration memory.

With the source code and the configuration files described above, the application is successfully executed on the target FPGA platform. As mentioned in Section 6.1, one of the major objectives of our code generator is retargetability. To retarget the ACC application to a different platform, the following steps need to be carried out: 1) replace the platform model accordingly; 2) map the application model to the new platform model, either using DSE or manually; 3) execute code generation. As the platform-specific code templates are directly associated with the platform model, the generated code is automatically adapted. For example, the ACC application has also been successfully deployed to a single-core NiosII platform running PikeOS as well as the local Windows host.

7.2 The Industrial Control Demonstrator

The proposed framework has also been used in the design of an industrial control demonstrator, which is built based on the Modular Production System (MPS) from Festo Didactic [121]. As an example of safety-critical automation applications, we select an MPS station that performs a sorting task. Figure 7.7 shows the mechanical setup of the sorting station. It receives work pieces (WP) from the previous station and sorts them according to the color and material. We use three kinds of work pieces: 1) non-black, metallic WPs to be sorted to the first slide; 2) black, non-metallic WPs to be sorted to the second slide; 3)non-black, non-metallic WPs to be sorted to the last slide. The MPS stations are originally controlled by standard Programmable Logic Controllers (PLCs). We replace the PLCs by an ACROSS MPSoC. To realize the desired functionality, the station is equipped with the following sensors and actuators:

- Sensor1: detects the availability of a new WP;
- Sensor2: detects if the work piece is black or not;
- Sensor3: detects if the work piece is metallic or not;
- Sensor4: a light barrier that detects if the storage slide is already full;

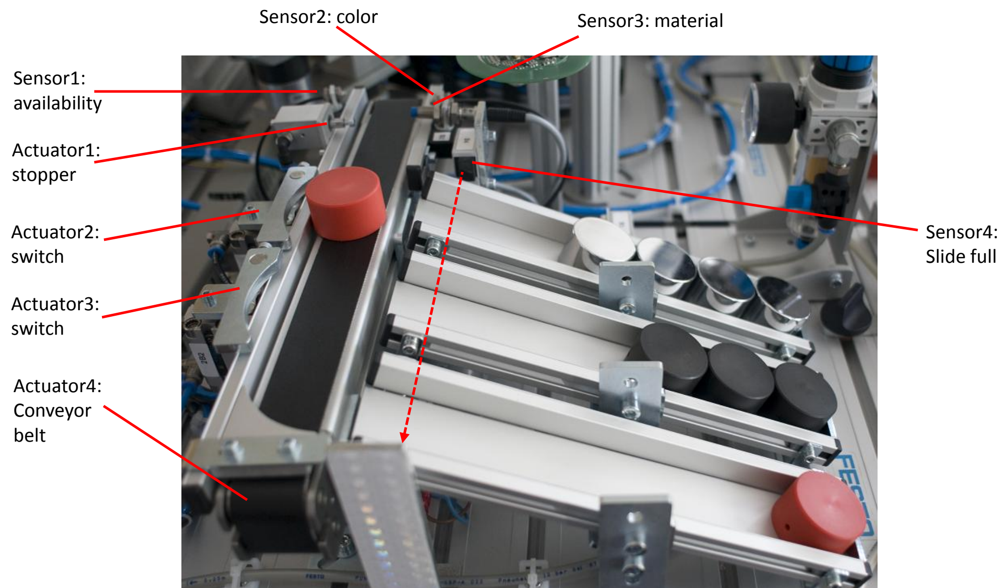


Figure 7.7: Mechanical Setup of Industrial Control Demonstrator

- Actuator1: a stopper that blocks the work piece. It is used to hold the WP until the station is ready for sorting, e.g., the color/material is already determined and the station is not occupied by the previous WP;
- Actuator2: a mechanical switch. When it is extended, the work piece will be sorted into the first storage slide;
- Actuator3: another switch to sort the work piece the second storage slide. If both switches are not extended, the work piece is sorted to the last slide by default;
- Actuator4: motor that drives the conveyor belt.

We again use KPN as a coarse-grained model to describe the application structure. The sorting application is modeled using 4 tasks as shown in Figure 7.8. The sensor task gathers the current value of all sensors and packs the data into a package with pre-defined format. The plausibility check task, as its name suggests, checks the validity of the input data. If an invalid state is detected, the system executes a safe-shutdown. The list of invalid vectors is specified by the user. For example, an invalid state could be that the sensor detects a black-metallic work piece, since no such WP is actually used. The controller implements the core application logic and computes the output value to the actuators. Finally, the sink task apply the control output to the actuators.

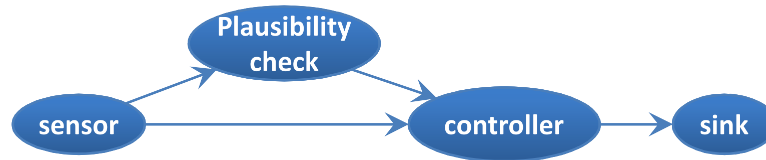


Figure 7.8: Application Model of Sorting Application

While the behavior of sensor, actuator and plausibility check tasks are specified using user-supplied C code, the controller logic is modeled using the industrial control Domain-Specific Languages (DSLs). This includes the State Flow Chart (SFC) and Function Block Diagram (FBD) languages specified in IEC61131 standard [105]. EasyLab [122] is adopted as a domain specific modeling tool to provide a graphical interface to create SFC and FBD models. The Easylab models can be directly imported as a fine-grained model to describe the behavior of `KPNComponents` (here the controller task). As mentioned in Section 6.1.2.1, our modeling framework supports functional code generation from the fine-grained models.

Figure 7.9 shows the SFC model of the sorting station. An SFC is similar to a state machine that captures the control flow of the application. It can be seen that the sorting task is performed in the following steps:

- State S1: initialization.
- State S2: start the conveyor belt.
- State S3: examine the color and material of the work piece
- State S4-S6: based on the type of WP, open switch1, switch2 or neither.
- State S7: open the blocker and let the conveyor belt run until the WP is sorted.
- State S8: reset to prepare for the next WP.

The deployment of the application is under a set of constraints. First of all, the system requires fail-safe behavior and the reliability is to be maximized. Second, the sensor and sink tasks must be executed on the I/O processor due to constraints of the ACROSS platform. To increase the system reliability, we implement a temporal TMR scheme for the sensor task, i.e., we read the sensor three times and vote the results. In contrast, the sink task cannot be replicated. Third, the sensor to actuator delay must be less than 10ms (end to end deadline constraint). Finally, the resource utilization (only the processor time is considered) is to be

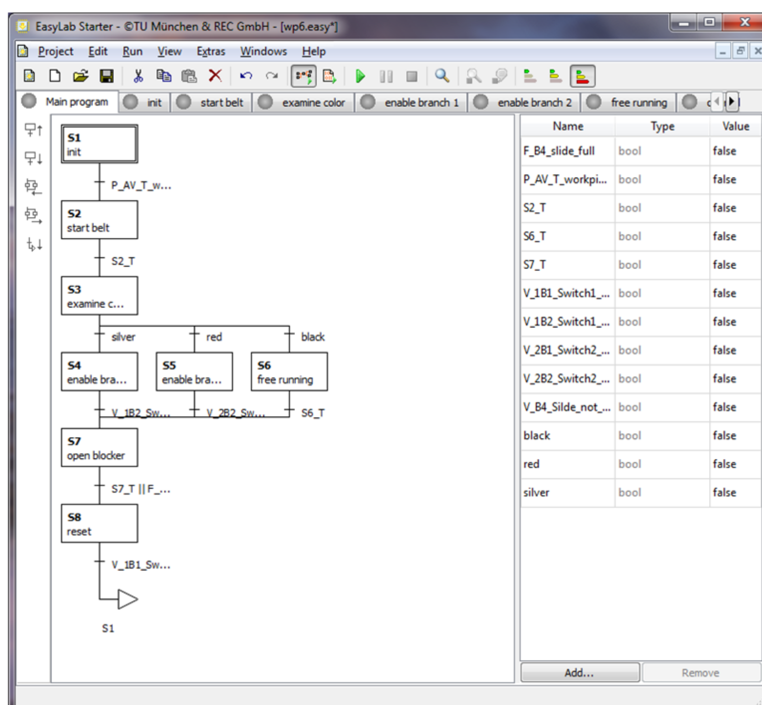


Figure 7.9: Application Logic of Sorting Task

minimized. The configuration and execution of the DSE process is similar as the ACC example. For this simple application, the DSE engine suggests only two design alternatives. The first one is a regular implementation without FTMs (the resource utilization is minimized in this case.). The second implementation is a DMR scheme in which the plausibility check and controller tasks are duplicated and distributed to two application processors (Figure 7.10) [1]. We select the DMR implementation and generate the application software. The platform configuration is also similar as in the ACC case study. The major challenge is the configuration of on-chip communication schedule and PikeOS.

Figure 7.11 illustrates the final setup of the demonstrator. The ACROSS MPSoC is again implemented in an FPGA broad from Altera. A HSMC extension board provides the physical interface to the FPGA board hosting the MPSoC. In this particular demonstrator, the MPS station runs the standard Profibus [123] protocol in the industrial automation domain to communicate sensor/actuator values. We use an Anybus [124] card to implement the Profibus interface for the control system.

[1] Since the system requires fail-safe, the DMR scheme has the same reliability as TMR but consumes less resource. Hence DMR dominates TMR in this particular case.

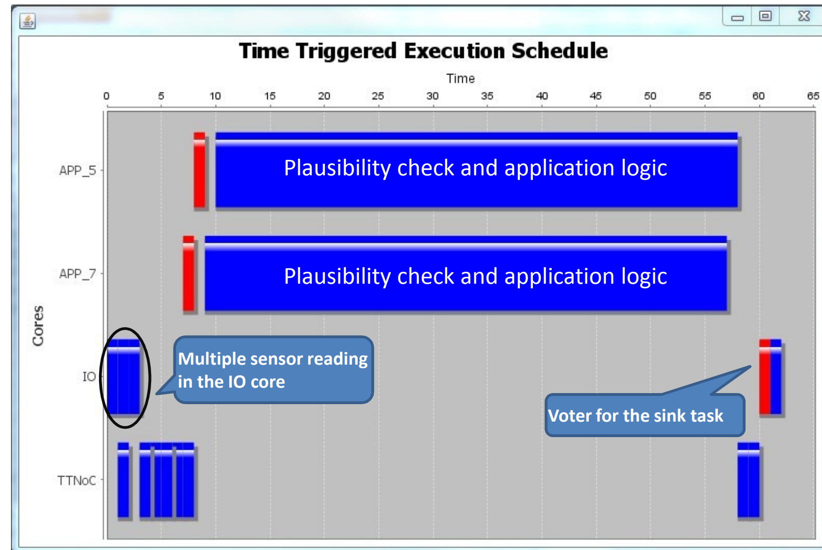


Figure 7.10: DMR implementation for the Industrial Control Demonstrator

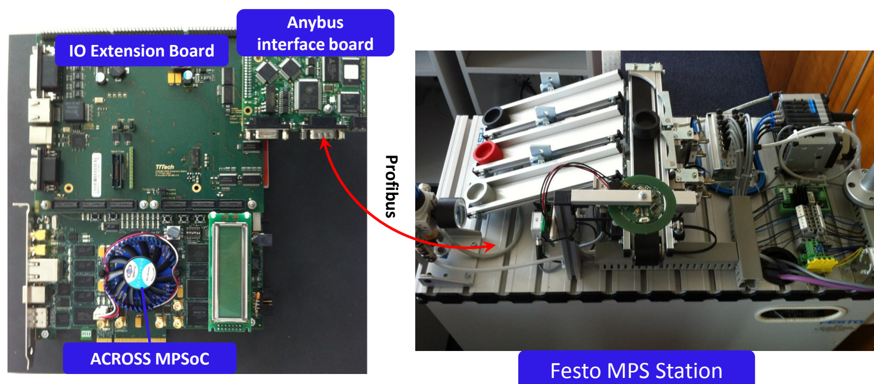


Figure 7.11: Deployment of Industrial Control Demonstrator

7.3 Discussion of Usability

Due to the high problem complexity, reliability-aware system design is a problem that is very difficult to solve, even if possible, without sufficient tool support. Hence, as one of the uniqueness of our approach, we provide not only the key algorithms but also an implementation as a tool-chain. We use the above case studies to evaluate the usability of approach. This section summarizes the aspects that is currently supported and also outlines the next steps for the tool implementation. The three-step development processing presented in Section 1.4 is reused.

Model the system. In the modeling phase, the Eclipse-based MDD environment provides the designer a central IDE (Integrated Development Environment). In principle, the generic meta-models can be used to model any application or platform as components, ports and channels. Nevertheless, to make the models more usable, the generic meta-models have to be refined to contain more details. For example, in the case studied presented above, we developed both refined application and platform meta-models, in order to describe the sorting station controller using industrial control DSLs and the ACROSS platform.

Currently, industrial control is the only domain where DSLs are supported. The applications from other domains, like the ACC from automotive domain, have to be described with the generic KPN meta-model. I will be a great advantage to add the support for other DSLs or standard models, e.g., Matlab Simulink [125] models. Another highly desired feature of the MDD tool-chain is a graphical modeling interface in addition to the tree-based editor from EMF.

DSE. In the DSE phase, we provide a generic multi-objective optimization framework. The following optimization goals are supported and can be freely combined by the user: 1) (possible multiple) end-to-end deadline; 2) reliability, considering DUF, SDC or combined; 3) resource consumption; 4) application specific constraints, e.g., a task must or must not be mapped to a certain core. For each optimization goal, the according analysis algorithm is equipped, which takes the design model as input and evaluates the fitness value. We are interested to extend the DSE framework to support other Non-Functional Properties (NFPs), such as energy consumption.

Another aspect to improve the usability of the tool-chain is to integrate third-party tools. Currently, certain analysis data still has to be obtained manually using third-party tools and annotated into the models, for example the WCET of tasks (using [54]) and the fault-detector

protected version of the components (using [126]). Integrating these tools will make the development process smoother.

Realize the system. To bring the virtual system (models) to a real-world implementation, we support C source code generation, build system generation and automatic platform configuration (e.g., PikeOS operating system and ACROSS architecture). What is currently not covered is the aspect of testing. After obtaining the source code, the user has to bring up the target system and manually develop and execute test cases. Automatic generation of test cases and regression test would be a very useful feature to implement in the next step.

With the above discussion, we believe that our approach is able to cover the key challenges of the reliability-aware design scenario. Nevertheless, many additional features would be desired to make it professional.

Chapter 8

Conclusion and Outlook

8.1 Main Results

This thesis tackles the problem of reliability-aware real-time embedded system design. We provide both algorithms and a tool implementation to address the major challenges in the design flow. To build systems with high reliability, we actively embed fault-tolerant mechanisms into the system while considering timing, resource and other non-functional constraints. In the development of our approach, we identify two major limitations in the current work, which constitute a gap between theory and practice in the area of fault-tolerant systems. On the one hand, the current approaches involve some unrealistic assumptions in the analysis. On the other hand, they lack sufficient tool support for system modeling and implementation. We present novel techniques to bridge this gap and thereby bring the research results further into practice. The framework proposed in this thesis is among the first to provide a complete model-driven reliability-aware design flow, covering design challenges from high-level system modeling/analysis/exploration down to low-level code implementation. The main results of this work are summarized as follows:

- We present a generic reliability-aware DSE approach supporting realistic fault models and a large set of FTMs. The tree-based probabilistic analysis computes system-level reliability in the present of active redundancy, voting, fault detection and other advanced FTMs. We use a hierarchical encoding scheme to transform the DSE problem into an instance of MOEA. The optimization process generates time-triggered schedules under reliability, timing, resource and other user-specified constraints. We especially focus on

removing the inappropriate assumption on perfect fault detection, which is a major issue that limits the practical usability of current approaches.

- We present a reliability-aware tool framework using a model-driven approach. At the front-end, the modeling tool provides a user-friendly interface to specify the design problems. At the back-end, automatic generation of implementation artifacts accelerates the design process. Moreover, the formally defined models serve as a central consistent representation of the system for easy tool integration. In this context, we combine our DSE, code generation and platform configuration techniques to tackle the challenges in fault tolerant system design. The implementation of the proposed approach in terms of a tool-supported design flow significantly improves the practical value, which is demonstrated using a real-world case study as well as a demonstrator. From another angle, our work can also be used as a backbone to integrate other existing approaches in the field. This aligns with the general objective of the underlying capability modeling framework, which aims at providing a tool framework that can be instantiated for different design scenarios.

8.2 Outlook

During the implementation of the approach, we keep extensibility as a first-order design criterion. On the one hand, we develop generic analysis and optimization algorithms that can be adapted to different fault and system models. On the other hand, components of the framework, such as meta-model packages, analysis/optimization tools and platform-specific tools are maintained in separate Eclipse plugins to keep the modularity. This allows us the use the MDD framework as the foundation for future research. Our work could be extended in several aspects.

- **Further fault tolerant mechanisms**. Although we aims at supporting a configurable set of FTMs, several techniques are currently not covered. One important FTM we plan to support is Check-Point and Roll-Back (CP-RB). So far, we consider fault detection and recovery only at task-level, i.e., the entire task is re-executed if it is faulty. Using CP-RB, we could deal with tasks in a more fine-grained manner. Check points can be embedded in the middle of a task and the recovery process can start from the most recent check point instead of the beginning of the task. The overall efficiency of the system could be improved in this case. As fault-tolerant system design is a very active research area, we would also be interested to consider new FTMs proposed by researchers.

- **Mix-criticality**. Mixed criticality systems are becoming increasingly important in many safety-related domains [96]. There, multiple functions with different importance and certification assurance levels are integrated using a shared computing platform. The mixed-criticality integration imposes new challenges in reliability-aware design, e.g., how to build cost-efficient system to meet the distinct reliability requirements of each application and how to guarantee sufficient separation between critical and non-critical applications to enable low-overhead certification [127]. We consider these new problems as an important research direction.

- **Fault-tolerance at other layers**. So far, our approach focuses on fault-tolerance at task-level. In some scenarios, it is also beneficial to implement FTMs at other layers of the design. For example, we could implement spatial redundancy at cluster-level by replicating the entire MPSoC platform and voting the results of the entire application. From the system point of view, cross-layer analysis and optimization are needed to find the optimal design alternative.

- **Other Non-Functional Properties**. Besides reliability and timing, safety-related applications may also involve other NFPs. One important one we see is energy consumption. Here, the major challenge is to combine energy management techniques such as Dynamic Voltage and Frequency Scaling (DVFS) with fault-tolerant techniques and evaluate the tradeoff between multiple NFPs. To support a new NFP, according analysis tools must be integrated and the optimization approach must be extended to take the new design freedom into account. For example, to support energy optimization, energy model and estimation techniques are needed and the DVFS-related configuration parameters must be covered. The joint energy-reliability optimization problem is considered as an important contribution.

References

[1] B. W. Johnson. **Fault-Tolerant Microprocessor-Based Systems**. *IEEE Micro*, **4**:6–21, 1984. 1

[2] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. **The impact of technology scaling on lifetime reliability**. In *International Conference on Dependable Systems and Networks*, pages 177 – 186, 2004. 1

[3] S. Borkar. **Designing reliable systems from unreliable components: the challenges of transistor variability and degradation**. *Micro, IEEE*, **25**, 2005. 1, 58

[4] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K.S. Kim. **Robust system design with built-in soft-error resilience**. *Computer*, **38**(2):43 – 52, 2005. 1

[5] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. **Modeling the effect of technology trends on the soft error rate of combinational logic**. In *International Conference on Dependable Systems and Networks (DSN)*, pages 389 – 398, 2002. 1

[6] W. Wolf. **The future of multiprocessor systems-on-chips.** In *Design Automation Conference (DAC)*, 2004. 2, 4

[7] Jianjiang Ceng, Jerónimo Castrillón, Weihua Sheng, Hanno Scharwächter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Tsuyoshi Isshiki, and Hiroaki Kunieda. **MAPS: an integrated framework for MPSoC application parallelization**. In *Design Automation Conference (DAC)*, 2008. 2, 4, 5

[8] N. Storey. **Safety-Critical Computer Systems.** In *Addison Wesley Longman Ltd*, 1996. 2

[9] Chanhee Lee, Hokeun Kim, Hae-woo Park, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. **A task remapping technique for reliable multi-core embedded systems**. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 307–316, 2010. 3, 18, 29, 35, 55

[10] C. Yang and A. Orailoglu. **Predictable execution adaptivity through embedding dynamic reconfigurability into static MPSoC schedules**. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 15–20, 2007. 3, 30

[11] Michael Glass, Martin Lukasiewycz, Christian Haubelt, and Jürgen Teich. **Incorporating graceful degradation into embedded system design**. In *Design, Automation and Test in Europe (DATE)*, pages 320–323, 2009. 3

[12] Viacheslav Izosimov, Paul Pop, Petru Eles, and Zebo Peng. **Design Optimization of Time-and Cost-Constrained Fault-Tolerant Distributed Embedded Systems**. In *Design, Automation and Test in Europe (DATE)*, pages 864–869, Washington, DC, USA, 2005. IEEE Computer Society. 3, 6, 21, 22, 23, 31, 32, 33, 34, 35, 70

[13] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M.J. Irwin. **Reliability-aware co-synthesis for embedded systems**. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 41 – 50, sep 2004. 3, 31, 33, 35

[14] Paul Pop, Viacheslav Izosimov, Petru Eles, and Zebo Peng. **Design Optimization of Time- and Cost-Constrained Fault-Tolerant Embedded Systems with Checkpointing and Replication**. *IEEE Transactions on VLSI*, pages 389 – 402, 2009. 3, 31, 33, 35, 70

[15] Ying Zhang and K. Chakrabarty. **A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems**. *IEEE Trans. CAD of Integrated Circuits and Systems*, pages 111 – 125, 2006. 3

[16] F. Liberato, R. Melhem, and D. Mosse. **Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems**. *IEEE Trans. on Computers*, pages 906 – 914, 2000. 3

[17] Ching-Chih Han, K.G. Shin, and Jian Wu. **A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults**. *IEEE Trans. on Computers*, pages 362 – 372, 2003. 3

[18] Claudio Pinello, Luca P. Carloni, and Alberto L. Sangiovanni-Vincentelli. **Fault-Tolerant Deployment of Embedded Software for Cost-Sensitive Real-Time Feedback-Control Applications**. In *Design, automation and test in Europe (DATE)*, pages 1164 – 1169, 2004. 3, 32

[19] Architectural Requirments. **IEC61508-2, chapter 7.4.3.1.1, Tab.2 and 3**. 3

[20] Int. Electrotechnical Commission. **IEC 61508: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems**. 3

[21] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition*. 2009. 4

[22] R. Ernst. **Certification of trusted MPSoC platforms**. In *MPSoC*, 2010. 4

[23] C. Buckl, A. Camek, G. Kainz, C. Simon, L. Mercep, H. Stahle, , and A.Knoll. **The software car: Building ICT architectures for future electric vehicles**. In *In Electric Vehicle Conference (IEVC)*, 2012. 4

[24] A. Girault and H. Kalla. **A Novel Bicriteria Scheduling Heuristics Providing a Guaranteed Global System Failure Rate**. *IEEE Transactions on Dependable and Secure Computing*, pages 1–13, 2009. 6, 17, 28, 30, 31, 34, 35, 40, 61, 62, 70, 76, 81

[25] Michael Glass, Martin Lukasiewycz, Thilo Streichert, Christian Haubelt, and Jürgen Teich. **Reliability-Aware System Synthesis**. In *Design, Automation and Test in Europe (DATE)*, pages 409–414, Nice, France, April 2007. IEEE Computer Society. 6, 18, 19, 29, 35

[26] Dakai Zhu and Hakan Aydin. **Reliability-Aware Energy Management for Periodic Real-Time Tasks**. *IEEE Transactions on Computers*, **99**:1382–1397, 2009. 6, 17, 21, 31, 32, 33, 35, 70

[27] Philip Axer, Maurice Sebastian, and Rolf Ernst. **Reliability Analysis for MPSoCs with Mixed-Critical, Hard Real-Time Constraints**. In *International*

conference on Hardware/software codesign and system synthesis (CODES+ISSS), pages 149–158, 2011. 6, 17

[28] Jia Huang, Kai Huang, Andreas Raabe, Christian Buckl, and Alois Knoll. **Towards Fault-Tolerant Embedded Systems with Imperfect Fault Detection**. In *49th Design Automation Conference (DAC)*, pages 188–196, San Francisco, CA, June 2012. 6, 25, 33

[29] Bart Kienhuis, Ed Deprettere, Kees Vissers, and Pieter van der Wolf. **An Approach for Quantitative Analysis of Application Specific Dataflow Architectures**. In *In Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 338–349, 1997. 7

[30] ACROSS Project. **http://www.across-project.eu/**. 8, 14, 18, 22, 23, 119

[31] Simon Barner, Andreas Raabe, Christian Buckl, and Alois Knoll. **Beschreibung der Plattformabhängigkeit eingebetteter Applikationen mit Dienstmodellen**. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme*, 2011. 12, 14, 36

[32] Gilles Kahn. **The Semantics of a Simple Language for Parallel Programming**. In *Proceedings of IFIP Congress*, 1974. 12

[33] Eclipse Modeling Framework. **http://www.eclipse.org/modeling/emf/**. 16

[34] D. K. Pradhan. *Fault-Tolerant Computer System Design*. 1996. 17

[35] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. **A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor**. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003. 17

[36] Yun Xiang, Thidapat Chantem, Robert P. Dick, X. Sharon Hu, and Li Shang. **System-level reliability modeling for MPSoCs**. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 297–306, 2010. 17, 28

[37] S. M. Shatz and J.-P. Wang. **Models and algorithms for reliability-oriented task-allocation in redundant distributed-computer systems**. *IEEE Transactions on Reliability*, **38**:16–27, 1989. 17, 28

[38] J.H. Lala and R.E. Harper. **Architectural principles for safety-critical real-time applications**. *Proceedings of the IEEE*, **82**(1):25 –40, jan 1994. 18

[39] S. Mitra, N.R. Saxena, and E.J. McCluskey. **Common-mode failures in redundant VLSI systems: a survey**. *IEEE Transactions on Reliability*, **49**(3):285 –295, sep 2000. 18

[40] S. Mitra, N.R. Saxena, and E.J. McCluskey. **A design diversity metric and analysis of redundant systems**. *IEEE Transactions on Computers*, **51**(5):498 –510, may 2002. 18

[41] R. Obermaisser and O. Hoftberger. **Fault containment in a reconfigurable Multi-Processor System-on-a-Chip**. In *International Symposium on Industrial Electronics (ISIE)*, 2011. 18

[42] Adam S. Hartman, Donald E. Thomas, and Brett H. Meyer. **A case for lifetime-aware task mapping in embedded chip multiprocessors**. In *International conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, pages 145 – 154, 2010. 18, 29

[43] Lin Huang, Feng Yuan, and Qiang Xu. **Lifetime reliability-aware task allocation and scheduling for MPSoC platforms**. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 51 – 56, 2009. 18, 29

[44] Christian El Salloum, Martin Elshuber, Oliver Hoftberger, Haris Isakovic, and Armin Wasicek. **The ACROSS MPSoC – A New Generation of Multi-core Processors Designed for Safety-Critical Embedded Systems**. In *Euromicro Conference on Digital System Design (DSD)*, pages 105–113, 2012. 18

[45] Jia Huang, Jan Olaf Blech, Andreas Raabe, Christian Buckl, and Alois Knoll. **Analysis and Optimization of Fault-Tolerant Task Scheduling on Multiprocessor Embedded Systems**. In *International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS)*, pages 247–256, Taipei, Taiwan, Oct 2011. 19, 23, 70

[46] Adrian Lifa, Petru Eles, Zebo Peng, and Viacheslav Izosimov. **Hardware/Software Optimization of Error Detection Implementation for Real-Time Embedded Systems**. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010. 20, 33, 35, 70

[47] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. **Software-Implemented Hardware Error Detection: Costs and Gains**. In *Third International Conference on Dependability*, 2010. 20, 33, 70, 74, 82

[48] K. Pattabiraman, Z.T. Kalbarczyk, and R.K. Iyer. **Automated Derivation of Application-Aware Error Detectors Using Static Analysis: The Trusted Illiac Approach**. *IEEE Transactions on Dependable and Secure Computing*, **8**(1):44 –57, 2011. 20

[49] C. LaFrieda, E. Ipek, J.F. Martinez, and R. Manohar. **Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor**. In *International Conference on Dependable Systems and Networks (DSN)*, pages 317 – 326, 2007. 20

[50] Viacheslav Izosimov, Ilia Polian, Paul Pop, Petru Eles, and Zebo Peng. **Analysis and optimization of fault-tolerant embedded systems with hardened processors**. In *Design, Automation and Test in Europe (DATE)*, pages 682–687, 2009. 21, 28, 31, 34, 35

[51] Baoxian Zhao, Hakan Aydin, and Dakai Zhu. **Enhanced reliability-aware power management through shared recovery technique**. In *International Conference on Computer-Aided Design(ICCAD)*, pages 63–70, 2009. 21, 28, 33, 70

[52] Prabhat Kumar Saraswat, Paul Pop, and Jan Madsen. **Task Mapping and Bandwidth Reservation for Mixed Hard/Soft Fault-Tolerant Embedded Systems**. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 89–98, 2010. 21, 31, 33, 70

[53] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. **Reliable and Precise WCET Determination for a Real-Life Processor**. In *International Conference on Embedded Software (EMSOFT)*, pages 469–485, 2001. 22

[54] AbsInt. **http://www.absint.com/timingexplorer/index.htm**. 22, 132

[55] Viacheslav Izosimov, Paul Pop, Petru Eles, and Zebo Peng. **Synthesis of Fault-Tolerant Schedules with Transparency/Performance Trade-offs for Distributed Embedded Systems**. In *Design, Automation and Test in Europe (DATE)*, 2006. 25, 32

[56] Jia Huang, Andreas Raabe, Kai Huang, Christian Buckl, and Alois Knoll. **A framework for reliability-aware design exploration for mpsoc based systems**. *Design Automation for Embedded Systems, Springer*, 2013. 25

[57] A. Birolini. **Reliability Engineering - Theory and Practice**. *Springer, Berlin, Heidelberg*, 2004. 27

[58] I. Koren and C. M. Krishna. *Fault-tolerant systems*. 2007. 27, 29

[59] R. Degraeve, G. Groeseneken, R. Bellens, M. Depas, and H.E. Maes. **A consistent model for the thickness dependence of intrinsic breakdown in ultra-thin oxides**. In *Electron Devices Meeting, 1995., International*, dec 1995. 27

[60] M. Gall, C. Capasso, D. Jawarani, R. Hernandez, and H. Kawasaki andP. S. Ho. **Statistical analysis of early failures in electromigration**. *Journal of Applied Physics*, **8**(2):732 – 740, 2001. 27

[61] Ayse K. Coskun, Tajana Simunic Rosing, Kresimir Mihic, Yusuf Leblebici, and Giovanni De Micheli. **Analysis and Optimization of MPSoC Reliability**. *Journal of Low Power Electronics (JOLPE)*, pages 56–69, 2006. 27

[62] T.S. Rosing, K. Mihic, and G. De Micheli. **Power and Reliability Management of SoCs**. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **15**:391 –403, 2007. 27

[63] A. Maheshwari, I. Koren, and N. Burleson. **Techniques for transient fault sensitivity analysis and reduction in VLSI circuits**. In *International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 597 – 604, 2003. 28

[64] Dakai Zhu and H. Aydin. **Energy Management for Real-Time Embedded Systems with Reliability Requirements**. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 528 – 534, 2006. 28, 32, 33

[65] D. Lloyd and M. Lipow. *Reliability: Management, Methods, and Mathsmatics*. 1962. 28

[66] P.A. Jensen and M. Bellmore. **An Algorithm to Determine the reliability of a complex system**. *IEEE Trans. Reliability*, **18**:169 –174, 1969. 28

[67] Int. Electrotechnical Commission. **Fault Tree Analysis. Edition 2.0. IEC 61025**, 2006. 28

[68] Marvin Rausand and Arnljot Hoylan. *System Reliability Theorie, Models, Statistical Methods, and Applications*. 2004. 28

[69] Michael Glass, Martin Lukasiewycz, Felix Reimann, Christian Haubelt, and Jürgen Teich. **Symbolic Reliability Analysis and Optimization of ECU Networks**. In *Design, Automation and Test in Europe (DATE)*, pages 158–163, Munich, Germany, 2008. 28, 29

[70] Rainer Feldmann, Christian Haubelt, Burkhard Monien, and Jürgen Teich. **Fault Tolerance Analysis of Distributed Reconfigurable Systems Using Sat-Based Techniques**. In *International Conference on Field Programmable Logic and Applications*, 2003. 29

[71] Felix Reimann, Michael Glass, Martin Lukasiewycz, Christian Haubelt, Joachim Keinert, and Jürgen Teich. **Symbolic Voter Placement for Dependability-Aware System Synthesis**. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 237–242, Atlanta GA, USA, October 2008. 29, 35

[72] C. Pinello, L.P. Carloni, and A.L. Sangiovanni-Vincentelli. **Fault-Tolerant Distributed Deployment of Embedded Control Software**. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **27**(5):906 –919, 2008. 29, 35

[73] Changyun Zhu, Zhenyu (Peter) Gu, Robert P. Dick, and Li Shang. **Reliable multiprocessor system-on-chip synthesis**. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 239–244, 2007. 29

[74] G. Manimaran and C. S. R. Murthy. **A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis**. In *IEEE Trans. Parallel and Distributed Systems*, pages 1137–1152, 1998. 29

[75] Brett H. Meyer, Adam S. Hartman, and Donald E. Thomas. **Cost-effective slack allocation for lifetime improvement in NoC-based MPSoCs**. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1596–1601, 2010. 29

[76] Chengmo Yang and Alex Orailoglu. **Towards no-cost adaptive MPSoC static schedules through exploitation of logical-to-physical core mapping latitude**. In *Design, Automation and Test in Europe (DATE)*, pages 63 – 68, 2009. 30

[77] Anne Benoit, Louis-Claude Canon, Emmanuel Jeannot, and Yves Robert. **Reliability of task graph schedules with transient and fail-stop failures: complexity and algorithms**. *Journal of Scheduling*, pages 1–13, 2011. 30, 39, 40, 76, 81

[78] Xiao Qin and Hong Jiang. **A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems**. *Parallel Comput.*, **32**, June 2006. 31

[79] F. Glover and C. McMillan. **The general employee scheduling problem: an integration of MS and AI**. *Computers and Operations Research*, 1986. 31

[80] B.P. Dave and N.K. Jha. **COFTA: hardware-software co-synthesis of heterogeneous distributed embedded systems for low overhead fault tolerance**. *IEEE Transactions on Computers*, **48**(4):417 –441, 1999. 31, 34, 35

[81] N. Kandasamy, J.P. Hayes, and B.T. Murray. **Transparent recovery from intermittent faults in time-triggered distributed systems**. *IEEE Trans. Computers*, pages 113–125, 2003. 32, 33, 35, 50, 70

[82] Arshad Jhumka, Stephan Klaus, and Sorin A. Huss. **A Dependability-Driven System-Level Design Approach for Embedded Systems**. In *Proceedings of Design, Automation and Test in Europe*, pages 372–377, 2005. 32, 35

[83] Peter van Stralen and Andy Pimentel. **A SAFE approach towards early design space exploration of fault-tolerant multimedia MPSoCs**. In *Proceedings of International conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, pages 393–402, 2012. 32, 35

[84] C. Bolchini and A. Miele. **Reliability-Driven System-Level Synthesis of Embedded Systems**. In *International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pages 35 –43, 2010. 32

[85] Cristiana Bolchini, Antonio Miele, and Christian Pilato. **Combined architecture and hardening techniques exploration for reliable embedded system design**. In *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI*, pages 301–306, 2011. 32

[86] C Bolchini and A Miele. **Reliability-driven System-level Synthesis for Mixed-Critical Embedded Systems**. *IEEE Transactions on Computers*, 2012. 32, 34, 35

[87] Rishad A. Shafik, Bashir M. Al-Hashimi, and Krishnendu Chakrabarty. **Soft error-aware design optimization of low power and time-constrained embedded systems**. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1462–1467, 2010. 32

[88] Paul Pop, Kåre Harbo Poulsen, Viacheslav Izosimov, and Petru Eles. **Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems**. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 233–238, 2007. 32, 33, 70

[89] Adrian Alin Lifa, Petru Eles, and Zebo Peng. **Performance optimization of error detection based on speculative reconfiguration**. In *Proceedings of the 48th Design Automation Conference*, pages 369–374, 2011. 33

[90] OMG. **UML Profile for MARTE**, June 2011. 34

[91] Peter H. Feiler, David P. Gluch, and John J. Hudak. **The Architecture Analysis & Design Language (AADL): An Introduction**. Technical Note CMU/SEI-2006-TN-011, Carnegie Mellon University, 2006. 36

[92] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. **eterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)**. Technical Report UCB/ERL M05/21, EECS, University of California, Berkeley, 2005. 36

[93] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. **Metropolis: an integrated electronic system design environment**. *Computer*, **36**(4):45 – 52, 2003. 36

[94] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng, and Qi Zhu. **A Next-Generation Design Framework for Platform-based Design**. In *DVCon 2007*, 2007. 36, 37

[95] L. Thiele, I. Bacivarov, W. Haid, and Kai Huang. **Mapping Applications to Tiled Multiprocessor Embedded Systems**. In *International Conference on Application of Concurrency to System Design (ACSD)*, pages 29 – 40, 2007. 36, 58, 59, 82

[96] Sanjoy Baruah, Haohan Li, and Leen Stougie. **Towards the Design of Certifiable Mixed-criticality Systems**. In *IEEE Real-Time and Embedded Technology and Applications Symposium(RTAS)*, pages 13–22, 2010. 46, 137

[97] Martin Lukasiewycz, Michael Glass, Christian Haubelt, and Jürgen Teich. **SAT-Decoding in Evolutionary Algorithms for Discrete Constrained Optimization Problems**. In *IEEE Congress on Evolutionary Computation*, 2007. 49

[98] Jia Huang, Jan Olaf Blech, Andreas Raabe, Christian Buckl, and Alois Knoll. **Reliability-Aware Design Optimization for Multiprocessor Embedded Systems**. In *Euromicro Conference on Digital System Design (DSD)*, pages 239 – 246, 2011. 52, 70

[99] Martin Lukasiewycz, Michael Glass, Felix Reimann, and Jürgen Teich. **Opt4J: a modular framework for meta-heuristic optimization**. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation (GECCO)*, pages 1723–1730, 2011. 58

[100] R. Baumann. **The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction**. In *International Electron Devices Meeting (IEDM)*, 2002. 58

[101] Jinkyu Lee, Insik Shin, and Arvind Easwaran. **Online robust optimization framework for QoS guarantees in distributed soft real-time systems**. In *International Conference on Embedded Software (EMSOFT)*, 2010. 70

[102] G. Lyle, S. Chen, K. Pattabiraman, Z. Kalbarczyk, and R. Iyer. **An end-to-end approach for the automatic derivation of application-aware error detectors**. In *IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 584 –589, 2010. 70

[103] D.C.Bossen. **CMOS Soft Errors and Server Design**. In *Reliability Physics Tutorial Notes, Reliability Fundamentals, pp. 121.07.1*, 2002. 71

[104] CMake. **http://www.cmake.org/**. 93

[105] Int. Electrotechnical Commission. **IEC 61131-3: Programmable controllers – Part 3: Programming languages**, 1993. 95, 129

[106] TTTech. **http://www.tttech.com**. 99

[107] Hardik Shah, Andreas Raabe, and Alois Knoll. **Bounding WCET of Applications Using SDRAM with Priority Based Budget Scheduling in MPSoCs**. In *DATE*, 2012. 99

[108] Christian Paukovits. *The Time-Triggered System-on-Chip Architecture*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, December 2008. 100, 101

[109] Haibo Zeng, Wei Zheng, Marco Di Natale, Arkadeb Ghosal, Paolo Giusto, and Alberto Sangiovanni-Vincentelli. **Scheduling the FlexRay bus using optimization techniques**. In *Proceedings of the 46th Annual Design Automation Conference (DAC)*, pages 874–877, 2009. 100

[110] Martin Lukasiewycz, Michael Glass, Paul Milbredt, and Jürgen Teich. **FlexRay Schedule Optimization of the Static Segment**. In *CODES+ISSS*, 2009. 100, 102, 104

[111] Paul Milbredt, Bart Vermeulen, Gökhan Tabanoglu, and Martin Lukasiewycz. **Switched FlexRay Increasing the Effective Bandwidth and Safety of FlexRay Networks**. In *EFTA*, Bilbao, Spain, 2010. 100

[112] T Schenkelaars, B Vermeulen, and K Goossens. **Optimal scheduling of switched FlexRay networks**. In *DATE*, 2011. 100

[113] Martin Lukasiewycz, Samarjit Chakraborty, and Paul Milbredt. **FlexRay switch scheduling - A networking concept for electric vehicles**. In *Design, Automation and Test in Europe (DATE)*, 2011. 100

[114] Wilfried Steiner. **An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-hop Networks**. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 375–384, 2010. 101

[115] N. Ntene and J. H. van Vuuren. **A survey and comparison of level heuristics for the 2D oriented strip packing problem**. In *submitted to Applied Discrete Optimization*, 2006. 109

[116] Leonardo de Moura and Nikolaj Bjørner. **Z3: An Efficient SMT Solver**. In *TACAS*, 2008. 111

[117] P. Milbredt, M. Glass, M. Lukasiewycz, A. Steininger, and J. Teich. **Designing FlexRay-based automotive architectures: A holistic OEM approach**. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 276 –279, march 2012. 116

[118] Dip Goswami, Martin Lukasiewycz, Reinhard Schneider, and Samarjit Chakraborty. **Time-triggered implementations of mixed-criticality automotive software**. In *DATE*, pages 1227–1232, 2012. 116

[119] Altera. **www.altera.com**. 119

[120] SYSGO. **www.sysgo.com**. 119

[121] Festo Didactic. **http://www.festo-didactic.com/**. 127

[122] Simon Barner, Michael Geisinger, Christian Buckl, and Alois Knoll. **EasyLab: Model-Based Development of Software for Mechatronic Systems**. In *IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications*, pages 540–545, October 2008. 129

[123] J. Weigmann and G. Kilian. **Decentralization with PROFIBUS DP/DPV1**. In *ISBN 978-3-89578-218-3*. 130

[124] Anybus. **http://www.anybus.com**. 130

[125] Matlab Simulink. **www.mathworks.de/**. 132

[126] SIListra Systems. **http://www.silistra-systems.com/**. 133

[127] DDC-I. **DEOS - A Time and Space Partitioned DO-178 Level A Certifiable Family of RTOS Products**. 2011. 137