Applications of scenarios in early embedded system design space exploration

van Stralen, P.

Publication date
2014
Document Version
Final published version

Designing an embedded system is a complex procedure. Not only are there many non-functional requirements, but also current embedded systems become more versatile and multi-functional. Therefore, this dissertation uses scenarios to model the dynamism in embedded systems during the early design space exploration.

Applications of Scenarios in Early Embedded System Design Space Exploration

Peter van Stralen

Applications of Scenarios in Early Embedded System Design Space Exploration

Peter van Stralen

UNIVERSITEIT VAN AMSTERDAM

Applications of Scenarios in Early Embedded System Design Space Exploration

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor promoties ingestelde
commissie, in het openbaar te verdedigen in de Aula der Universiteit
op vrijdag 31 januari 2014, te 13:00 uur

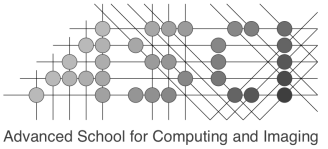
door

Peter van Stralen

geboren te Heerhugowaard.

Promotor:	Prof. C.R. Jesshope
Co-promotor:	Dr. A. D. Pimentel
Overige Leden:	Prof. dr. ir. C.T.A.M. de Laat
	Prof. dr. ir. G.J.M. Smit
	Prof. dr. H. Corporaal
	Prof. dr. ir. J. Teich

Faculteit der Natuurwetenschappen, Wiskunde en Informatica



This work was carried out in the ASCI graduate school. ASCI dissertation series number 291 and funded by the NWO project 612.063.715 *Embedded Adaptive Streaming Systems*.

Cover: Image courtesy by `http://www.flickr.com/photos/cfaobam/`
Printed and bound by Off Page Amsterdam
ISBN: 978-94-6182-394-6

Contents

Part I

Background

1 Introduction

In the year of 2011 more than ten billion CPU cores were sold. This huge amount of cores, which is higher than the world population, clearly shows that CPU cores are a major part of our daily world. Most of these CPU cores, however, are hidden inside embedded systems: all kind of devices that contain processors to support and extend the functionality of the system. Examples of embedded systems can be found everywhere: mobile phones contain a processor to perform the communication and multimedia operations. Similarly, cars incorporate processors to control systems like adaptive cruise control or ABS. From these embedded system examples, many embedded systems are not directly clear from the device itself: household appliances like the microwave, washing machines and dishwashers also contain processors. Next to consumer electronics, embedded systems are also quite pervasive in industrial applications. In this case, we do not talk about small devices, but about large machines that control the production process of products like cars and (computer) chips. Even a traditional branch like agriculture gets more and more dependent on embedded systems.

The design of such systems is a challenging business. In most cases the design is purely application driven and many design objectives are present. Not only performance, power and production costs play a role, but also metrics like memory requirements, battery lifetime, area and security. This is in contrast to the commodity computer market where the number of objectives typically is limited to a few objectives like performance and cost. Most of the objectives involve a trade-off between benefit and cost. A higher frequency for a processor may, for example, lead to an improved performance (benefit), but it also reduces battery lifetime (cost). Related to the trade-off between benefit and cost is the problem of the conflicting objectives. The earlier example of the processor frequency already showed the conflicting nature of the performance and the battery lifetime. In most cases a higher performance will result in a lower the battery lifetime. The large number of, potentially conflicting, objectives for embedded systems complicates the embedded system design severely.

1.1 Heterogeneous MPSoCs

To design a system that is able to support the large number of design objectives of modern embedded systems, heterogeneous architectures are used as a basis for the embedded systems. Heterogeneous architectures can contain a wide range of tunable components:

- **General-purpose microprocessors:** At first, a general-purpose microprocessor core can be used for the computation. Examples of microprocessors are MIPS and ARM cores. Each of these individual cores is highly tunable. For example, a simple MIPS core can be used, but also a complex out-of-order MIPS processor.

- **Soft microprocessors:** Related to the general-purpose microprocessors are soft microprocessor cores. Such soft cores are microprocessors that can be synthesized on programmable logic. As a result, it can be used as a building block of a heterogeneous architecture. There are many types of soft microprocessors each with their own characteristics (performance, power, cost, etc.).

- **Dedicated hardware components:** Programmable logic can also be used for a dedicated function. Some operations in the embedded system may be used so often that it pays off to have a special component that is specialized in that function. Examples are *Application Specific Integrated Circuits (ASICs)* specialized for Discrete Cosine Transform (DCT) operations or Variable Length Encoding (VLE). Although these ASICs are only capable of executing a limited number of tasks, the cost of running this limited set of tasks is lower than it would be on the microprocessor (higher performance, lower power, etc.).

- **Communication:** Different types of communication structures can be used. Shared communication buses may be cheap, but a crossbar provides better throughput and / or latency.

- **Memory subsystems:** Memory subsystems also allow for different trade-offs. Low latency and low capacity (SRAM) or high latency but also a larger capacity (DRAM).

To combine these architectural components in a system, different packing techniques are available. Traditionally, a printed circuit board was used that placed the different components onto a single board and the connected between the components was done with conductive pathways. Other techniques, like System-in-Package and Package-on-Package also combine the components in a similar fashion, but they do not limit themselves to a single board: a Package-on-Package vertically combines multiple dies. The most common technique is to stack memory packages onto the logic die to increase the memory bandwidth. System-in-Package uses a similar technique, only in this case also passive devices like sensors or antennas can be stacked.

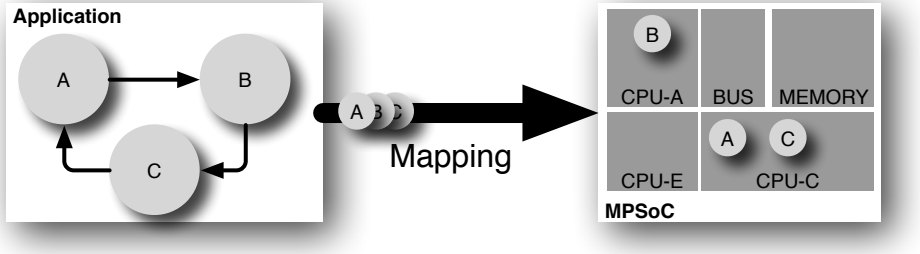


Figure 1.1: *An example of the mapping of an application onto a heterogeneous MPSoC.*

These two techniques also have the benefit that the stacked dies can be developed separately or, can use an off-the-shelf component. These packing techniques, however, do not always provide the best performance. Stacking involves off-chip communication, which is more expensive than on-chip communication. A *System-on-Chip (SoC)* is a packing technique that places all the components on the same die to keep the communication between the components as fast as possible.

Due to the improvement of the lithographic process technology, a SoC has a significant amount of transistors that can be used to implement the different components of a heterogeneous architecture. Moore's law predicts the number of transistors on a single chip doubles roughly every two years. This roughly means that, given the same components, the chip of today can contain twice as many components as two years ago. A result of this trend is that *Multi-Processor System-on-Chips (MPSoCs)* became common for embedded systems [79]. A chip can contain billions of transistors and, therefore, the challenge becomes to use them efficiently.

By composing the architecture from different components and tuning them to the need of the applications that are used in the embedded systems the constraints of embedded systems can be met. The number of potential designs of a heterogeneous system, however, is enormous.

1.2 Design Space Exploration

The set of the potential embedded system designs are called the design space. In the previous section, we already addressed the large potential number of designs for heterogeneous architectures. For an embedded system design, this is not the only degree of freedom. All of the applications in the embedded system must also be mapped onto a heterogeneous architecture. An example for such a mapping is shown in Figure 1.1. The selected MPSoC contains three microprocessors, a communication bus and a shared memory. For each of the processes in the application (A, B and C), one of the processors is selected to execute the process.

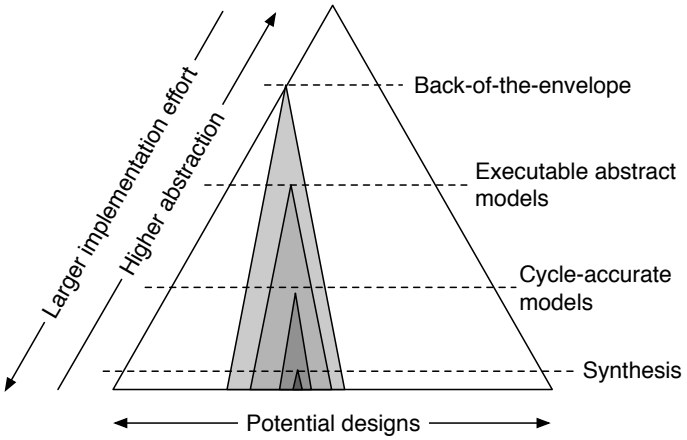


Figure 1.2: *The system level design pyramid.*

Just as the potential set of heterogeneous MPSoCs, the number of potential mappings is large. For the given toy application in Figure 1.1 there may be only 27 possible mappings, but the number of potential mappings grows exponentially with respect to the number of processes. Therefore, the design space of potential embedded system designs is huge and it can only be efficiently explored using a so-called design space exploration.

1.3 System Level Design

To automate the design space exploration of an embedded system a design methodology is required. A design methodology provides a description of a structured approach to handle the complex task of embedded system design. One example of a design methodology is platform-based design. During embedded system design, platform-based design efficiently copes with the large design space by making a clear distinction between two phases: 1) the design of a platform and 2) the adaptation of a platform for a specific embedded system. Hence, a platform is only a partial definition of the final system that contains elements like hardware and software components, interfaces and APIs. An important aspect is that it is designed with flexibility in mind to make it suitable for a range of applications. The advantage of a flexible, but powerful platform is that it can be manufactured in large volumes. This reduces the cost per system. After the manufacturing, the platform can be specialized for a wide range of applications. These applications may be sold in small volumes and still benefit from the reduced production prices of the platform. As a consequence, a platform leverages the high costs of the chip manufacturing with the quality of the range of embedded systems it is meant for.

As discussed before, a platform already fixates a part of the design decisions: the

components that are available. These components, however, can still contain some flexibility: different types of microprocessors may be available, but it can also be the case that dedicated hardware is present. For the embedded system designer the challenge becomes to select a platform instance and to make the remaining design decisions: on which components is the application mapped and, given that a component has flexibility, how are the individual components configured. In our case, the platform instance is defined by the mapping that was introduced in the previous section.

Platform-based design eases the design of an embedded system by splitting the complex design in the design of the platform and the specialization of the platform for the embedded system. The design of a platform is hard, but it can be reused for different embedded systems. Specialization of a platform is not trivial either. To deal with both the platform design and specialization, a system level design methodology is used that considers the system at different levels of abstraction. More specifically, system level design takes a system specification and, via multiple abstraction layers, it will end up with a system implementation that matches the provided design constraints. The higher the abstraction level, the less details are required to model the design instance. In this way, system level design tries to minimize the design effort as much as possible. This is especially important for embedded system with a stringent time-to-market requirement such as mobile phones.

An illustration of system level design is given in Figure 1.2 [75, 74]. Horizontally the level of abstraction is shown, where a higher level in the pyramid corresponds to a higher level of abstraction. A higher abstraction level also involves less implementation details. Hence, the number of potential designs is smaller. For this purpose, the width of the pyramid symbolizes the number of potential designs at the given abstraction layer. The designs at the base of the pyramid correspond to the system implementations. Generally, the system level design makes the design more efficient by iteratively reducing the set of potential designs. At the highest abstraction level, the first decisions are made in order to reduce the number of potential design points. These high level decisions are used to determine which of the design points are explored at a lower abstraction level. This process is repeated until the design methodology ends up at the complete system implementation at the base of the pyramid.

In our example, four different abstraction layers are shown. At first, back-of-the-envelop calculations are done to manually make some predictions about the embedded system. Results from this phase are very general, but they can be used to specify the initial design space boundaries. Next, abstract execution models and cycle accurate simulation models further bound the design space. This process of discarding unsuitable designs is also called design space pruning. Figure 1.2 also visualizes this pruning by the filled pyramids inside the large pyramid. Each of these pruning pyramids bounds the number of designs that is explored by the abstraction layer below it. The main rationale of the bounding of design points is the increased analysis and

implementation effort of the lower abstraction layers. An abstract execution model may, for example, evaluate a design within a second, whereas an evaluation at the Register Transfer Level (RTL) level can take days to complete.

1.4 Problem Definition

The current trend is that the dynamism within embedded system increases. A first example of the dynamism can be found in the application workloads of embedded systems. Often, this application workload does not consist of a single algorithm, but it contains multiple algorithms to make an embedded system multi-functional. Each of these algorithms performs a different task for the embedded system. Due to this dynamic application workload, the application behavior is not static, but it changes over time. This dynamism may manifest in several ways: amount of computation, memory bandwidth and access patterns. For each of the different behaviors, the application demands different qualities of the architecture. As a result, different architectural components can be used to support the application workload in the different phases of its execution.

Next, the dynamism in the MPSoC architecture is also increasing. One of the driving forces is the decreasing technology scale. In 2012, 22nm chips were available for consumers and the international technology roadmap for semiconductors [31] even predicts that around 2020 the technology scale will be around 10nm. One of the disadvantages of smaller technology scales is the unreliability of the architecture due to its increased susceptibility to faults. The closer the size of a silicon-based transistor is to the size of a real atom, the higher the probability that a fault affects the computation of a chip. The presence of faults in the architecture may result in failures of the embedded system.

This thesis will address the dynamism during the design of the embedded system. To capture the dynamism in the embedded system so-called scenarios are used. Workload scenarios will be used for the dynamism in the application workload, whereas the dynamism of the unreliable architectures will be described in so-called architecture scenarios. Our aim is to use these scenarios to design an embedded system that is capable to cope with the dynamism in the embedded system. For this purpose, we try to find optimal mappings (Figure 1.1 gives an example of a mapping) that optimize the system objectives for all potential scenarios. As a result, a static mapping is obtained that, on average, is able to deal with the dynamism in the embedded system as good as possible. Therefore, the research question of this thesis is:

How can scenarios be used to enhance the design space exploration of dynamic embedded systems?

1.5 Overview and Contribution

The work presented in this thesis has been performed in the context of the Sesame system-level simulation framework [57]. As will be discussed in Chapter 2, this framework is able to evaluate non-functional requirements (like performance, power and cost) of an embedded system at a high level of abstraction. Therefore, it is able to quickly perform an early design space exploration. So far, Sesame only took static application workloads and static architectures into account. In order to be able to address the increasing dynamism in the embedded systems, this thesis will investigate how scenarios can be applied to statically design an embedded system that is able to cope with all the dynamism in an embedded system. The main contributions of this thesis are:

- Introduction and storage of inter- and intra-application scenarios and their automatic discovery in applications.
- Presentation of a scenario-based design space exploration approach to model embedded systems with dynamic multi-application workloads. A dynamic multi-application workload has many possible behaviors for which the embedded system needs to be optimized. The scenario-based design space exploration uses fitness prediction techniques to enable a quick evaluation of the average quality of the embedded system under all the possible workload scenarios.
- An extension of the mapping methodology of Sesame that takes the fault tolerance of embedded systems into account. The mapping of Sesame is based on a separation of concerns of the application, the architecture and the mapping. To transparently model the fault tolerance, a separate layer is introduced to automatically add fault tolerance mechanisms to applications.
- The design and implementation of SAFE: A Sesame Automated Fault-tolerant Explorer that identifies (sub-)optimal fault-tolerant mappings. The fault-tolerant mapping performs two tasks: make an application fault tolerant and map the fault-tolerant application onto the architecture. The exploration framework needs to explicitly take these steps into account.

To summarize, this thesis studies how scenarios can be used to model the dynamism of embedded systems during the early design space exploration. Both the dynamic multi-application workloads and the fault behavior of an embedded application are examples of scenarios where the dynamism of the embedded system is explicitly modeled. The dynamic multi-application workload is an example of an application scenario as it shows the dynamism inside and between applications. Fault tolerance, on the other hand, is related to the dynamism in the architecture due to (temporal) unreliability of its computations. The thesis is, therefore, split into three parts: 1) background (Chapters 1 and 2), 2) application scenarios (Chapters 3, 4 and 5) and 3) architecture scenarios (Chapters 6 and 7).

In the background part, the work is placed into its context by giving the theoretical preliminaries that are used later on. Chapter 2 discusses embedded system design space exploration by using a multi-objective evolutionary algorithm. This chapter includes a description of the Sesame framework to model embedded systems. Next, a description of evolutionary algorithms is given. An evolutionary algorithm is a heuristic search method to find the best set of embedded system design. To compare several sets of embedded system designs, the end of Chapter 2 discusses several comparison techniques.

Chapter 3 is the first chapter of the part on application scenarios. Therefore, it introduces the concept of application scenarios. First, a general description is given that explains how application scenarios model the dynamic behavior of applications. Next, the detection of application scenarios is discussed. The detected application scenarios need to be stored in a so-called scenario database. As the number of potential application scenarios may be large, this storage must be as efficient as possible. In order to show the storage efficiency of a scenario database, a number of experiments show the performance of the scenario storage for four real world applications.

A next step is to discuss how the application scenarios can be utilized to explore the design space of embedded systems with dynamic multi-application workloads. Chapter 4 will introduce an exploration framework that searches for (sub-)optimal mappings of embedded systems that statically optimizes the mapping for all the possible application scenarios. As the number of application scenarios is exponentially related to the number of applications, it is infeasible to exhaustively evaluate the explored mappings using the complete set of application scenarios. Therefore, a representative subset of scenarios is used to predict the quality of the mappings. The chapter ends with some case studies of scenario-based design space exploration.

This dynamic selection of the representative subset of scenarios is investigated in further detail in Chapter 5. Chapter 5 formalizes the scenario-based design space exploration that was presented in Chapter 4 and focuses on fitness prediction techniques. These fitness prediction techniques select the representative subset of scenarios using three different techniques: a genetic algorithm, a feature selection algorithm and a hybrid approach. The experiments will investigate several aspects of the fitness prediction techniques: the efficiency of the different methods, the effect of the subset size on the outcome of the design space exploration and the subset quality during the design space exploration.

The third part of this thesis on architecture scenarios starts with a chapter (Chapter 6) that introduces fault-tolerant mappings as part of the Sesame Automated Fault-tolerant Explorer. A fault-tolerant mapping aims to map a normal application onto an architecture that is resilient to transient faults. Transient faults are faults that temporarily disrupt the normal computation of the architectural processors in the MPSoC. As a result, processors can produce wrong output or no output at all. A fault-tolerant embedded system should be able to deal with these

faults in order to prevent erroneous output. Therefore, a fault-tolerant mapping first applies fault tolerance patterns onto the application to transform the application into a fault-tolerant application and then maps this transformed application onto the architecture. By integrating fault tolerance into the mapping, the reliability of the embedded system can be explored during the early design stages as one of the objectives.

Chapter 7 will focus on the definition of an architecture scenario and how these architecture scenarios can be used to search for a fault-tolerant embedded system. To this end, a complete framework is developed that both explores the design of (sub-)optimal fault-tolerant mappings and simultaneously picks a representative subset of architecture scenarios. One of the features of this framework is that for each mapping the objectives are evaluated for different reliability classes. A reliability class is used to obtain the worst case system objectives of the dynamic architecture with a given specific certainty. The certainty describes the probability that the number of faults is lower or equal to a specific value. A low certainty means that the number of faults is low and, therefore, the worst-case objectives are low. When the certainty becomes higher, the number of potential faults also becomes higher. As a result, the worst case system objectives are also increasing. The given case study shows how these reliability classes can be used to observe the capability of mappings to deal with transient faults in the architecture.

Finally in Chapter 8, we will look back and summarize what we have achieved, and then look ahead what could be accomplished next.

2 Embedded System Design using a MOEA

Multi-Objective Evolutionary Algorithms (MOEAs) aim at solving multi-objective optimization problems. Quite often the objectives in these problems are conflicting in nature. Take, for example, the cost and performance within embedded system design. Generally, the design with the highest performance is also the most expensive one. Since embedded system design is concerned about both requirements, a trade-off must be made during the search for the most optimal design.

MOEAs were initially developed in the eighties and since then applied to many scientific and engineering problems. In this thesis, a MOEA will be applied to support the embedded system design. The first section of this chapter will discuss Sesame [57]. Sesame is a simulation framework that is able to model MPSoC design instances and analyze their quality. Section 2.2 will describe *Genetic Algorithms (GAs)*. A GA is a search algorithm that can be used to optimize the solution to a certain problem. It is the technique that we have chosen to optimize embedded system design with Sesame. A search will result in a set of (sub-)optimal solutions and Section 2.3 will describe the different metrics that can compare different sets of solutions. Finally, a short conclusion is given.

2.1 Sesame

Sesame [57] is an abbreviation for "Simulation of Embedded Systems Architectures for Multi-level Exploration". It supports a designer to take early design decisions by providing an evaluation mechanism for design space exploration.

One of the main features is the *separation of concerns* that simplifies the development of an embedded system with Sesame. As concerns are separated, they can be developed individually. This also enables the reuse of the individual concerns, which is one of the main features of platform-based design (see Section 1.3).

The concerns in the Sesame model are visualized in Figure 2.1 and the Sesame model contains three components (i.e., concern):

- Application Model
- Architecture Model

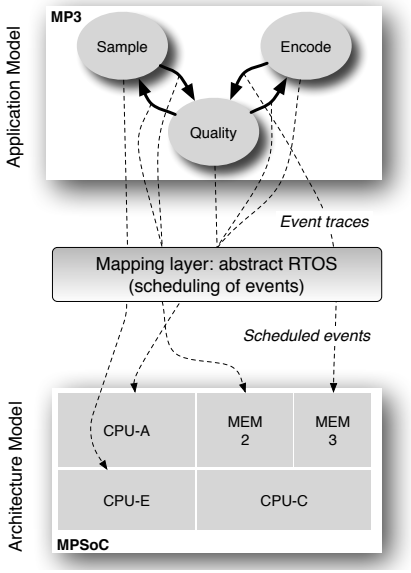


Figure 2.1: *The Sesame model.*

- Mapping Layer

The application model describes the functional behavior of the embedded system. In the example of Figure 2.1 an MP3 application is shown. Since the application description is only functional, the model is limited to the description of how the MP3 application works. It does not describe timing and real time requirements.

Non-functional requirements like timing are described in the architecture model. The architecture model contains the components of the MPSoC including their non-functional behavior. Non-functional requirements in the example of Figure 2.1 may be the frequency of the processor CPU-A, CPU-C and CPU-E, the voltage, but also the number of cycles required to execute functions of the MP3.

To connect the application model and the architecture model, a mapping layer is used that connects the application components to architecture components. During the simulation, the application will emit events that describe what the application is doing. These functional behavior descriptions are processed by the architecture components in order to obtain the non-functional requirements of the mapped application running on the given architecture. In the coming subsections the individual layers are described in more detail. Afterwards, the final subsection will show an example of a Sesame model to illustrate the way Sesame is used to judge the quality of an application mapping.

2.1.1 Application Model

To describe the functional behavior of an application, Sesame uses *Kahn Process Networks (KPNs)* [36] as the *Model of Computation (MoC)*. A KPN is defined as a network of concurrently running processes. The only way of communication between the processes is by Kahn channels. Kahn channels are one-directional FIFO buffers with an unbounded capacity. The only operations allowed are a read and a write operation. As the channels are unbounded, the write operation is non-blocking. The read operation, on the other hand, is blocking since data may not yet be available. Other operations, like testing if data is available on the channel, are not allowed.

Since communication can only be done over the Kahn channels, there is no notion of shared memory. Hence, the only way a process can influence another process is by sending tokens on the Kahn channels. The example MP3 application in Figure 2.1 has three Kahn processes (SAMPLE, ENCODE and QUALITY) and four Kahn channels. Important to notice is the possibility of having multiple channels between processes.

The choice of using KPNs is motivated by the target application domain of Sesame. Currently, it is mainly focused on the multimedia and signal-processing domain. These kinds of applications mainly have a data-flow style of processing. A data-flow program has a stream of data that is passing though a series of actors. Each actor processes the data and passes it on to the next actor. KPNs fit well to this type of processing. Another desirable property of KPNs is determinism. The order of the tokens sent over the channels does not depend on the order the Kahn processes are scheduled by the simulation. As a result, Sesame will produce the same output regardless of the scheduling of the processes or the architectural characteristics.

Internally, the Kahn processes may be implemented in any high level programming language, as long as the Kahn semantics are observed. In order to emit events to the architectural model, the Kahn processes are annotated. Currently, Sesame has the following type of events:

- READ(channel, size)
- WRITE(channel, size)
- EXECUTE(opcode)
- QUIT

Each event has a set of arguments to describe what is done. The argument of EXECUTE describes which operation is performed. Examples are 'discrete cosine transform' and a 'convolution operation'. For READ and WRITE events, the Kahn channel is given and the amount of data that is communicated as, depending on the application, reading / writing may involve different communication units like a pixel or a complete video frame. Finally, the QUIT event signals the end of an event stream.

2.1.2 Architecture Model

The architecture model describes the hardware components of the system. It is responsible for modeling non-functional properties, like latencies and energy, associated with the components of the system. Hardware components are described using the Pearl language [78]. Pearl is a C-based discrete event simulation language with two main features: 1) an object oriented approach for defining model components and 2) integrated communication and synchronization primitives

Model components are defined in a similar fashion as C++ classes. An important property is that each component has its own execution environment. Components cannot directly access other components. For this purpose, a kind of *Remote Procedure Calls (RPCs)* is used. Pearl uses two types of RPC calls 1) synchronous and 2) asynchronous. In case of a synchronous call, the component halts until the other component returns a value. In the asynchronous case, however, the calling component can continue execution.

The Pearl compiler will generate a binary that can process the event traces that are emitted by the application model. This can be done via different interfaces. In this thesis, we only used the file interface. The application will write the emitted events to a trace file that is read in by the Pearl binary. As a Pearl binary reads in both the mapping and the trace file, the trace file only needs to be generated once for each application. For a different mapping, only the mapping file needs to be changed. The Pearl binary uses a discrete event simulator to process the trace events. As a result of the simulation both system wide statistics (like execution time and energy) and user-defined statistics are obtained.

2.1.3 Mapping Layer

Because of the separation of the application and the architecture model, an explicit mapping layer is required to connect the individual models. The main purpose of the mapping layer is to gather the events from the application model and schedule them to the architectural components. Not only the Kahn processes must be mapped, but also the Kahn channels.

A more detailed view of the mapping layer is given in Figure 2.2. To connect Kahn processes to the architecture, each Kahn process is assigned a virtual processor. Similarly, each Kahn channel is assigned a virtual channel. For this reason, the mapping layer is also called the virtual layer. Connecting the application layer to the virtual layer can be done implicitly as the projection from the application layer to the architectural layer is bijective. In a bijective projection each element has its own private individual virtual element and, therefore, this element can be generated automatically. For connecting the virtual layer to the architecture layer, however, an explicit connection is required. There is a many-to-one relation between the virtual layer and the architectural layer. A virtual element may only be connected to one architectural component, but an architectural component may be connected

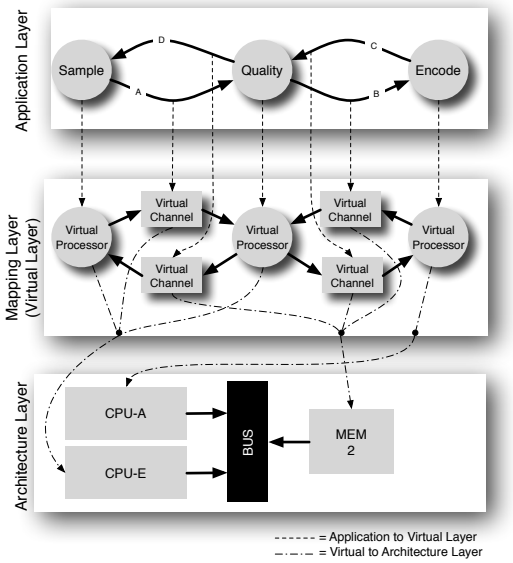


Figure 2.2: *The different layers in the Sesame model and their connections.*

```
<?xml version="1.0" encoding="UTF-8"?>
<mapping xmlns="http://sesamesim.sourceforge,net/YML_Map"
         side="source" name="application">
  <mapping side="dest" name="architecture">
    <map source="Sample" dest="CPU-E" dtmpl="vproc"/>
    <map source="Quality" dest="CPU-E" dtmpl="vproc"/>
    <map source="Encode" dest="CPU-A" dtmpl="vproc"/>
    <map source="Sample.A->Quality.A" dest="CPU-E" dtmlpl="vself"/>
    <map source="Quality.B->Encode.B" dest="MEM-2" dtmlpl="vmem"/>
    <map source="Encode.C->Quality.C" dest="MEM-2" dtmlpl="vmem"/>
    <map source="Quality.D->Sample.D" dest="MEM-2" dtmlpl="vmem"/>
  </mapping>
</mapping>
```

Figure 2.3: *Mapping YML*

to multiple virtual elements.

To describe a mapping, the *Y-chart Modeling Layer (YML)* is used. YML is an XML-based language that is capable of describing process networks and is also used to describe the application and the architecture layer. Figure 2.3 shows the YML file for the example mapping in Figure 2.2. The SOURCE parameters are the application processes (like SAMPLE and QUALITY) and the DEST parameters are components from the architecture (like CPU-A and MEM-2). Finally, the DTMPL field describes the template for virtual component. In this example, VPROC is a virtual processor and VSELF and VMEM are virtual channels.

Each virtual processor must be mapped to an architectural processor. A Kahn

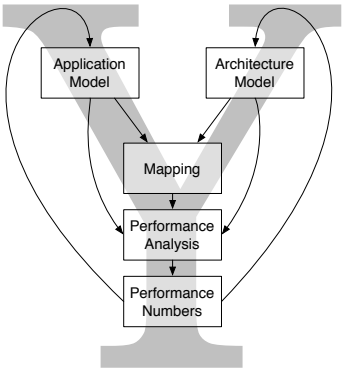


Figure 2.4: *Y-chart design methodology.*

channel, however, can be mapped to either an architectural processor (e.g., template VSELF) or an architectural memory (e.g., template VMEM). In the example of Figure 2.2, channel A is mapped onto processor CPU-E. This is only possible because both the reader and writer of the channel (SAMPLE and QUALITY) are both mapped to the same processor.

In case multiple elements are mapped onto the same component, scheduling is required. Sesame supports many types of schedulers, both static and dynamic. Static schedulers define the ordering of the events of the application layer before the simulation is started. Dynamic schedulers, on the other hand, use the order of arrival. In this thesis, only the dynamic *First-Come-First-Serve (FCFS)* scheduler will be used.

2.1.4 Design Space Exploration

In the previous section, the Sesame model was described. Given the application model, the architecture model and the mapping description, it is able to predict the quality of the MPSoC instance. There are, however, many potential design possibilities for all of the layers. Therefore, Sesame adopted the Y-chart exploration method [38]. The Y-chart exploration method, as illustrated in Figure 2.4, is an iterative way of refining an MPSoC design. The *Design Space Exploration (DSE)* starts with a full definition of an *MPSoC design instance*: an application model, an architectural model and a mapping. Given the definition, Sesame can evaluate the performance and obtain the performance numbers. Depending on the performance numbers (i.e., the quality of the design) the Sesame model can be adapted. This can be a different mapping, but also a different application and / or architecture model.

In this thesis, the application and the architecture models remain fixed. Therefore, the term mapping is used interchangeably with MPSoC design instance. Although the architecture remains fixed, the DSE is able to explore different MPSoCs. For this purpose, the meta-platform approach is used. The architectural platform

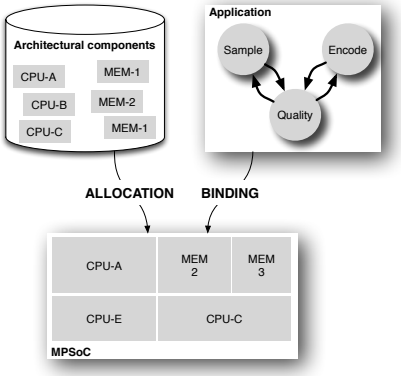


Figure 2.5: *Mapping implicitly describes allocation and binding.*

that is described may actually be larger than actually fits on the final platform. Therefore, the mapping implicitly consists of two aspects: 1) *allocation* and 2) *binding*. This approach is illustrated in Figure 2.5. Allocation selects the architectural components for the MPSoC platform architecture. Only the selected components will be placed on the MPSoC platform. These components will include processors, memories and supporting structures like crossbars and communication buses. Subsequently, the binding specifies which application task or application communication is performed by which MPSoC component.

The number of potential mappings is enormous. In the trivial example of Figure 2.2 there are already 50 valid mappings. The example in Figure 2.1 has no less than 2187 potential mappings. If the application of Figure 2.1 contains twice as much processes and communication channels (i.e., six processes and eight channels), the number of potential mappings grows exponentially to more than 4.7 million. For an application thrice as big, the number of potential mappings grows to a number no less than 10.4 billion. It is infeasible to exhaustively search within such a huge amount of mappings. That is why an efficient search technique is required.

In the coming subsections, the DSE is described in more detail. The first subsection will describe the objectives that are taken into account during the DSE. Next, our search algorithm of choice is described: the genetic algorithm. Finally, metrics are described to compare outcomes of the different runs of the search algorithm.

2.1.5 Objectives

As discussed earlier, the main purpose of Sesame is to predict the quality of an MPSoC design instance. In general, Sesame supports all kinds of statistics. In this subsection the objectives will be described that will be used in this thesis.

Figure 2.6 shows an example of the output Sesame gives as a result of a simulation. In this output there are two important fields: elapsed time and used energy. Elapsed

```
Elapsed time: 187195 units
Used energy: 18986100 units
KPN application.architecture.memory.statistics()
  Reads 4400
  Writes 4400
  Read time 44000
  Write time 44000
  Idle time 99195
KPN application.architecture.p0.statistics()
  Channels 1
KPN application.architecture.p0-Total units read 4400
KPN application.architecture.p0-Total units written 4400
KPN application.architecture.p0-Time spent reading 11110
KPN application.architecture.p0-Time spent writing 21905
KPN application.architecture.p0-Time spent executing 22000
KPN application.architecture.p0-Idle time 132180
KPN application.architecture.p1.statistics()
  Channels 1
KPN application.architecture.p1-Total units read 4400
KPN application.architecture.p1-Total units written 4400
KPN application.architecture.p1-Time spent reading 11060
KPN application.architecture.p1-Time spent writing 11000
KPN application.architecture.p1-Time spent executing 165000
KPN application.architecture.p1-Idle time 135
KPN application.architecture.p2.statistics()
  Channels 2
KPN application.architecture.p2-Total units read 4400
KPN application.architecture.p2-Total units written 4400
KPN application.architecture.p2-Time spent reading 22030
KPN application.architecture.p2-Time spent writing 11010
KPN application.architecture.p2-Time spent executing 49500
KPN application.architecture.p2-Idle time 104655
REAL TIME of simulation: 0.04 secs.
```

Figure 2.6: *Raw output of a Sesame simulation.*

time gives the total number of simulated cycles required to perform the complete application workload. This is the *execution time* of the MPSoC design instance.

Next to performance prediction, Sesame is also extended to support *energy* prediction [86]. The used energy field gives the predicted amount of energy based on an abstract activity-based model:

$$E = \sum_{r \in \mathrm{R}} E_r \tag{2.1}$$

$$E_r = t_{\text{r-idle}} * P_{\text{r-idle}} + t_{\text{r-busy}} * P_{\text{r-busy}} \tag{2.2}$$

Thus, the total energy consumption of the system is the sum of the energy consumption of the individual architectural resources $r \in \mathrm{R}$. For the individual energy consumption of each resource r the more detailed statistics of Sesame are used. Take, for example, processor P2 in the example of Figure 2.6. This processor has an idle time ($t_{\text{r-idle}}$) of 104655 cycles. As the total number of cycles is 187195, the number of busy cycles ($t_{\text{r-busy}}$) is 82540 cycles. Given the idle and busy power usage (P_{r-idle}, P_{r-busy}), that are provided explicitly in the architecture model, the energy of processor P2 can be obtained:

$$\begin{aligned} E_{\text{p2}} =& t_{\text{p2-idle}} * P_{\text{p2-idle}} + t_{\text{p2-busy}} * P_{\text{p2-busy}} \\ =& 104655 * 5 + 82540 * 30 \\ \approx& 3.0 \times 10^6 \end{aligned}$$

Finally, the *cost* of the system is taken into account. The cost will be defined in terms of area. In the architectural model, the cost of each individual architectural resource is described. The total cost of a system is the sum of the cost of all allocated components.

2.2 Genetic Algorithms

The goal of the DSE is to find (sub-)optimal mapping with respect to execution time, energy and cost. For this multi-objective optimization problem many potential search algorithms exists [10]. Examples are genetic algorithms, simulated annealing, tabu search, ant colony optimization and particle swarm optimization. We have chosen for the genetic algorithm. Most of the other approaches have difficulty to keep the diversity in the search population [10]. Additionally, some of them are computationally too expensive. An example is the tabu search that performs a local search (which may lead to a lower diversity) in the neighborhood of the current design instance. The size of this neighborhood, however, is exponential with respect to the number of application elements.

Genetic Algorithms [45] are able to maintain the diversity in a population and were invented by John Holland in the 1960. A GA provides a way to search through

a huge number of possibilities. There are many examples in literature where a GA is successfully used. Still, care must be taken when using a GA. Although there is no hard set of conditions for deciding if a GA is applicable, the following conditions play a role for successfully using a GA:

- The search space must be large.
- The search space is not well understood.
- The fitness function is noisy (similar individuals may have a completely different fitness value).
- The DSE does not search for a global optimum, but only for a solution that meets the requirements.

The search space of the embedded system DSE perfectly complies with these properties. Individuals may be close in the search space (i.e., the mapping is mostly the same), but the change of a single task mapping may have a drastic effect on the global quality of the MPSoC design instance. On top of that, due to the short time-to-market, designers often search for acceptable solutions. These solutions not necessarily need to be the most optimal as long as they fulfill the system requirements.

2.2.1 Biological Terminology

Genetic algorithms are based on natural mechanisms as they mimic natural evolution. Therefore, it is useful to introduce some biological terminology. As a basis for natural evolution a population of individuals is used. An individual (a mapping in our case) is a potential solution to the problem we are trying to solve. In nature it is survival, in MPSoC design space exploration it is a high-quality solution.

Each individual is described using a *chromosome*. A chromosome contains a set of *genes* and it is a blueprint of the individual. Base of the blueprint are the properties that are described by the individual genes. Properties, like eye color, can be assigned different values. These possible values are called *alleles*. For the eye color, for example, the possible alleles are colors like blue and brown. In the MPSoC design space exploration, on the other hand, a gene encodes a task mapping. The position of the gene refers to the application task and the possible alleles are the potential architectural processors on which the application task can be mapped. Designing such a chromosome is a non-trivial task. Not only the choice of the encoding (which alleles are used) have a major effect on the efficiency of the GA, but also the ordering of the genes.

Next, the *genotype* is the complete set of properties an individual. All off these properties may have an effect on the development of an organism to its final characteristics. These characteristics are called the *phenotype* of the organism. The

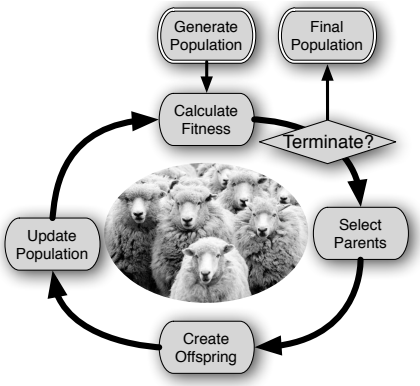


Figure 2.7: *The general procedure for a genetic algorithm.*

phenotype of our DSE problem is a design instance as described by a mapping file like the one shown in Figure 2.3.

A population of individuals does not remain fixed over time. After each generation new offspring is created. During this reproduction, genes of selected parents are combined into a new child chromosome. This combination is called *crossover*. The offspring's chromosome may also change slightly due to *mutation*. Mutation introduces imperfections in the copying of the parent's chromosome. This mutation greatly drives the search to improved individuals. Without it, no new properties would be investigated, as all the properties would be derived from the selected parents. As a result, the search algorithm is not able to escape from local optima: solutions which are better than all neighboring solutions, but that are worse than the global optimum.

The final building block is the *fitness*: the quality of the individual (in our case the execution time, energy consumption and cost of the MPSoC design). Based on the fitness the probability is determined that the individual will be selected to generate the offspring of the next generation. Thus, by the survival of the fittest, the evolution will eventually end up with the high quality individuals.

2.2.2 Procedure

As described in the previous subsection, the genetic algorithm uses the survival of the fittest to search for the best solutions. A basic procedure for a genetic algorithm is illustrated in Figure 2.7. The steps of the procedure are as follows:

- Generate Population: The first step is to create the initial population. Depending on the problem, this population may be created randomly or based on some heuristics.

- Calculate Fitness: To determine the quality of all the individuals in the

population, the individual fitness must be determined. Before this fitness can be obtained, the genotype must be converted into a phenotype. In the search for optimal MPSoC design instances, for example, the list of genes must be translated into an MPSoC design instance. The used fitness function is problem-specific and it analyzes how well the individual solves the specific problem.

- TERMINATION: Generally, a genetic algorithm will terminate after a fixed time or when a satisfactory solution is found. In case the genetic algorithm is not terminated, it will continue with the selection of the parents for the next generation.

- SELECT PARENTS: Selection is of parents is an important aspect in a genetic algorithm. In order to achieve a survival of the fittest, the best parents must be picked to create the offspring for the next generation. However, to maintain diversity it is unwise to only select the individuals with a high fitness. Therefore, selection methods will in general take a good trade-off between both. An example of a selection mechanism is tournament selection. Tournament selection will randomly pick a number of potential parents. From this subset of potential parents the strongest parent is selected. This selection is with replacement, so the same parent may be picked multiple times.

- CREATE OFFSPRING: Next, the offspring chromosomes are created using the chromosomes of the selected parents. Genetic operators will combine the parent chromosomes in such a way that the variance in the population remains intact.

- UPDATE POPULATION: After applying the genetic operators, the offspring individuals are, together with the selected parents, placed in the population that is used in the next generation.

The effectiveness of the procedure is highly dependent on the choice of the parameters. Examples of parameters for the GA are: population size, crossover probability and mutation probability. On top of that, the choice of these parameters cannot be picked individually as they are highly dependent. In general, the best option is to use values that worked quite well in previous cases. A study of Schaffer et al. [62] deduced a set of parameter values that work optimally for a small set of optimization problems. These ranges are the bases of our choice for parameter values.

2.2.3 Genetic Operators

Genetic operators are an important part of a genetic algorithm. Their purpose is to maintain genetic diversity in order to obtain a decent exploration of the design space. There are two types of genetic operators: crossover and mutation (Figure 2.8).

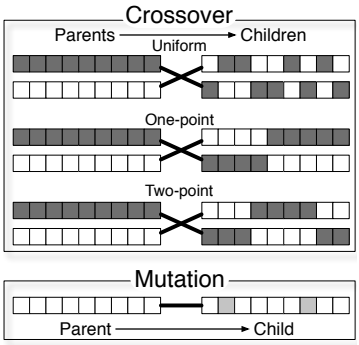


Figure 2.8: *An illustration of some of the genetic operators.*

Crossover

Crossover combines two parent chromosomes to create two new chromosomes. Figure 2.8 shows three flavors of this procedure: 1) one point crossover, 2) two point crossover and 3) uniform crossover. One point crossover randomly selects a position within a chromosome. Every gene before this position is swapped at the chromosomes for both of the children. One of the main shortcomings of this approach is the positional bias. The first genes have a lower probability to be exchanged, whereas the last genes have a higher probability to be exchanged. As a result, properties in the first genes are changed less often. A simple solution is to use two point crossover. Two point crossover selects two random positions. Between these two positions the genes are exchanged. This approach still has some positional bias. An approach without any positional bias is the uniform crossover. In this approach the gene at each position exchanges with a certain probability.

The question of which crossover operator needs to be picked is both problem dependent and chromosome dependent. For certain problems a one / two point crossover works better as a complete range of genes is exchanged, whereas other problems benefit from the uniform pick of the exchanged genes.

Mutation

Next to crossover, mutation is used. Without mutation, new properties could not be introduced in the population and the genetic diversity of the population would reduce over time. Mutation can randomly alter a subset of the genes in the chromosome. If the variation is successful with respect to the fitness, the individual will be selected for the next generation. Otherwise, it will likely be discarded.

2.3 MOEA Performance Measuring

The previous section introduced the GA. One of the highlighted issues was the difficulty of selecting the parameters for the GA. As this is mostly done in an empirical fashion, we need a way of comparing the outcomes of a GA.

A GA is an example of a *Multi-Objective Evolutionary Algorithm (MOEA)*: it is a heuristic search method that maintains a population of solutions to explore the design space of optimal solutions given multiple objectives. Formally, a MOEA is a technique that solves a *Multi-Objective Optimization Problem (MOP)* by looking for the set of optimal solutions. Given a set of m decision variables (the degrees of freedom in the problem specification), a fitness function must optimize the n objective values. The fitness function is defined as:

$$f_i : R^m \rightarrow R^1 \tag{2.3}$$

A potential solution $x \in R^m$ is an assignment of the m decision variables. The fitness function f_i translates a point in the solution space X into the i-th objective value (where $0 \leq i < n$). As a result, the combined fitness function $f(x)$ translates a point in the solution space into the objective space Y. The formal definition of a MOP tries to identify a solution that minimizes the objective values:

Definition 1 (Multi-Objective Optimization Problem) *A multi-objective optimization problem is an optimization problem with m decision variables and n objective functions:*

$$\begin{aligned} \textit{Minimize } y &= f(x) = (f_1(x), f_2(x), ..., f_n(x)) \\ \textit{Where } x &= (x_1, x_2, ..., x_m) \in X \\ y &= (y_1, y_2, ..., y_n) \in Y \end{aligned}$$

Without loss of generality a minimization procedure is assumed. A maximization problem can be converted into a minimization problem by multiplying the fitness value y_i with -1. Hence, with a MOP minimization and maximization can be mixed. The remainder of this section will discuss several metrics in order to reason about the ability of a MOEA to successfully solve a MOP.

2.3.1 Pareto Dominance

With an optimization of a single objective the comparison of solutions is trivial. A better fitness (i.e., the objective value) means a better solution. With multiple objectives, however, the comparison becomes nontrivial. Take for example two cars: a sports car and a family car. A sports car is generally more expensive than a family car, but it goes much faster. In case there is no preference defined with respect to the objectives and there are also no restrictions for the objectives, one cannot say if the sports car is better or the family car. A MOPs can have many different objectives:

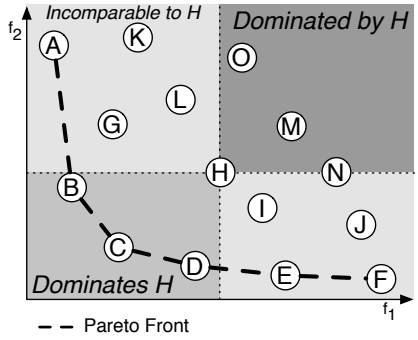


Figure 2.9: *A Pareto front and an example of the dominance relation.*

two additional objectives for a car may be, for example, gas usage and looks. Given the number of objectives, there must be a way to compare different solutions. With a single objective this is trivial, but for multiple objectives the non-trivial comparison may be done using the Pareto dominance relation:

Definition 2 (Pareto Dominance Relation) *A solution $x_1 \in X$ is said to dominate solution $x_2 \in X$ if and only if $x_1 < x_2$:*

$$x_1 < x_2 \iff \forall i \in \{1, 2, ..., n\} : f_i(x_1) \leq f_i(x_2) \wedge \\ \exists i \in \{1, 2, ..., n\} : f_i(x_1) < f_i(x_2)$$

Hence, a solution x_1 dominates x_2 if its objective values are superior to the objective values of x_2. For all of the objectives, x_1 must not have a worse objective value than solution x_2. Additionally, there must be at least one objective in which solution x_1 is better (otherwise they are equal).

An example of the dominance relation is given in Figure 2.9. This example shows a two dimensional MOP. For solution H the dominance relations are shown. Solution H is dominated by solution B, C and D as all of them have a lower value for both f_1 and f_2. On the other hand, solution H is superior to solution M, N and O. Important to notice is that solution N has a similar value for f_2, but the value for f_1 is worse. Finally, some of the solutions are not comparable to H. These solutions are better for one objective but worse for the other.

The Pareto dominance relation only provides a partial ordering. This was already clear in our car example: the sports car is faster, but the family car is cheaper. Similarly, the solutions A to F of the example in Figure 2.9 cannot be ordered using the ordering relation. Since not all solutions $x \in X$ can be ordered, the result of a MOP is not a single solution, but a front of non-dominated solutions:

Definition 3 (Pareto Front) *A set X' is called a Pareto front of the set of solu-*

tions X if and only if:

$$\forall x \in X' : \nexists x_i \in X(x_i < x)$$

The Pareto front of Figure 2.9 contains six solutions: $A - F$. Each of these solutions does not dominate the other. An improvement on objective f_1 is matched by a worse value for f_2. Generally, it is up to the designer to decide which of the solutions provides the best trade-off.

Although the dominance relation determines if a solution is superior or not, it does not tell how far off a solution is. For this purpose, the Euclidean distance between two solutions can be used:

Definition 4 (Euclidean Distance) *The Euclidean distance d between solutions x_1 and x_2 for an n dimensional MOP is defined as:*

$$d(x_1, x_2) = \sqrt{\sum_{i=1}^{n} (f_i(x_1) - f_i(x_2))^2}$$

A downside of the direct usage of the Euclidean distance with a MOP is that the ranges of the different objectives may be quite different. Take for example the case where a two-dimensional MOP has one objective in the range $(0, 1)$ and one objective in the range $(0, 10^{10})$. In this case, it is quite likely that the distance between the values of the second objective will mainly determine the Euclidean distance metric. The distance between the values of the first objective, on the other hand, will hardly influence the Euclidean distance.

To suppress this discrepancy between the ranges of the different objectives, a normalized distance can be used:

Definition 5 (Normalized Euclidean Distance) *The normalized Euclidean distance $\overline{d}$ between solutions x_1 and x_2 is defined as:*

$$\overline{f_i}(x) = \frac{f_i(x) - f_i^{min}}{f_i^{max}(x) - f_i^{min}}$$

$$\overline{d}(x_1, x_2) = \sqrt{\sum_{i=1}^{n} (\overline{f_i}(x_1) - \overline{f_i}(x_2))^2}$$

The normalized distance translates all the objectives to a range between 0 and 1. For this purpose, the minimal ($f_i^{\min}$) and maximal value ($f_i^{\max}$) for the objective must be known.

There are many types of metrics that can be used to compare different Pareto fronts [76, 82] each with their own merits. In this thesis two metrics are used: generational distance and hypervolume. The generational distance provides an accuracy

relation (i.e., what is the distance between the front and the known reference front) and the hypervolume is used for convergence (i.e., how does the size of the front change over time). We have chosen for these metrics because of the simplicity with respect to other metrics. On top of that, embedded system design often is not trying to find the best solutions, but acceptable solutions as quickly as possible. Therefore, metrics like error ratio (percentage of points within the non-dominated set that are found) are not usable.

2.3.2 Generational Distance

The generational distance [77] is a metric to obtain the accuracy of a Pareto front. For this metric, two fronts are required: the front that needs to be judged and a reference front. Ideally, the reference front is the real Pareto front. Practically, however, it is infeasible to obtain the real Pareto front. Hence, in most cases the reference Pareto front is obtained by combining the results of all the experiments that are performed on the MOP.

In order to obtain the accuracy, the generational distance takes the average minimal distance between all points in the Pareto front and the reference front:

Definition 6 (Generational Distance) *The generational distance GDist represents how far the known Pareto front PF_{known} is from the reference front PF_{ref}:*

$$GDist(PF_{known}, PF_{ref}) = \frac{1}{|PF_{known}|} * \sqrt{\sum_{x \in PF_{known}} \min_{x_r \in PF_{ref}} \overline{d}(x, x_r)}$$

The lower the generational distance, the better the Pareto front is. Figure 2.10 shows an example of how to obtain the generational distance. Front B has three points. One point is exactly on the reference front (point $(0.9, 0)$) and the distances of the other points are $\sqrt{0.04}$ and $\sqrt{0.05}$. The sum of the squared distances becomes 0.09. After normalization, the generational distance is 0.1.

A downside of the generational distance is that it is not compatible with the Pareto dominance relation [82]. This can easily be seen in the example of Figure 2.10. Front A is clearly better than front B. One point is equal (0.9, 0), the other three points in front A dominate the points in front B. Still, the generational distance of front B to the reference front is smaller than the generational distance of front A to the reference front. If only the generational distance is taken into account, one would judge that front B is better. The poor generational distance of front A is caused by the additional Pareto point $(0.5, 0.3)$. This additional point gives a better spread for the Pareto front, but it is quite far from the closest point in the reference front. Hence, the average distance is significantly affected.

One can conclude that the generational distance may give an incomplete view of the quality of the Pareto front when the number of points in the Pareto front is low. The higher the number of non-dominated points, the lower the effect of the distance

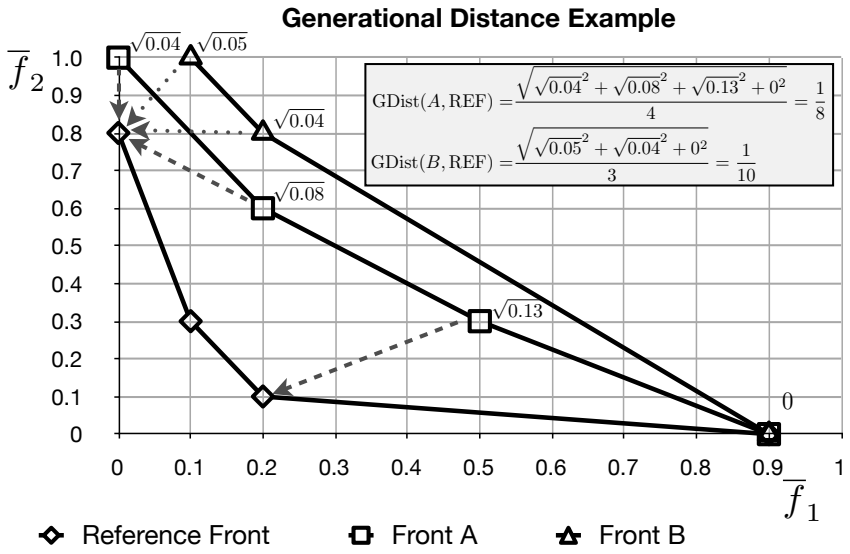


Figure 2.10: *An example of the generational distance. The arrows show the shortest distance to the reference front. For each front the generational distance calculation is shown (distance* 0 *is discarded from the equation).*

of an individual point. Generally, a MOP has sufficient solutions in order not to suffer from misleading conclusions. However, some of the DSE experiments used in this thesis have a small number of non-dominated designs in their Pareto fronts. In these cases it is better to use another metric for additional comparisons.

2.3.3 Hypervolume

This other metric that is used is the hypervolume [19]. The hypervolume does not directly provide the distance to a reference front, but it gives the size of the Pareto front. In this case, a larger Pareto front corresponds to a better Pareto front.

The hypervolume of a Pareto front is the volume of the area that is spanned by the front:

Definition 7 (Hypervolume) *The hypervolume of a Pareto front PF is defined as:*

$$Hypervol(PF) = \int_{(0,\ldots,0)}^{(1,\ldots,1)} \alpha_{PF}(x)dx$$

$$\alpha_{PF}(x) = \begin{cases} 1 & \exists x' \in PF : x' < x \\ 0 & else \end{cases}$$

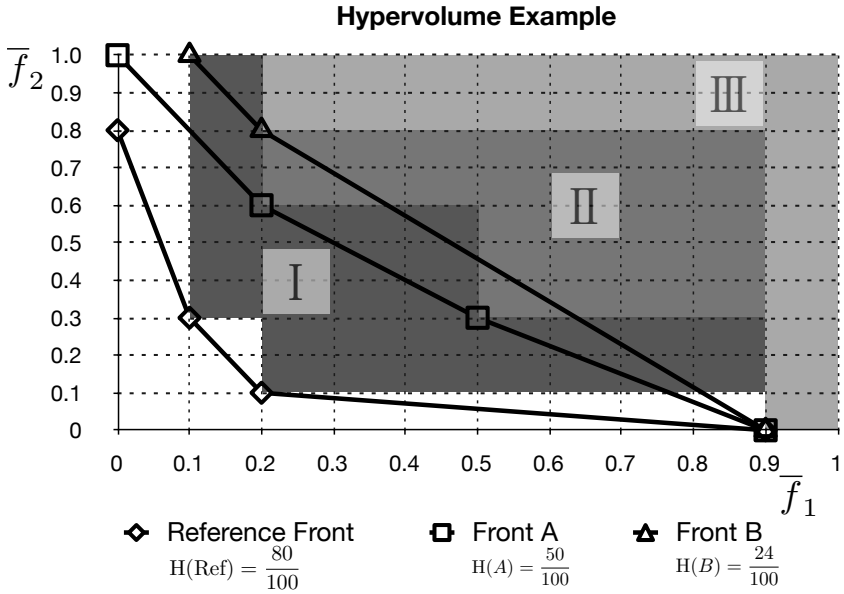


Figure 2.11: *An example of the hypervolume. For three Pareto fronts, the volume is shown with respect to reference point* $(1, 1)$*. The gray boxes show the blocks that are part of the three hypervolumes.*

Basically, the hypervolume is equal to the region spanned between the Pareto front and the reference point. As the objective values are normalized, the reference point is $(1, ..., 1)$ (no Pareto point should be larger than 1 after the normalization). The attainment function α describes if a point is within the spanned region by the Pareto front (i.e., there is a point in the Pareto front that dominates the point), or not.

The approach is illustrated in Figure 2.11. As this MOP has two dimensions, the hypervolume is equal to the area of the Pareto front. In this example, the reference point $(1, 1)$ is used. The hypervolume of front B is equal to area of region III: 24 boxes with an area of 0.01. Similarly, the hypervolume of front A is equal to the area of region II and III: 0.5.

An advantage of the hypervolume is that it is compatible with the Pareto dominance [82]. Front A, which was clearly better than front B, has now a higher hypervolume than front B. However, a downside of the hypervolume is that the hypervolume indicator is biased with respect to the choice of the reference point [80] and it is sensitive to the absence or presence of the extreme points.

2.4 Conclusion

This chapter focused on MOEAs and embedded system design. We have shown that a MOEA is a general technique to solve engineering problems with multiple objectives. Embedded system design is one of those engineering problems. To judge the quality of an individual embedded system design, the Sesame simulation framework is used. Sesame uses a strict separation of concerns. It has separate models for applications, architectures and the mapping between them. Therefore, it is a perfect simulation tool to explore the different possibilities of mapping embedded applications onto an MPSOC. Next to a evaluation tool, a search technique is introduced for exploring the design space. As a genetic algorithm is capable of exploring large and not well-understood design spaces, it is used for the exploration of the design space of MPSoC designs. This exploration results into a set of (sub-)optimal solutions that can be compared using metrics like the generational distance and the hypervolume.

Part II

Application Scenarios

3 Multi-Application Workload

This chapter is based on:

- P. van Stralen and A. D. Pimentel. 'A Trace-based Scenario Database for High-level Simulation of Multimedia MP-SoCs'. In: *Proc. of the Int. Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS '10)*. Samos, Greece, July 2010

In the previous chapter Sesame was introduced. Sesame targets efficient system-level DSE of embedded multimedia systems. To this end, it allows for rapid exploration of different MPSoC architecture designs, applications to architecture mappings and hardware / software partitioning. It used to be the case that Sesame only supported fixed single application workloads to be mapped on the underlying architecture model. For modern embedded systems this is not sufficient anymore. Users expect their embedded devices being highly multi-functional: a mobile phone has become a multimedia device that is not only used for calling, but it is also connected to the Internet. With the increased capabilities of current smart phones, they have almost the same possibilities as desktop computers. Another trend is to make consumer devices "smart". Smart digital cameras are able to directly share your photo to the Internet. This photo can be edited on the camera and is also tagged with GPS location where it is taken. Similarly, smart televisions also enhance the functionality of the television. It does not only show an incoming video stream from a decoder, but it is also able to show photos from a memory card, or to install additional applications for renting videos or playing games. This trend of smartness does not only increase the number of applications, but also the dynamism in the applications. Old analogue television with CRT used to exactly know the characteristics of the incoming video stream: the size of the frames, the frame rate, etc. Currently, however, these streams become more dynamic: High Definition (HD) television may have varying frame rates and frame sizes. Additionally, 3D television may double the amount of frames that need to be decoded.

To support the high functionality of embedded systems, Sesame is extended to support dynamic multi-application workloads. To achieve this, application workload

scenarios [56, 21] are used. Application scenarios are able to describe the dynamism of embedded applications and the interaction between the different applications on the embedded system. This chapter will discuss these application scenarios and the way they are stored. The first section will describe what application scenarios are. Next, a so-called scenario database will be introduced. After that, the detection of application scenarios will be discussed. Section 3.4 will describe some experiments to illustrate the efficiency of the scenario database and the diversity of application scenarios. Finally, a section on related work and a short conclusion wraps up this chapter.

3.1 Application Scenarios

Scenario-based design [9] is an approach to formalize the design of complex systems. Generally, it is based on the description of how people achieve tasks. In this definition people should be interpreted in a broad sense: it can, for example, also be interpreted as an embedded system application. One of the most well known examples of scenarios is a use-case scenarios. A *use-case scenario* can be seen as a storyboard. It describes the users of a system; their activities and the expectations the users have of the system. A user of a car radio, for example, can switch between stations. If the user switches between stations, the user expects to listen to the new station within a second.

In our case, we do not look at the system from the perspective of the user, but from the perspective of the application. An application has a certain kind of behavior and expects a certain *Quality-of-Service (QoS)* of the underlying architecture. Therefore, application scenarios [21] are used to model the dynamism of application(s). An *application scenario* gives a designer's perspective on the system. An example of such a designer's perspective can be given based on an MP3-application. A use-case scenario of a MP3 application can only describe the music that is played and the contents of the display. An application scenario for a MP3 application, on the other hand, knows which applications are running inside and what their behavior is. One of the benefits of application scenarios is that they allow for optimizing the efficiency of the system both statically and dynamically. When an application scenario is encountered at runtime, the system can be optimized to exploit the knowledge about the application behavior. For the MP3 application, a user can, for example, play sound in mono or stereo. When the user decides to switch from stereo to mono, the system can lower the voltage to increase battery lifetime. This can be done because static optimization can deduce that mono music requires less computational power than stereo music.

The concept of application scenarios is illustrated in Figure 3.1. An application scenario is based on two concepts: an inter- and an intra-application scenario. An *inter-application scenario* describes the interaction between multiple applications. As our application model is based on a KPN, the different applications are not able

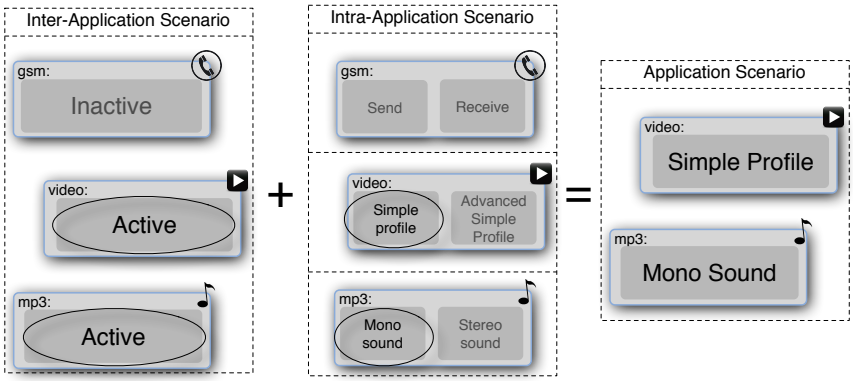


Figure 3.1: *An illustration of application scenarios.*

to influence each other (see Section 2.1.1). Therefore, the inter-application scenario describes which applications can run concurrently. Inter-application scenarios can be used to prevent the overdesign of a system. If some of the applications cannot run concurrently, then there is no need of reserving resources for the situation where these applications are running together.

After describing the valid combinations of active applications, the behavior of the individual applications must be described. To describe this behavior (or operation modes) of the individual applications, *intra-application scenarios* are used. Although there are multiple ways of describing the behavior, there is chosen to use event traces. Other techniques, like input parameters, are not applicable within the high level Sesame model. Due to the usage of events, Sesame is able to quickly evaluate mappings in seconds. Since scenarios should not significantly affect the evaluation time of Sesame, the description of scenario behavior should be limited to the usage of events. The event types described in Section 2.1.1, however, are not sufficient and are extended:

- Read
- Write
- Execute
- While
- Wend
- StartScenario
- EndScenario
- Wait
- Quit

The Read, Write, Execute and Quit events were already defined in Section 2.1.1. All the other events are introduced to support the modeling of application scenarios. At first, the While and Wend events are used to define loops within event traces. Secondly, Wait is used to model a timing delay, where the specified argument specifies the time until which the process must be halted. If the time has already passed, nothing is done. Otherwise, the process will halt execution until the specific point in time. Finally, the StartScenario and EndScenario events

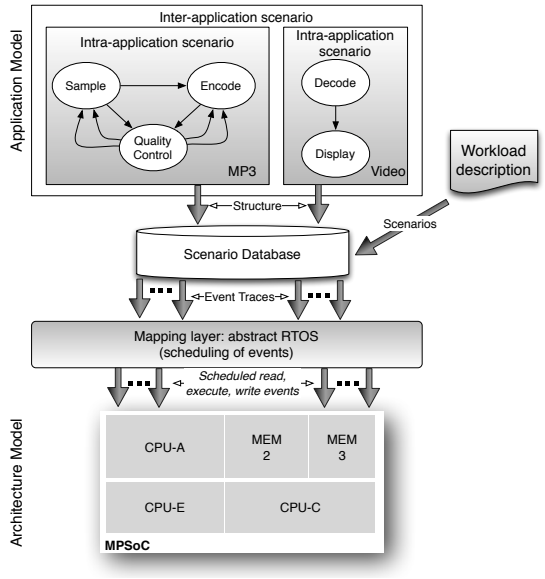


Figure 3.2: *The scenario-aware version of Sesame.*

specify explicit boundaries of an application scenario.

As was explained earlier, an application scenario is a combination of an inter-application scenario and an intra-application scenario. An example of an application scenario can be seen in Figure 3.1. On the left hand side, the selected inter-application scenario is shown. In this case, the Video and the MP3 applications are active and the GSM application is inactive. In the middle, the intra-application scenarios are shown. The Video application can, for example, decode video using a simple profile and an advanced simple profile. For the intra-application scenario, it is decided to decode video using a simple profile and to play mono music with the MP3 application. As the GSM is inactive, no operation mode needs to be selected for the GSM application. Hence, the application scenario is the sum of the inter- and intra-application scenarios: the Video application is decoding using a simple profile and the MP3 application is playing music in a mono sound.

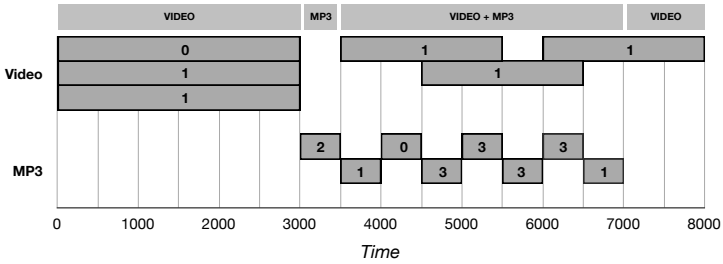
An important aspect to realize is the exponential number of potential application scenarios. As applications are independent and cannot influence each other, the number of potential application scenarios is exponential with respect to the number of applications.

3.2 Scenario Database

To facilitate the usage of application scenarios during DSE, a scenario database is used. A *scenario database* compactly stores the collection of application scenarios

```
<workload>
    <app name="Video" unit="500">
        <scenario id="0" start="0" deadline="6"/>
        <scenario id="1" start="0" deadline="6"/>
        <scenario id="1" start="0" deadline="6"/>
        <scenario id="1" start="7" deadline="11"/>
        <scenario id="1" start="9" deadline="13"/>
        <scenario id="1" start="12" deadline="16"/>
    </app>
    <app name="MP3" unit="500">
        <scenario id="2" start="6" deadline="7"/>
        <scenario id="1" start="7" deadline="8"/>
        <scenario id="0" start="8" deadline="9"/>
        <scenario id="3" start="9" deadline="10"/>
        <scenario id="3" start="10" deadline="11"/>
        <scenario id="3" start="11" deadline="12"/>
        <scenario id="3" start="12" deadline="13"/>
        <scenario id="1" start="13" deadline="14"/>
    </app>
</workload>
```

(a) *XML description of the workload.*



(b) *The frames over time specified by the workload.*

Figure 3.3: *A workload description for the MP3 and Video application.*

using a set of compressed event traces. From the scenario database, the workload (in the form of event traces) belonging to a specific application scenario can easily be generated in order to enable scenario-aware simulation within Sesame.

Scenario-aware Sesame, as illustrated in Figure 3.2, has two important differences with the original model of Sesame (Figure 2.1). First, there is a built-in support for multiple applications. In the example of Figure 3.2, two active applications are shown: an MP3 application and a video application. For each of the individual applications, a separate KPN is defined. Additionally, the inter-application scenario defines the active applications and the behavior of each active application is given using a separate intra-application scenario. For this purpose, the scenario database is used as an additional input for the mapping layer.

As a result of the introduction of the scenario database, the application does not directly emit event traces anymore. Instead, a workload description is used which, together with the scenario database, can be used to generate the event traces. As an example, Figure 3.3a shows a workload description for the multi-application work-

load with a MP3 and a video application. For each of the applications, a sequence of intra-application scenarios is given with a START and a DEADLINE. The START specifies when the application should be started, whereas the DEADLINE specifies when the task *should* be finished. As the application model purely describes functional behavior, this workload description only describes what the application expects. It depends on the mapping of the application onto the architecture if the deadlines are met. This separation of functional and non-functional behavior should be kept in mind when reasoning about application scenarios. It is important not to confuse application scenarios with system scenarios [21]. System scenarios also have information about the non-functional properties (properties of the architecture like execution time and energy), which is abundant in application scenarios. Application scenarios can only be transformed into system scenarios after having performed the static DSE.

Next to a START and a DEADLINE field, the workload description also has an ID field that specifies which intra-application scenarios are scheduled. Such a scheduled intra-application scenario within a workload is also called a *frame*. In this case, a frame corresponds to a single unit of work in the system. A unit of work can be a data packet, but also a part of a single frame in a video stream. This completely depends on which granularity the intra-application scenarios are defined. In contrast to the intra-application scenarios that are explicitly defined in the workload, the inter-application scenarios are implicitly defined. These inter-application scenarios can be deduced by constructing a time diagram of the individual frames. Figure 3.3b shows such a diagram. In this workload all the possible inter-application scenarios are used: Video application active ($[0, 3000)$ and $[7000, 8000)$), MP3 application active ($[3000, 3500)$) and both applications active ($[3500, 7000)$).

After introducing the workload description, the remaining part of this section will focus on the structure of the database. This structure is derived from the hierarchical nature of the application scenarios. There are three levels in the storage hierarchy: 1) the multi-application level, 2) the KPN level and 3) the Kahn-process level. In the multi-application level the inter-application scenarios are stored. Next, the KPN level contains the intra-application scenarios. Finally, the Kahn-process level keeps all the compressed event traces. As an example, Figure 3.4, provides a schematic view of a scenario database for the multi-application workload of the MP3 and video application that is shown in Figure 3.2.

3.2.1 Multi-application Level

A multi-application workload consists of multiple parallel applications. Each of these applications may have multiple intra-application scenarios. As each application is independent, any of the intra-application scenarios can be coupled in order to define an application scenario in which multiple applications are active simultaneously. Consequently, the only information that is stored is which applications can be active simultaneously.

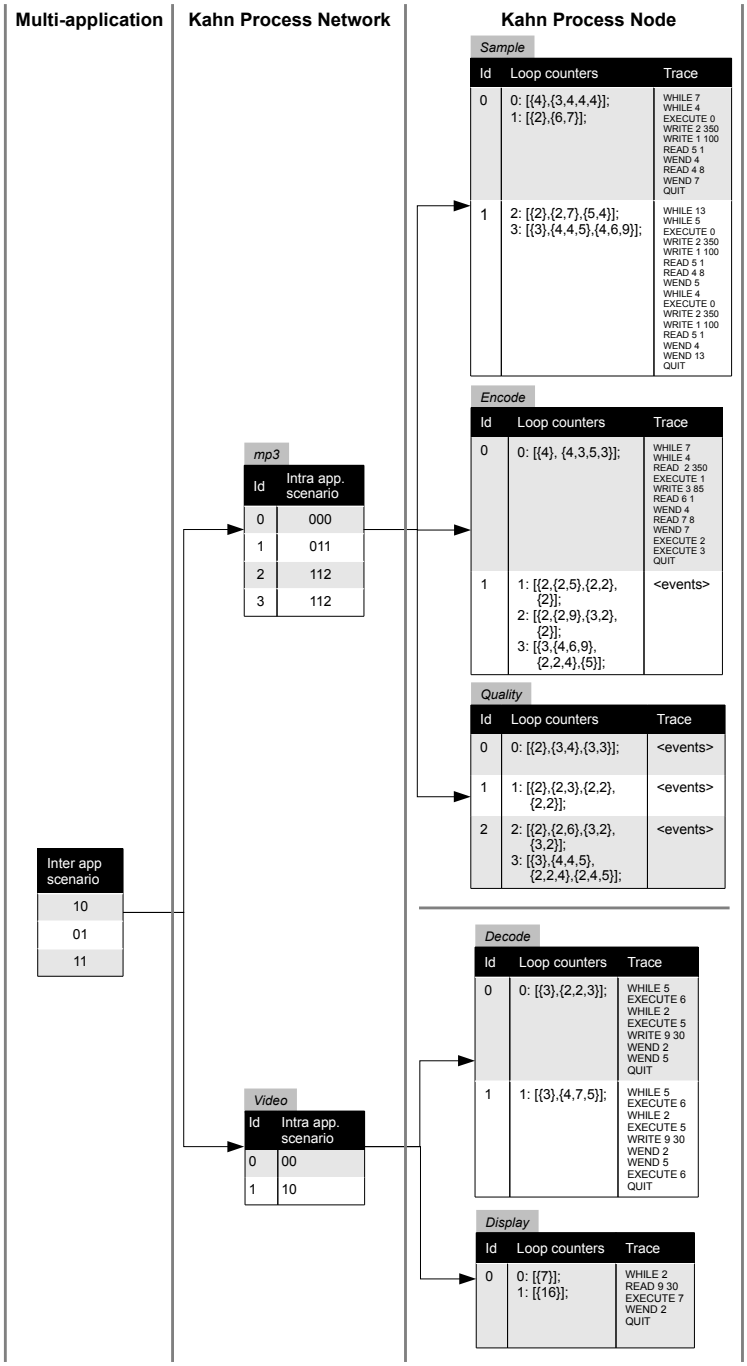


Figure 3.4: *The structure of the database for the MP3 / Video example.*

Within the database, the inter-application scenario is defined as a list of boolean values. A '0' stands for an inactive application, whereas a '1' is used for an active application. The list of inter-application scenarios directly affects the number of possible application scenarios. More precisely, the list of possible application scenarios is the Cartesian product of the intra-application scenarios of the active applications. If an application is inactive the list of intra-application scenarios only contains the empty scenario $\emptyset$. The database that is shown in Figure 3.4 has three inter-application scenarios: '1 0', '0 1' and '1 1'. Inter-application scenario '1 0' specifies that only the MP3 application is active, whereas in inter-application scenario '1 1' both the MP3 and video applications are active. Application scenarios corresponding to the inter-application scenario '1 0' are $\{< 0, \emptyset >, < 1, \emptyset >, < 2, \emptyset >, < 3, \emptyset >\}$. Similarly, the inter-application scenario '1 1' encodes the applications scenarios that are the Cartesian product of the two individual sets of intra-application scenarios: $\{0, 1, 2, 3\} \times \{0, 1\}$.

3.2.2 Kahn Process Network Level

These intra-application scenarios are described in more detail in the Kahn Process Network level. In contrast to the multi-application level, the Kahn Process Network level stores a data table per individual application. For efficiency reasons, this table does not store the event traces (this will be done in the Kahn Process Node level), but only the valid combinations of the traces of the individual Kahn processes in the KPN. Referring to our example in Figure 3.4, it is not valid to use trace 1 for all of the processes in the MP3 application ('1 1 1' is not present in the table of MP3). It is possible, however, that all of the processes of MP3 execute trace 0 as the entry '0 0 0' is contained in the table at the KPN level of the MP3 application.

3.2.3 Kahn Process Node Level

The individual event traces for all the Kahn processes are stored in the Kahn process node level. Sesame uses linear traces of sequential events as shown in Figure 3.5d. These linear, or *unfolded*, traces may contain a large number of events. As a consequence, the memory requirements of the traces may be quite large. As the scenario database will contain many of these event traces, it is undesirable to store unfolded event traces within the database. Therefore, three memory reductions are applied: 1) loop detection, 2) loop independent traces and 3) reuse of traces.

A first memory reduction is to fold the event traces by the detection of loops. An unfolded event trace is an event trace that does not contain WHILE and WEND events, whereas a *folded event trace* is allowed to contain WHILE and WEND events. Basically, folding replaces all consecutive sequences of repeating events by a single sequence that is enclosed by a WHILE and WEND event.

Secondly, using a loop independent trace can reduce the memory usage of a scenario database. Quite often an application has data dependent behavior where a

certain loop is executed for a varying number of times depending on the input. In this case, the unfolded event trace is different for each individual set of input data. If the loop counters were part of the WHILE and WEND events, the folded event traces would have been different as well. For a loop independent trace, however, the loop counters are stored separately from the trace. Hence, for each individual set of input data, the same event trace can be used. Only a different loop counter is required and, as a result, the reuse of the traces in different intra-application scenarios is improved.

Given these memory reductions, the table structure is as follows. Each of the folded traces is stored with a unique identifier. This is the identifier that is used within the intra-application scenario defined in the KPN layer. Next, each of the folded traces has an associated map with a sequence of loop counters. For each of the WHILE events an entry is present with the number of times the loop is executed. In case it is an outer loop this is a single value, whereas in the case of an inner loop a list of numbers is used for each time the loop is encountered during the unfolding of the event trace. The explicit storage of the iteration counts is required in order to keep the balance between the number of reads and writes of different processes. In case a process writes too few tokens on a channel, the reading process will deadlock on reading the specific channel. Obviously, this is undesirable behavior for a scenario-aware simulation.

Figure 3.5 contains an example of unfolding trace number 0 of the decode process of the video application from Figure 3.4. First, Figure 3.5a shows the folded trace. This is clearly visible by the presence of the WHILE and WEND events. The integer parameters of WHILE and WEND event refer to the length of the loop, whereas the arguments of the WRITE and EXECUTE are Sesame specific. To unfold the event trace a set of loop counters is required. In this example, the event trace is unfolded for intra-application scenario 0 for which the set of loop counters is $\{\{3\}, \{2, 2, 3\}\}$. When the loop counters are assigned to the WHILE events in the event trace, the trace in Figure 3.5b is obtained. As a first step during the unfolding the complete trace, the outer loop is removed by replicating the loop body three times (as defined by the loop counter). Next, the loop counters of inner loops must be split for every individual iteration of the loop, which results in the trace shown in Figure 3.5c. Since the trace still contains WHILE events, the unfolding is not yet finished. There are still three outer loops that can be unfolded. After unfolding these loops, no WHILE events are present anymore and the final trace is obtained (Figure 3.5d).

3.3 Scenario Detection

In the previous section, the structure of the scenario database was described. To fill this database, scenarios must be detected *within* and *between* applications. Therefore, we developed a method to identify application scenarios. A bottom-up approach first identifies intra-application scenarios, after which the inter-application scenarios are identified.

```
WHILE 5
    EXECUTE 6
    WHILE 2
        EXECUTE 5
        WRITE 9 30
    WEND 2
WEND 5
QUIT
```

(a) *Folded trace.*

```
WHILE 5 (3)
    EXECUTE 6
    WHILE 2 (2, 2, 3)
        EXECUTE 5
        WRITE 9 30
    WEND 2
WEND 5
QUIT
```

(b) *Added loop counters.*

```
EXECUTE 6
WHILE 2 (2)
    EXECUTE 5
    WRITE 9 30
WEND 2
EXECUTE 6
WHILE 2 (2)
    EXECUTE 5
    WRITE 9 30
WEND 2
EXECUTE 6
WHILE 2 (3)
    EXECUTE 5
    WRITE 9 30
WEND 2
QUIT
```

(c) *Unfolding step 1.*

```
EXECUTE 6
EXECUTE 5
WRITE 9 30
EXECUTE 5
WRITE 9 30
EXECUTE 6
EXECUTE 5
WRITE 9 30
EXECUTE 5
WRITE 9 30
EXECUTE 6
EXECUTE 5
WRITE 9 30
EXECUTE 5
WRITE 9 30
EXECUTE 5
WRITE 9 30
QUIT
```

(d) *Unfolded event trace.*

Figure 3.5: *An illustration showing the unfolding of an event trace.*

To enable scenario detection, we assume that the processes within a KPN are periodic. That is, each application process has a structure in which it continuously iterates a certain task. For multimedia applications (the main target of Sesame) this behavior is quite typical. Examples are the processing of a (macro)block of pixels or a complete frame in a video application. There may be some initialization and termination code in an application, but this code is not allowed to emit any trace events. Intra-application scenarios are based on a single iteration of an application. For a video application, for example, this can be one of the multiple ways in which

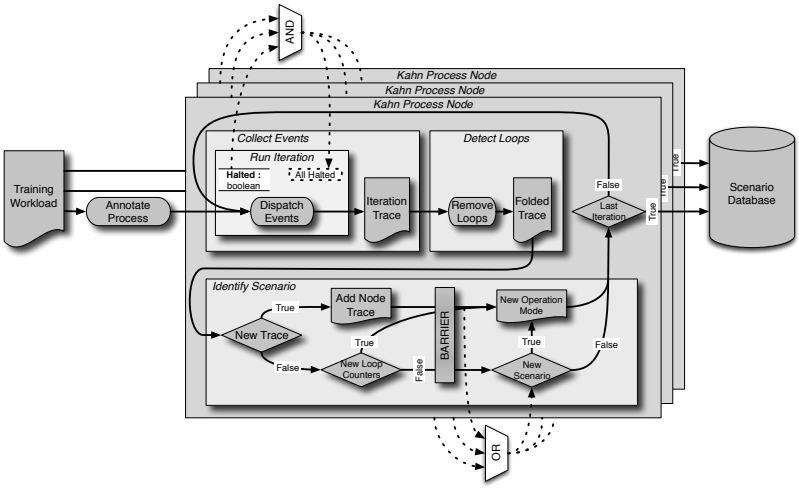


Figure 3.6: *Detecting scenarios in a KPN.*

a single frame can be processed. State that must be kept in-between iterations must be explicitly communicated [1].

As stated earlier, our bottom-up scenario detection approach splits the detection in two steps: the detection of the intra-application scenarios and the detection of the inter-application scenarios. For the discovery of intra-application scenarios there are multiple potential methods. In [21], three potential ways are identified: profiling, analytical and a hybrid combination of both. In case of *profiling*, the real runtime behavior is used to identify the scenarios. This approach is very sensitive to the supplied training input data. It can be the case that a scenario is missed because it is not encountered during the training. The analytical discovery method, on the other hand, is capable of identifying all the scenarios.

Due to our application model, we cannot use the analytical model. An analytical model requires the explicit modeling of input parameters in the application. A Sesame model only requires a runnable set of processes (if there are input parameters, they are modeled implicitly) and it only explicitly models the trace events (like READ, WRITE and EXECUTE). Therefore, the profiling method is chosen. The profiling method conforms to the high-level paradigm of the Sesame framework that is concerned with the early identification of promising embedded system designs. Hence, a small fraction of missed application behaviors will probably not significantly affect the outcome. An embedded system designer should be able to construct a training set of training data that leads to the detection of all the significant intra-application scenarios.

[1] A process send its state to itself at the end of each iteration. At the begin of the iteration it reads in its current state

Figure 3.6 illustrates the procedure to identify intra-application scenarios within a KPN. The procedure starts out with a KPN and a set of input data. This set of input data is used to collect the events of each scenario in the application. After loop detection, the (compressed) event traces are analyzed to determine whether or not they should be stored as a new scenario in the resulting scenario database. The following subsections will provide a detailed description of the individual steps during the scenario discovery.

3.3.1 Collecting Events

As a first step in the intra-application scenario detection, the event traces of a single iteration of an application are collected. Before even collecting a single iteration, the boundaries of the iteration must be determined. These boundaries can both be defined manually or automatically. In case of an automatic discovery of the boundaries, however, there is a potential ambiguity for picking the iteration boundaries. For a Sobel edge detector, for example, a single iteration can correspond to a complete image, but it can also correspond to a macroblock of an image. When these iteration boundaries are annotated manually, the designer can control the granularity of the scenarios. This is important as the granularity affects the scenario database and, thus, the outcome of the DSE.

Inserting the two special instructions StartScenario and EndScenario manually annotates an application. Each of the instructions needs to be added exactly once. To mark the start of an iteration a StartScenario is used. For scenario detection, StartScenario may be sufficient to automatically determine the iterations, but to enable the iteration-statistics in the architectural layer a marker for the end of the iteration is added (i.e., the EndScenario event). As the architectural layer may process multiple iterations simultaneously, the point in time where an iteration is finished does not need to coincide with the start of the next iteration. Based on these annotations, the iteration traces of all the individual processes of the application can be determined. For the annotated processes an iteration boundary takes place at a StartScenario or EndScenario event. Unannotated processes, however, are run until all of the processes are blocked on a read. In case both the annotated and unannotated processes are halted, the iteration of the application is finished.

Figure 3.7 shows the detailed procedure of the dispatching of events of a single iteration with automatic iteration detection. As input, the stream of events originating from the application layer of Sesame is used. During the dispatch, events are popped from the event trace and, potentially, added to the iteration trace. For this purpose, three supporting states are used:

1. halted: A boolean flag for each of the process to keep the state of a process. There are only two options: the process is halted or not.

2. All Halted: A signal that is used to determine if all processes are halted.

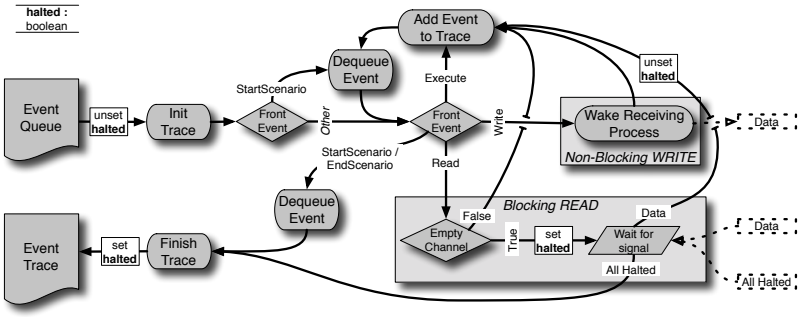


Figure 3.7: *A detailed view of the dispatching of events of a single process in order to obtain the event trace of a single iteration. In this diagram "halted" is a process state that is readable by other processes. Similarly, "Data" and "All Halted" are signals used for conditional waiting on other processes.*

Its value is equal to the logical AND-operator applied to the halted flags of the individual processes.

3. DATA: Per communication channel a single signal is used to notify a blocked reader that new data is available on the channel.

The dispatching of events depends on the type of event. Basically, there are five cases: 1) STARTSCENARIO, 2) ENDSCENARIO, 3) EXECUTE, 4) READ and 5) WRITE. If the first event of the iteration is a STARTSCENARIO event, the STARTSCENARIO event is ignored. Otherwise, it signals the start of the next iteration and, therefore, the process will be halted. Similarly, the first encounter of ENDSCENARIO also halts the process. Next, EXECUTE events are independent of other processes and can directly be added to the iteration trace. READ and WRITE processes, however, are dependent on the process they communicate with. WRITE operations can always be handled. As a side effect, the writing process will send a "Data" signal is sent to the, potentially halted, reading process on the channel. READ operations, on the other hand, are blocking and, therefore, the operation must wait when the channel is empty. In case data is present on the channel, the READ operation can be handled, otherwise there are two options: 1) the writing process has not yet written the data or 2) the current iteration is finished. That is why, on the unavailability of the data, a READ operation will halt the process until data is available (there is a "Data" signal) or the iteration has finished (the "All Halted" signal). Upon continuation, the READ event will only be moved to the iteration trace if the READ is really performed. Otherwise, all the processes are halted and the READ will be done in the next iteration.

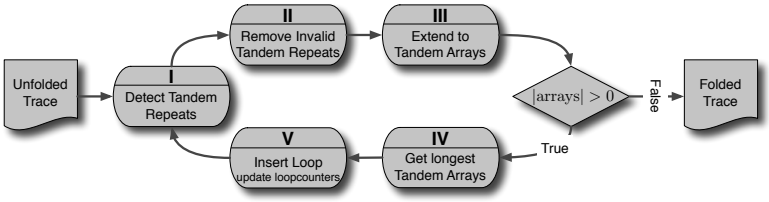


Figure 3.8: *The used procedure to detect loops within a trace.*

3.3.2 Detecting Loops

After collecting the event traces of a single iteration, loop detection is applied on each of the iteration traces of the individual processes. One of the merits of loop detection is that it keeps the event traces as small as possible, which is extremely usable for the large number of event traces that need to be stored in the scenario database. The steps of the loop detection procedure are shown in Figure 3.8. Initially, the iterative procedure starts with an unfolded trace (a trace without WHILE and WEND events) and, eventually, it produces a folded trace. An illustration of the folding procedure can be seen in Figure 3.5. This earlier example showed how a trace is unfolded, but, as folding is the reverse procedure of unfolding, Figure 3.5d-3.5a shows how a trace is folded. In the following paragraphs the five steps are elaborated:

I: Detecting Tandem Repeats

For loop detection, a simplistic linear search for a contiguous sequence of repeated events is infeasible. Therefore, the loop detection uses the technique of Stoye et al. [70] that exploits a suffix tree to detect branching tandem repeats. Within a suffix tree the possible suffices of an event trace are compactly stored. An example of such a suffix tree is shown in Figure 3.9. Each of the leaves of the suffix tree corresponds to a single suffix. By using internal nodes to merge identical parts of suffices, less storage is required for the suffix tree. Each suffix tree is based on an event trace S where i-th event can be addressed using $S[i]$. Subtraces from event i to j can also be extracted using $S[i..j]$. The leaf K in the suffix tree of Figure 3.9, for example, corresponds to the suffix $S[3..6]$ that starts at event $S[3]$ (WRITE 20).

The procedure of Stoye at al. [70] detects branching tandem repeats based on the structure of the suffix tree. A tandem repeat is special class of a tandem array. Tandem arrays are repetitions of a certain substring and they can be described as $S[i..j] = (\alpha w)^k$. In this case, α is a single event, whereas w is a subtrace of zero or more events. When $S[j+1]$ does not equal α, the tandem array is branching. Finally, k is the number of repetitions. For tandem repeats k equals two.

A first step in the detection of tandem repeats is the labeling the nodes in the suffix tree. Initially, each leaf gets the number of the first event of the suffix it

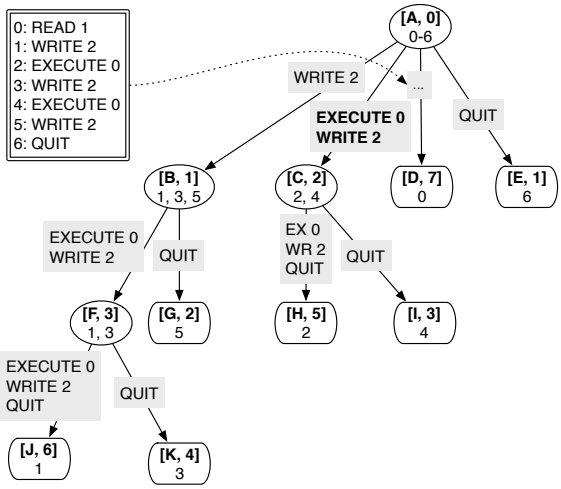


Figure 3.9: *An example for a suffix tree for the event trace shown in the top-left corner of the image. Each leaf (D, E, G-K) encodes one of the suffixes of the word. This tree can be used to detect the loop that starts at event* 2 *(the events* EXECUTE 0*;* WRITE 2*).*

encodes. Within Figure 3.9 the leaf K gets number 3 as that leaf encodes the suffix starting at index 3. Similarly, leaf J gets number 1. Next, the label of an internal node is the merge of the labels of its children. Node F gets label $1, 3$, the merge of the labels of leaf J and K. Similarly, the node B gets label $1, 3, 5$, the merge of the label $1, 3$ of node F and the label 5 of leaf G.

Secondly, the detection of tandem repeats is continued by identifying the branching tandem repeats. This is done by a pre-order tree traversal of the internal nodes. Each internal node is tagged with the length of the common prefix. Node F, for example, has a common prefix of length three and node C has a common prefix of length two. During the tree traversal, it is verified if the common prefix is a tandem that is repeated within the trace. Due to the structure of the suffix tree, the common substring is only part of a tandem repeat when there are two indices in the label that differ by the length of the common substring (i.e., the tandem). Therefore, node F does not contain a tandem repeat (the difference between label 1 and 3 is not equal to the length of the common prefix: three). The node C, on the other hand, has a tandem repeat: label 2 and 4 differ 2 events, which is exactly the length of the common prefix. As $S[2..5]$ is a tandem repeat, the event trace in Figure 3.9 has a loop starting at index 2 of two events that are repeated twice (EXECUTE 0; WRITE 2).

As loop detection is done iteratively, the compared events do not only involve READ, WRITE and EXECUTE events, but also WHILE and WEND events may be encountered. These WHILE and WEND are equal if and only if the length of the loop

```
WHILE 1 (3)
=============
    READ 3 30
WEND 1
WHILE 1 (2)
    EXECUTE 6
-------------
    READ 3 30
WEND 1
WHILE 1 (4)
    EXECUTE 6
=============
    EXECUTE 5
WEND 1
QUIT
```

(a) *The event trace with the invalid tandem repeat between the '=' line.*

```
WHILE 1 (3)
    =================
    WHILE 3 (2)
            READ 3 30
        WEND 1
        WHILE 1 (...)
            EXECUTE 6
    WEND 3
    =================
    EXECUTE 5
WEND 1
QUIT
```

(b) *The inconsistent trace after folding of the invalid tandem repeat.*

Figure 3.10: *The consequence of folding of an invalid tandem repeat.*

is the same. As loop counters are stored separately, the number of times the loop is iterated is irrelevant.

II: Removing Invalid Tandem Repeats

Although WHILE and WEND events do not significantly complicate the comparison of events, their introduction may lead to the detection of invalid tandem repeats. Intrinsically, the loops in the event trace are a subgroup of events within the trace. Hence, a valid tandem repeat must contain the complete subgroup events. Therefore, if a WHILE event is contained in the tandem repeat, the corresponding WEND event must also be part of the tandem repeat. Similarly, a WEND event in the tandem repeat must be matched by the corresponding WHILE event.

As illustrated in Figure 3.10, a violation of the complete containment of the loop body within a tandem repeat may lead to an invalid folded trace. In Figure 3.10a, an invalid tandem repeat is shown that does not completely contains the loop bodies. If this tandem repeat would be used to fold the trace with this newly detected loop, the trace in Figure 3.10b would be obtained. This trace is clearly inconsistent with the unfolded one.

To efficiently verify if the complete loop body is contained within the tandem repeat, a so-called event depth is introduced that describes in how many loops the event is contained. An event that is not in a loop has depth of zero. Similarly, an

event within the first inner loop (like the EXECUTE 5 event in Figure 3.5a) has depth of two. If the first and last events of a tandem repeat both have a depth of zero the tandem repeat is valid. If the depth of the first and last events is unequal, on the other hand, the tandem repeat is invalid. In the case that both of the depths are equal and nonzero the tandem repeat must be analyzed in more detail. The repeat is scanned for WHILE and WEND events and, in case one of those events is encountered, it is checked if the corresponding WEND or WHILE event is also present in the tandem repeat. This analysis is the sole reason for the use of WEND events for defining loop bodies. Without the use of WEND events the whole trace must be analyzed in order to find out if all the loop bodies within the tandem repeat are complete.

III: Extending to Tandem Arrays

For using the branching tandem repeats to remove the loops, the tandem repeats are extended to tandem arrays. Since the tandem repeat is branching (i.e., a different event is found in the forward direction), the only required step to achieve this is to potentially extend the tandem in the backward direction using the created suffix tree. As an example, take the suffix tree in Figure 3.9. It has a branching tandem repeat that starts at event 2 that is shown in node C. To extend this branching tandem repeat to a tandem array, there is iteratively checked if the repeating pattern is also present in front of the current tandem array. In case node C can be extended with an additional tandem, node C should also contain event 0 (which is 2 minus the length of the tandem). As this is not the case, the branching tandem repeat in node C cannot be extended and the resulting tandem array has a size of two.

IV: Getting Longest Tandem Array

At this point in the procedure, we have a list of tandem arrays that can potentially be removed. However, as these tandem arrays may overlap, not all of the tandem arrays can be used for loop compression. Thus, the tandem arrays must be handled one by one. Different approaches can be used to determine the ordering of tandem removal. Potentially, the ordering may even affect the size of the folded trace. In our case, we have chosen for a greedy approach where the largest tandem array is removed first. The score of a tandem array is based on the number of occurrences, which do not necessarily need to be consecutive. In the example of Figure 3.5d, two tandem arrays are present: 1) "EXECUTE 5; WRITE 9 30" and 2) "EXECUTE 6; EXECUTE 5; WRITE 9 30; EXECUTE 5; WRITE 9 30". The first tandem array starts at the trace index $1, 6$ (with $k = 2$) and 11 (with $k = 3$). Consequently, the total number of occurrences is $2 * 2 + 3 = 7$. Next, the second tandem array is only used once (starting at event 0) and has three occurrences.

Since only one tandem array is selected at the time, the loop detection procedure, which is shown in Figure 3.8, involves multiple iterations until all tandems are

removed. For our example in Figure 3.5, the second iteration (Figure 3.5c) is also the last iteration. Additionally, this example shows how the ordering can affect the size of the folded trace. In case the second tandem array is removed in the first iteration, the inner loops would not be detected and the folded trace would have contained ten events instead of eight events.

V: Inserting Loop

After identifying the tandem array that is going to be compressed, a loop must be created within the folded event trace. First, the events of the tandem array are removed and replaced by a single iteration of the tandem. This tandem is enclosed by a WHILE and a WEND event that is parameterized using the length of the tandem. Secondly, the loop counters are generated. The loop counter of the newly generated loop will be a list containing a single number: the number of occurrences of the tandem. As loop detection may involve multiple iterations, the generated loop body may contain inner loops. In this case, the loop counters of the inner loops need to be updated by concatenating all the individual loop counters.

As an example, in the trace of Figure 3.5 two loops are inserted. In the first iteration (Figure 3.5d), the tandem array "EXECUTE 5; WRITE 9 30" at indices $1, 6$ and 11 is removed. Since there are no inner loops, the loop generation is trivial: the tandem arrays are replaced by "WHILE 2; EXECUTE 5; WRITE 9 30; WEND 2". For the tandem arrays at index 1 and 6 the loop counter is $\{2\}$, whereas the tandem array at index 11 is replaced by a loop with a loop counter of $\{3\}$. During the second iteration (Figure 3.5c), the tandem "EXECUTE 6; WHILE 2; EXECUTE 5; WRITE 9 30; WEND 2" is compressed. Except for the presence of inner loops, the approach is in this case exactly the same as in the first iteration. To generate the loop counter of the inner loop, however, the individual loop counters $\{2\}, \{2\}$ and $\{3\}$ are concatenated to $\{2, 2, 3\}$. In the third iteration (Figure 3.5b) there are no tandem repeats in the folded trace anymore and, therefore, the compression of the folded trace has been completed.

3.3.3 Identifying Scenarios

The scenario identification procedure, which is illustrated in Figure 3.6, concludes the identification of the intra-application scenarios using the folded event traces that were obtained in the previous steps. This identification procedure has two parts: an individual part and a collective part. In the individual part it is analyzed if the behavior of the individual process is new, whereas the collective part determines whether or not a new operation mode of the application has been found. In case any of the individual processes shows some new behavior, a new operation mode is present.

At first, the individual part of the identification procedure fills the Kahn process node level (see Section 3.2.3) of the scenario database that contains the individual

event traces of the Kahn processes. During this first step of the identification, the event trace will only be added to the database if it shows some new behavior. New behavior is found if the folded event trace and / or the set of loop counters belonging the event trace are not yet present in the scenario database.

Secondly, the collective identification will determine whether or not a new intra-application scenario is present. This is the case if and only if at least one of the individual processes has observed new behavior (i.e., a new event trace and / or a new set of loop counters). For this reason, the collective identification can only start if all the processes have determined whether or not there is new local behavior. These observations are combined (as the OR port in Figure 3.6 illustrates) and in the case of a new intra-application scenario (i.e., operation mode), all the process nodes must store the loop counters of the current event traces. This loop counter will be addressed using the identifier of the newly created intra-application scenario.

3.3.4 Inter-application Scenarios

After automatically identifying the intra-application scenarios of the individual applications, only a partial scenario database is obtained. With the intra-application scenarios the behavior *within* the applications is determined and not the behavior *between* them. The identification of the inter-application scenarios is done separately from the intra-application scenarios and, therefore, the intra-application scenarios can be reused for different projects. In contrast to the intra-application scenarios, however, the generation of the inter-application scenarios is a manual procedure.

Basically, the embedded system designer selects the set of applications from the application library that will be used for the current MPSoC design. As the intra-application scenarios are precalculated, the designer only needs to specify which inter-application scenarios can actually occur. Potentially, the inter-application scenario could also be detected from a training workload, but as the designer creates the training workload the designer would in this case also manually define the applications that can run concurrently. Therefore, manual definition of inter-application scenarios is done.

3.4 Experiments

An important aspect of the scenario database is the compact storage of the application scenarios. In this section, we present several experiments to evaluate the effectiveness of the scenario database. This effectiveness is evaluated by comparing the storage size of the plain traces and the storage size of the scenario database. The second experiment will illustrate the dynamism within a single application and its effect on the DSE.

For the first experiment four applications are used: 1) a Motion JPEG (MJPEG) encoder, 2) a Simple Profile MPEG4 decoder (taken from [73]), 3) an MP3 decoder

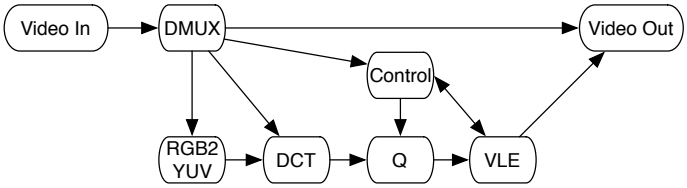


Figure 3.11: *The MJPEG encoder with quality control that is used for our experiments*

and 4) Sobel edge detector. In the second experiment only the MJPEG encoder is used. The following subsections will describe both experiments.

3.4.1 Storage Efficiency

In this experiment, we will analyze the storage efficiency of the scenario database. As the system designer manually defines the inter-application scenarios, we will focus on the intra-application scenarios that are detected and stored automatically. Before any intra-application scenario can be detected, the grain size of the intra-application scenarios must be defined. For the MJPEG encoder, the MPEG4 decoder and the Sobel edge detector the grain size is defined at the granularity of a complete image. This means, for example, that for the MJPEG encoder a single intra-application scenario corresponds to the compression of a single raw image. For the MP3 application, however, two different granularities are used for the intra-application scenarios: the coarse grained scenarios are detected on a second of played music, whereas the fine-grained detection considers a single sound sample as an iteration.

Figure 3.12a shows the number of detected intra-application scenarios per application. The MJPEG encoder, as shown in Figure 3.11, is equipped with quality control and it has 11 detected scenarios for the 11 images that are used in the training workload. Although for each of the images a different intra-application scenario is obtained, four different behaviors can be observed when the varying behavior of the highly dynamic VLE (variable length encoding) is discarded. One of the dynamic behaviors is the situation that the image is directly encoded. In the other cases the quality control updates some of the parameters (e.g., Huffman tables and / or quantization tables) that are used for the image encoding. The Sobel application, on the other hand, has static behavior where, irrespective of the input, the same intra-application scenario occurs.

When changing the granularity of scenario detection, the effect on the number of scenarios is unpredictable. In our experiment, a fine-grained detection (a single iteration corresponds to a single sound sample) of the scenarios in the MP3 application results in three detected intra-application scenarios, whereas a coarse-grained detection (a single iteration corresponds to a second of music which are multiple

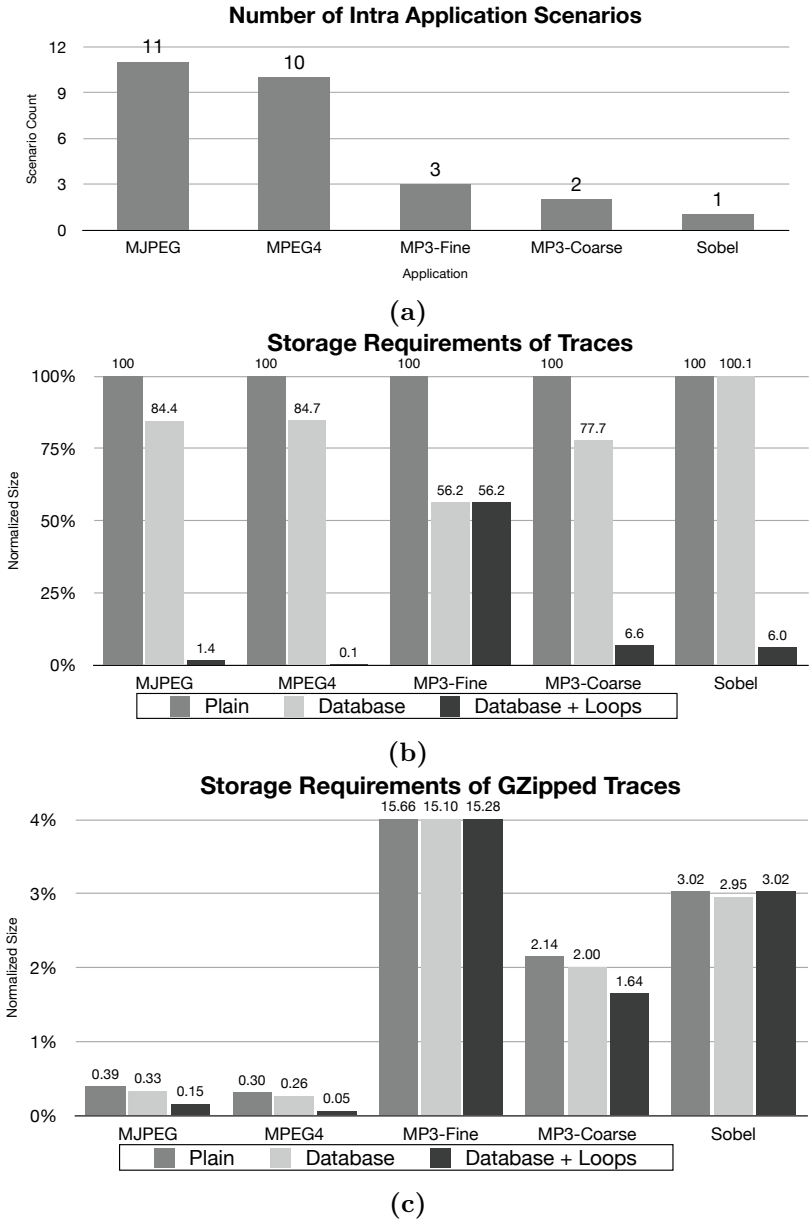


Figure 3.12: *A series of experiments to show the efficiency of the scenario database with four different applications. For one of the applications the scenarios are detected with a fine grained and a coarse grained granularity.*

sound samples) results into two scenarios. The two scenarios in the fine-grained detection can be clearly be identified as the building blocks of the three scenarios that are detected with a coarse-grained detection. Nonetheless, it is not necessarily the case that a finer granularity results in lower number of detected scenarios. As an example, more fine-grained detection granularity may reveal different phases within an iteration of a program.

The Scenario Database Storage Gain

To analyze the efficiency of the scenario database, the intra-application scenarios are stored in three different ways: 1) Plain, 2) Database and 3) Database + Loops. The plain technique stores the set of event traces (one event trace per process) for each individual intra-application scenario separately. Our proposed scenario database tries to significantly reduce the amount of storage by storing the intra-application scenarios into a single database. In this way, storing duplicate event traces for the same process is circumvented. Finally, additional loop detection not only compresses the event traces, but it also separates some of the data dependent behavior from the event traces. The compression, as applied by the scenario database and the loop detection, is lossless. Hence, the event traces that are used during scenario-aware simulation are exactly the same as the event traces in the original application workload.

The graph in Figure 3.12b shows the difference in storage requirements for the three techniques. For illustration purposes, we divide the absolute storage size with the storage size of the plain traces to normalize the storage size. Consequently, the lower the normalized size is, the better. As our experiment shows, a scenario database without loop detection can already significantly reduces the storage requirements: for the MJPEG encoder and the MPEG4 decoder the storage is reduced by 16%. In case of the MJPEG encoder, this means that 21.8MB is required instead of 25.8MB. In contrast to the dynamic image compression workload of the MJPEG and MPEG4 applications, the behavior of the Sobel application is static. As a result, there is only a single intra-application scenario and the scenario database does not provide any storage benefits. Due to the overhead of the scenario database, the storage requirements are even 0.1% larger. This difference between the static Sobel application and the dynamic MJPEG application clearly illustrates the storage benefit of the scenario database. The storage reductions of the scenario database are achieved due to the fact that the scenario database avoids the separate storing of the traces from all the individual frames. If the traces of multiple frames are equal within a process, then only a single trace is stored. As a result, the storage requirements decrease in case there is recurring behavior in the different intra-application scenarios. In contrast to the other applications, the Sobel application has a single intra-application scenario and, therefore, it lacks recurring behavior between different intra-application scenarios. Similarly, this explains the difference between the fine grained MP3 and the coarse grained MP3. The fine grained MP3 has more intra-application scenarios and,

therefore, has more recurring behavior (44% reduction versus 23%).

Another compression technique is the loop detection. The gain of the loop detection is twofold. Not only the traces of the individual frames become smaller, but the separate loop counters also expose more recurrent behavior for the event traces of the individual processes. When loop detection is enabled, the storage requirements are reduced by 98.6% for M-JPEG (only 358KB instead of 25.8MB) and 99.9% for MPEG4. Both the static Sobel application and MP3-Coarse is able to gain 94% with respect to storage requirements. Although the Sobel application has a static behavior, it contains a significant number of loops that can be compressed. The more fine grained MP3-Fine does not benefit from the storage gain loop detection. Where MP3-Coarse has a loop that repeatedly decodes a sample, the unit of work of MP3-Fine is a single sample without any loops

The Effect of GZip on our Storage Gains

One can conclude that our proposed scenario database compactly stores the intra-application scenarios. However, there are other common techniques to compress generic data such as the GZip algorithm. To compare the generic compression of GZip with a domain specific compression like our scenario database, we have investigated the effect of GZip on the gain in storage of the event traces. For this purpose, our previous experiment is extended with a compression of the stored intra-application scenarios into GZipped tar archive. The results are shown in Figure 3.12c. Generally, the GZipped traces are slightly smaller than the traces compressed with our own compression technique. The only exception is the MPEG4 application. For this application, the size of the GZipped plain traces is reduced to 0.3%, whereas for the (uncompressed) scenario database with loop detection is 0.1%. For the applications with the most dynamic behavior, the GZipped database is still smaller than the GZipped plain traces. Similarly, the loop detection still has a storage benefit. This is purely because of the recurrent behavior over the different intra-application scenarios. The scenario database utilizes this by storing less traces, and, due to the separation of loop counters, the loop detection results in even more recurrent behavior. These results indicate that the scenario database is an efficient way of storing a sequence of related traces, as multimedia applications typically contain many loops.

The compression of the individual traces due to loop detection, on the other hand, does not provide a storage benefit when the traces are compressed using GZip. This is illustrated by the Sobel application. In the previous experiment (Figure 3.12b), we observed that the Sobel application only benefits from loop detection and not from the scenario database. Hence, no recurrent behavior is present. As a consequence, the same number of event traces are compressed for the scenario database with / without loop detection and the plain traces. Since GZip also exploits repetitions during the compression, the resulting storage sizes are similar for all the techniques.

To conclude, a higher amount of dynamism in the workload results in a higher compression of our scenario database. The best example is the MPEG4 application

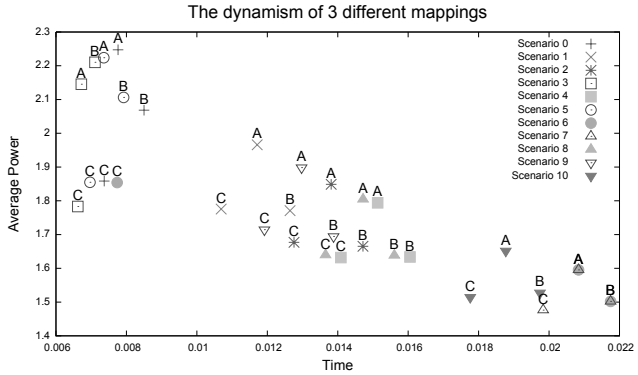


Figure 3.13: *An illustration of the dynamism of different scenarios. For three different mappings the fitness of each individual scenario is shown.*

from which the unzipped scenario database with loop detection is even smaller than the GZipped traces. Additionally, the benefit of our unzipped scenario database is that the traces can directly be read from disk. GZipped traces, however, need to be unzipped before they can be used. This involves an additional latency and requires additional storage during decompression. The additional latency of GZipping is especially a problem given the fact that our system-level simulations need to be extremely fast, where each simulation typically takes less than a second. Loop detection also involves overhead, but this detection is just a one-time effort during the creation of the scenario database. While reading events from the scenario database, no additional storage or computing time is required. Therefore, the one-time overhead of the loop detection is fully compensated by the gain in storage reduction.

3.4.2 Dynamism

The previous experiment addressed the dynamism of the workload of the intra-application scenarios of the individual applications. Up to now this dynamism was only discussed, but not shown. For this purpose, this experiment shows the dynamism of the MJPEG application.

One of the ways to observe the dynamism inside an application is to analyze the non-functional requirements of its different intra-application scenarios. In order to obtain the non-functional properties, an architecture is required and a mapping from the application onto the architecture. In principal, any architecture can be chosen to illustrate the dynamism of the MJPEG application. For this experiment, we have chosen to use a bus-based architecture with four processors and a single shared memory (somewhat similar to the architecture in Figure 2.2, but with four instead of two processors).

Non-Functional Difference of Intra-Application Scenarios

First, we have randomly picked three mappings of the MJPEG application on our bus-based architecture. For each of these mappings, we used Sesame to determine the execution time and power usage for each individual intra-application scenario. The resulting fitness values of these mappings are shown in Figure 3.13, where the horizontal and vertical axis refer to time and power consumption, respectively. These values are only used to compare different mappings and intra-application scenarios. Therefore, they do not have a unit. Within this graph, the letters A-C are the different mappings, whereas 0-10 are the different intra-application scenarios of the MJPEG application.

Irrespective of the mapping, scenario 7 is the intra-application scenario that consumes the least amount of power. In contrast to the low power solution, the scenario with the highest power consumption depends on the mapping. For mapping A and C, scenario 0 has the highest power consumption. In case of mapping B, however, scenario 3 has the highest power consumption. To explain this behavior, both scenarios and the mappings must be analyzed. At first, scenario 3 describes the decoding a frame that has a much higher compression ratio than scenario 0. As a consequence, scenario 3 requires less communication than scenario 0. Secondly, for mapping B the shared bus is fully utilized. In case of mapping A and C, on the other hand, there is still some capacity left on the shared bus. As a result, the reduction in communication between scenario 0 and 3 has a more significant effect on the execution time of mapping B than it has on mapping A and C. Although the consumed energy for scenario 3 is lower than the consumed energy for scenario 0 for all of the three mappings, the larger difference in execution time results in higher power consumption for mapping B.

Figure 3.13 also shows other interesting behavior with respect to scenario 6. Take, for example, scenario 6 that has for mapping A and mapping B the same fitness as scenario 7. For mapping C, however, the execution time of scenario 6 is much lower than scenario 7. Without going into details, scenario 6 would lead to the conclusion that mapping C consumes more power than mapping A. This conclusion contrasts with the conclusion that can be drawn from the comparing the power consumption using the other scenarios (i.e., the power consumption of mapping C is lower than mapping A). In a sense, scenario 6 gives a deceiving view of the relation between the different mappings. As we will further elaborate in the next experiment, this makes the identification of the Pareto front uncertain.

The Relation between Mappings

In the previous experiment, we discussed the unexpected non-functional behavior of the different intra-application scenarios. The problem of the unexpected behavior is that it complicates the comparison of different mappings. As discussed earlier, we use the Pareto dominance relation in order to compare different mappings. The

Scenario	A ... B	A ... C	B ... C	Front
0	$\|\|$	$>$	$>$	C
1	$\|\|$	$>$	$\|\|$	B, C
2	$\|\|$	$>$	$\|\|$	B, C
3	$\leq$	$>$	$>$	C
4	$\|\|$	$>$	$>$	C
5	$\|\|$	$>$	$>$	C
6	$\|\|$	$\|\|$	$\|\|$	A, B, C
7	$\|\|$	$>$	$>$	C
8	$\|\|$	$>$	$\|\|$	B, C
9	$\|\|$	$>$	$\|\|$	B, C
10	$\|\|$	$>$	$>$	C

Table 3.1: *The Pareto dominance relations comparing the mappings from Figure 3.13 for each individual scenario. The symbol || stands for incomparable fitness values.*

potential dominance relations between the mappings of the experiment illustrated in Figure 3.13 are listed in Table 3.1. In the three columns in the middle of the list the unique mapping comparisons are shown: mapping A versus B, mapping A versus C and mapping B versus C. Next, for each individual intra-application scenario, the fitness value for the different mappings are compared. In this way, three different types of relations are obtained: 1) A mapping is equal to or fully dominates the other mapping ($\leq$), 2) A mapping is dominated by another mapping ($>$) and 3) the mappings are not comparable using the Pareto dominance relation ($||$). Finally, the last column shows the Pareto front based on the fitness values of the specific intra-application scenario.

As a first observation, one can see that none of the relations the scenarios fully agree on the type of the relation. For the first two relations (where mapping A is compared with mapping B and C) only one scenario differs with respect to the relation type. In case of the comparison between mapping A and mapping B, the fitness values for most of the scenarios determine that mapping A is incomparable with mapping B. Only the fitness values of scenario 3 leads to a different conclusion: mapping A dominates mapping B. Similarly, mapping C dominates mapping A for most of the intra-application scenarios. Only the fitness values of scenario 6 are incomparable for mappings A and C.

The problem arises with the relation between mapping B and mapping C. Judging on 6 out of the 11 scenarios, mapping B is better than mapping C. The other 5 scenarios, however, lead to the conclusion that mapping B is incomparable with mapping C. These kinds of uncertainties complicate the scenario-aware DSE. The DSE ends up with a Pareto front, but not all the intra-application scenarios agree on what the Pareto front should be. In the example of Table 3.1, three different Pareto fronts are

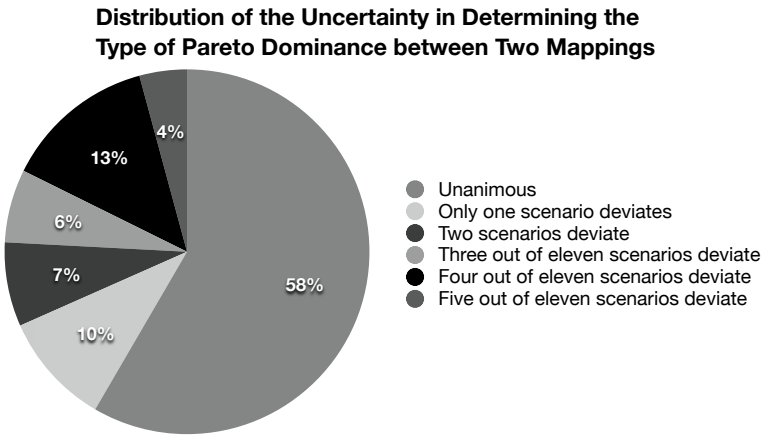


Figure 3.14: *The uncertainty when determining the relation type between different mappings based on all the intra-application scenarios of MJPEG.*

observed, from which the front with only mapping C is the most common.

Based on the most common Pareto front with only mapping C, one could conclude that the set $\{0, 3, 4, 5, 7, 10\}$ of intra-application scenarios is representative for the MJPEG application. This representativeness, however, is completely dependent on which mappings are evaluated. In case only mappings A and B would have been taken into account, intra-application scenario 3 would have been interpreted as an unrepresentative scenario. If this scenario 3 is excluded for the comparison between mapping B and mapping C, there is no majority anymore for one of the Pareto dominance relation types.

The experiment of Table 3.1 only uses three random mappings. As a result, it does not quantify the amount of uncertainty and how often this occurs. To quantify the uncertainty we randomly generated 5000 mappings. These mappings are evaluated for all the 11 different scenarios of the MJPEG application. The 5000 mappings correspond to approximately 2.5 million unique mapping relations. For each of these relations, the relation type for each of the individual scenarios is compared (e.g., dominating, dominated or incomparable). In this way, the relation type of the majority of the scenarios can be obtained and the number of scenarios that deviate from the majority. The results of this experiment are shown in Figure 3.14.

Uncertainty about the relation between the mappings is correlated with the average Euclidean distance between the fitness values of the different mappings. Therefore, Table 3.2 shows the average distance between the mappings given a specific uncertainty. At first, in 58 percent of the mapping relations the scenarios unanimously determine the relation type. Not surprisingly, the average Euclidean distance for these mapping relations is significantly larger than the other degrees of uncertainty (Table 3.2). The closer the fitness values of the mappings are, the more uncertainty. In not less than 13 percent of the mapping relations the fitness values

Uncertainty	Average Euclidean Distance
Unanimous	0.063398
Only one scenario deviates	0.000233
Two scenarios deviate	0.000098
Three out of eleven scenarios deviate	0.000086
Four out of eleven scenarios deviate	0.001616
Five out of eleven scenarios deviate	0.000028

Table 3.2: *The average Euclidean distance between the mappings that are compared grouped with respect to the number of intra-application scenarios that predict a different type of dominance relation between the compared mappings than the majority.*

of four out of the eleven scenarios deviate for the majority. Even worse, in 4 percent of the mapping relations there is hardly a majority as five out of the eleven scenarios predict a different relation than the other six scenarios.

From this experiment, we can conclude that the mapping comparison is not trivial when multiple application scenarios are taken into account during the DSE. This uncertainty only becomes larger when multiple applications are taken into account. When a subset of scenarios is used to evaluate the mappings of a dynamic multi-application workload it is absolutely not sure if the decisions that are taken are the decisions of the majority of all the scenarios. Therefore, it is absolute crucial that such a subset is representative. To complicate matters even more, the experiment of Figure 3.1 shows that the representativeness of a scenarios also can change with respect to different mappings. To address this the next chapter will address the automatic search for a representative subset of scenarios.

3.5 Related Work

The main contribution of this chapter is the description of the application scenarios and the approach that is used to detect these application scenarios. A few approaches for application scenario detection have been proposed before, both statically [23] and based on profiling of applications [22], like the technique that is discussed in this chapter. Both of these approaches use system scenarios. The main difference of system scenarios and application scenarios is that the system scenarios also take the non-functional requirements into account. Therefore, our application scenarios can only be seen as a system scenario in case they are combined with a single mapping. In [22], first the program variables are determined after which the execution time of different parameter instances is used to identify groups of parameter settings that can be seen as a single scenario. This profiling technique is not applicable in our case, as the non-functional properties like execution time are not known yet. On top of that, the Sesame model does not have a notion of variables.

Most of the approaches that model a multi-application workload for embedded

systems only take the inter-application scenarios into account. This can either by using use-case scenarios [48, 55] or using multimode multimedia terminals [50, 26, 39]. The most important property of multimode multimedia terminal is that the embedded system has multiple modes; the user specifically selects the current mode, where a mode describes which tasks are active. None of the aforementioned approaches take the dynamism within applications into account.

The dynamism within the applications and between the different applications is growing. To store this large amount of dynamism, the application scenarios must be stored as efficient as possible. For this purpose, our scenario database applies event trace compression. In the domain of trace-driven micro-architecture simulation, trace compression techniques are widely studied (e.g., [44, 35]). The efforts in this domain also include loop detection techniques [14]. However, the traces from these simulations typically consist of machine instructions or memory references, instead of high-level application events like in our case.

3.6 Conclusion

Application scenarios describe the dynamic workloads of modern embedded systems. The growing dynamism in a workload of embedded systems is twofold: inter-application scenarios describe which applications are active simultaneously, whereas intra-application scenarios describe the dynamism within applications. A first step when working with application scenarios is to identify them. The identification is partly automatic and partly manual. Intra-application scenarios are detected automatically using a profiling method, whereas the inter-application scenarios are defined manually. To store the identified application scenarios a scenario database is used to compactly store the event traces of all the processes in the multi-application workload. For this purpose, the event traces are compressed by removing recurrent behavior and to apply loop detection. Experiments showed the efficiency of the scenario database that was between the 0.1% and the 6.6% of the size required by the uncompressed event traces.

In this chapter, we have also showed the dynamism of the MJPEG application by comparing the non-functional properties (like execution time and power). Clearly, the execution time and power differs per mapping, but this variance is also unpredictable. As a result, comparing two mappings is non-trivial. The fitness corresponding to different intra-application scenarios may lead to different conclusions with respect to the Pareto dominance relation.

The next chapter will discuss the modified DSE procedure that takes into account uncertainty with respect to the Pareto dominance relation for different intra-application scenarios. Such a DSE will obtain a representative subset that is capable of correctly predicting the Pareto dominance relation between different mappings.

4

Scenario-based Design Space Exploration

This chapter is based on:

- P. van Stralen and A. D. Pimentel. 'Scenario-Based Design Space Exploration of MPSoCs'. In: *Proceedings of IEEE International Conference on Computer Design (ICCD '10)*. Oct. 2010

In the previous chapters it became clear that modern embedded systems become more and more dynamic. The application behavior of these, typically software-centric, MPSoC based embedded systems is described using application scenarios. As the number of application scenarios is relatively large and diverse, the diversity of the application scenarios needs to be taken into account during the embedded system design.

To cope with the design complexities of MPSoC based embedded systems, system-level design has become a promising approach for raising the abstraction level of design, and thereby increasing the design productivity. Early design space exploration (DSE) is an important ingredient of such system-level design, which has received significant research attention in recent years. However, the majority of these DSE efforts still evaluates and explores MPSoC architectures under single-application workloads. In this chapter, we therefore exploit the concept of application scenarios to introduce *scenario-based DSE*.

An important problem that needs to be solved by scenario-based DSE is the fact the number of possible application scenarios is too large for an exhaustive evaluation of *all* the design points with *all* the scenarios during the MPSoC DSE. Therefore, a representative subset of scenarios must be selected for the evaluation of MPSoC design points. This representative subset must compare mappings and should lead to the same performance ordering as would have been produced when the complete set of the application scenarios would have been used. Looking back at the experiment in Section 3.4.2, the selection of such a representative subset is not trivial. Not only did the experiment illustrate that a single application scenario is not sufficient to judge the relative quality of two different mappings, but also that the representative subset is dynamic with respect to the current set of mappings that are explored.

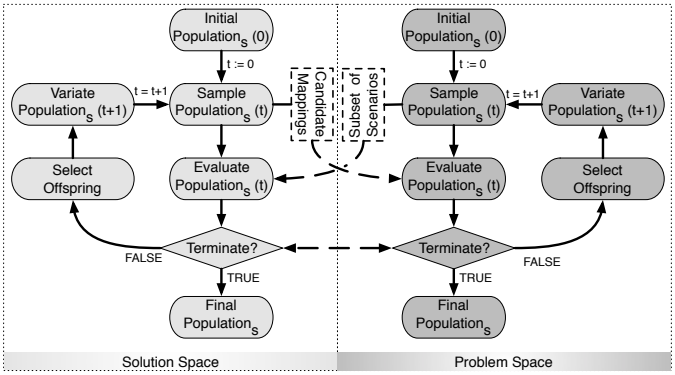


Figure 4.1: *A coevolutionary genetic algorithm.*

Depending on the set of mappings, a different subset of application scenarios may reflect the relative mapping qualities of the majority of the application scenarios.

As a result, the representative subset cannot statically be selected. For the static selection one should have a large fraction of the mappings that are going to be explored during the MPSoC DSE. These mappings are only available during the DSE and, therefore, a dynamic selection method must be used. To this end, we use a coevolutionary genetic algorithm [53, 43] where during the coevolution the search for the representative subset is done simultaneously with the search for the optimal mapping. In this way, the representative subset is able to adapt to the current set of mappings.

This chapter is organized as follows. Section 4.1 will introduce the coevolutionary genetic algorithm. Next, the solution space and the problem space will be discussed in Section 4.2 and Section 4.3. The solution space is concerned with the search for the optimal mapping, whereas the problem space deals with the identification of the representative subset. Section 4.4 discusses some experiments. Finally, the last two sections will discuss some related work and provide a short conclusion.

4.1 Coevolutionary Genetic Algorithm

The goal of scenario-based DSE is to identify a set of optimal mappings of the applications onto the architecture. Such a mapping defines both the allocation of the architectural components for the MPSoC and the binding of the application tasks and communication channels onto the architecture. To boost performance, a representative subset of scenarios is used instead of evaluating each design point using all scenarios. As the representativeness of this scenario subset is dependent on the specific mapping, the DSE problem is ill-defined.

To solve the DSE problem both the representative subset of scenarios and the best set of mappings have to be found. These two elements are dependent and need to be

solved at the same time. The most suitable technique to solve an ill-defined problem [43] is to use a *coevolutionary genetic algorithm*. Generally, a coevolutionary GA can be split in two parts: a solution part and a problem part. The GA tries to solve the problem, even if the problem is not completely defined yet. While searching for the solution, the problem is also being fully defined. To implement a coevolutionary GA, there are two approaches [43]: a combined and a separate genotype. The combined genotype encodes the solution and the problem within a single chromosome, whereas the separate approach uses two separate chromosomes. For scenario-based DSE, the separate genotype approach is used and the GA is based on the widely used SPEA-2 GA [81].

Strictly, a coevolutionary genetic algorithm with two separate genotypes can be seen as two GAs running in parallel. As illustrated by Figure 4.1, one GA is dedicated to the *solution space* and one GA is dedicated to the *problem space*. For scenario-based DSE the solution space consists of a set of mappings that allocate the architecture and bind the application(s) onto it. The goal of the solution space is to optimize the mapping given a representative subset of scenarios. This representative subset of scenarios is searched by the problem space. The embedded system designer limits the size of such a representative subset. In this way, the required time for the early DSE can be limited.

One of the crucial aspects of a coevolutionary genetic algorithm is the interaction between the two populations. Common types of behavior are symbiotic or competitive [53]. In a symbiotic situation both populations cooperate in order to maximize their fitness. An example of a symbiotic relation is the interaction between bees and flowers. Bees gain benefit from the flower by gathering nectar to use as food, whereas the flower benefits from bees as they take care of spreading the pollen of the plant to the next plant. Hence, the bee is taking care of the reproduction of the flower. The competitive behavior is the other way around. What is good for one population is bad for the other. An example is a predator prey behavior. The predator means to improve his hunting capabilities, whereas the prey tries to improve his defensive measures. The scenario-based DSE can be classified as symbiotic. The goal of the representative subset is not to find the worst-case scenarios, but the scenarios that are able to correctly predict the ordering of the mappings. Hence, if the fitness of the representative subset is high, the quality of the identified mappings also should improve.

4.2 Solution Space: MPSOC Mapping

The solution space is responsible for exploring the mapping design space. As can be seen in Figure 4.1, a traditional GA is used where a population of mappings is evaluated over time. The only addition is the exchange of information between the problem space and the solution space. Each generation a sample of the mapping population is taken and communicated to the problem space. Afterwards, the most

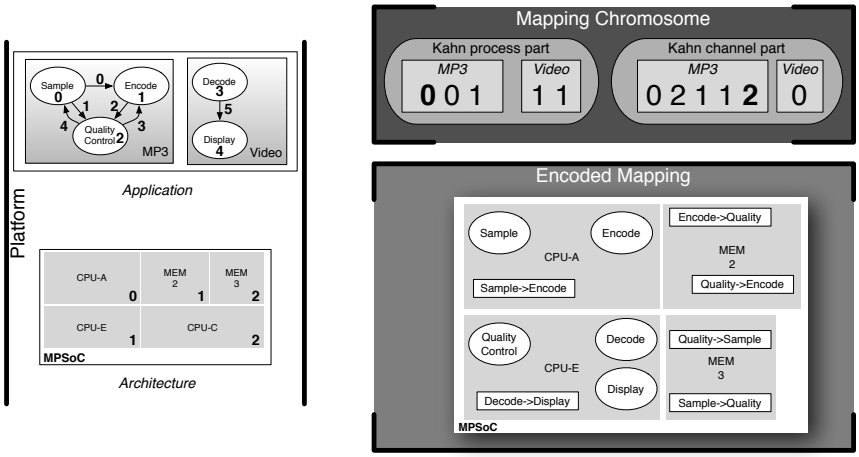


Figure 4.2: *Chromosome representation of a mapping. Both the gene sequence is shown and the mapping that is encoded by the gene sequence.*

recent representative subset of scenarios is taken from the problem space. This representative subset of scenarios is used to evaluate the current population of mappings.

Figure 4.2 shows the chromosome design for the solution space. The mapping chromosome consists of two parts: 1) a Kahn process part and 2) a Kahn channel part. Within these parts, all of the applications are encoded consecutively. The gene values encode the architectural components on which the elements of the applications are mapped; The Kahn processes are mapped onto processors and the Kahn channels are mapped onto memories. A special memory is the internal memory. The internal memory is only available when both the reader and writer of a Kahn channel are mapped onto the same processor. If a channel is mapped onto internal memory, only the local memory of the processor is used.

The example chromosome in Figure 4.2 has 11 genes. Five genes are dedicated to the processes and six genes are dedicated to the communicational channels As there are three potential processors, the gene value for the Kahn process part is between 0 and 2. For the memories there are three possibilities: two memories and a reserved entry for the internal memory. In this way, the bounded architectural component is encoded for each of the processes and channels. The first process gene of the MP3 application, for example, has gene value 0. Looking at the platform, gene 0 is the SAMPLE process. This process is to be mapped on the first processor: CPU-A. Similarly, the channel 4 (QUALITY → SAMPLE) is mapped on MEM-3. The complete encoded mapping is illustrated in Figure 4.2.

4.2.1 Fitness Function

As discussed earlier, it is infeasible to do an exhaustive evaluation, using all the scenarios, of all the individuals in the solution space. Therefore, we need to estimate the

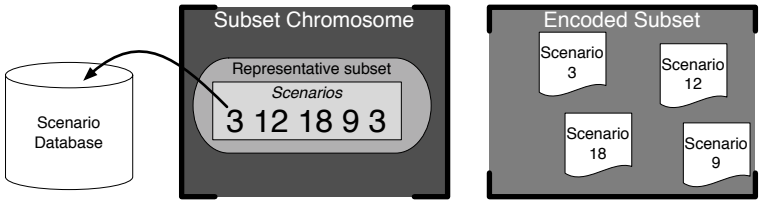


Figure 4.3: *Chromosome representation of a subset. Both the gene sequence is shown and the representative subset of scenarios that is encoded by the gene sequence.*

fitness of the application mappings. There are several methods for estimating the fitness of a solution [34]. Examples are problem approximation, data-driven functional approximation and fitness inheritance. Data-driven functional approximation tries to train a function that, given the inputs, provides the fitness of the solution. Fitness inheritance, on the other hand, bases the fitness of an individual chromosome on its parents. For scenario-based DSE problem approximation is used. *Problem approximation* tries to replace the original problem statement by an alternative statement that is approximately the same, but that is cheaper to solve. Scenario-based DSE tries to statically identify the mapping that, *on average*, behaves the best for all the scenarios.

4.3 Problem Space: Application Scenario Subset

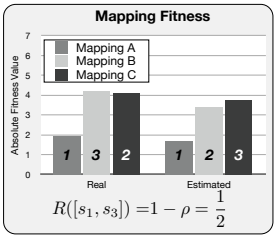
The problem space is responsible for identifying the representative subset of scenarios. Just as the solution space, the problem space is based on a traditional GA. As Figure 4.1 shows, the only additional step is that every generation the best representative subset of the population is sampled and communicated to the solution space. Next, a set of mappings from the solution space is obtained. This set of mappings will be used for the trainer, as we will discuss later.

A chromosome (as illustrated in Figure 4.3) for the problem space encodes a complete representative subset. Each gene encodes a single scenario of the representative subset. The scenario is encoded using the integer key of the scenario in the scenario database. A single scenario is allowed to occur more than once in the representative subset. In this case, fewer scenarios need to be evaluated and the repetitive scenario gets more weight within the average fitness of the evaluated mappings.

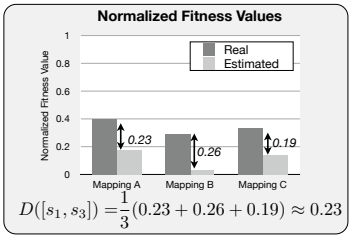
4.3.1 Fitness Function

The fitness of a subset of scenarios should reflect the representativeness of the scenario subset. In order to obtain the representativeness of a subset of scenarios, the behavior of the complete set of scenarios must be known. For this purpose, a *trainer* T is used that consists of a small number of mappings that are evaluated exhaustively (i.e.,

(a) *Exhaustively evaluate training mappings.*



(b) *Get real and estimated fitness.*



(c) *Calculate average fitness deviation.*

$$F_p([s_1, s_3]) = 0.5D([s_1, s_3]) + 0.5R([s_1, s_3]) = 0.5 * 0.23 + 0.5 * \frac{1}{2} = 0.37$$

(d) *Determine rank correlation and obtain the representativeness of the subset.*

Figure 4.4: *An example fitness evaluation of the subset* $[d_1, s_3]$.

using all the scenarios). Hence, the exact fitness for each of the mappings is known and, therefore, the quality of a subset in the problem population can be obtained. The quality of a subset $F_p(s)$ is determined for each individual objective (like execution time and energy):

$$D(s) = \frac{1}{|T|} \sum_{i=0}^{|T|} \text{abs}(\overline{F}(T[i]) - \overline{F^s}(T[i])) \tag{4.1}$$

$$R(s) = 1 - \rho(T, F, F^s) \tag{4.2}$$

$$F_p(s) = \alpha * D(s) + (1 - \alpha) * R(s) \tag{4.3}$$

The problem fitness F_p is composed of two individual metrics: *fitness deviation* (Equation 4.1) and *rank correlation* (Equation 4.2). Main reason of using two metrics is that the individual flaws of the metrics make it infeasible to use them individually. Therefore, the metrics are combined using the weighting factor α (Equation 4.3). These two metrics are not inherently conflicting. If the fitness deviation of a subset is zero, the subset also has an optimal rank correlation. Currently, we have only used an equal weighting ($\alpha = 0.5$) for both metrics.

Figure 4.4 shows an example of the fitness calculation. In this example, the scenario database consists of four scenarios. First, a trainer is required. This trainer is shown in Figure 4.4a and consists of three mappings (A, B and C). Each of these mappings is exhaustively evaluated for all of the four scenarios. Using this trainer, the representativeness of a scenario subset can be obtained.

To obtain the representativeness of the scenario subset (i.e., the fitness of the subset), its fitness approximations for the training mappings have to be calculated (Figure 4.4b). In this way, the real fitness of the training mappings and the estimated

fitness of the scenario subset are known. Using the real and the estimated fitness values of the training mappings, the two metrics for the representativeness of the subset can be evaluated:

1. **Fitness Deviation:** At first, the fitness deviation measures the difference between the real fitness $\overline{F}$ (based on exhaustive simulation) and the estimated fitness $\overline{F^s}$ based on the representative subset s. The fitness deviations are normalized with respect to the fitness values of the specific mapping (see Definition 5 on page 28).

   Figure 4.4c shows how the fitness deviation of our example subset fitness evaluation can be calculated. In the case of mapping B, for example, the real fitness is 4.2 and the estimated fitness is 3.4 (as shown in Figure 4.4b). The range of fitness values of mapping B is between 3.3 and 6.4 (see Figure 4.4a). As a result, the normalized value of the real fitness is $(4.2 - 3.3)/(6.4 - 3.3) = 0.29$. Similarly, the normalized estimated fitness is $(3.4 - 3.3)/(6.4 - 3.3) = 0.03$. The absolute difference of these normalized fitness values is 0.26. As shown in the formula for $D([s_1, s_3])$ in Figure 4.4c, the fitness deviation for mapping A and mapping C is 0.23 and 0.19. This results in an average fitness deviation of approximately 0.23.

2. **Rank Correlation:** Secondly, the Spearman's rank correlation is used [69] to measure the correlation between two variables. In our case, the real fitness and the estimated fitness are compared. At first, the mappings are ranked according to the real and the estimated fitness. This can be seen in Figure 4.4b. For the real fitness, the ranking is as follows: 1) Mapping A, 2) Mapping C and 3) Mapping B (this means that mapping A is the best and mapping B is the worst). The estimates given by our scenario subset are: 1) Mapping A, 2) Mapping B and 3) Mapping C. As there are no ties, the Spearman's rank correlation can be calculated as follows:

$$\begin{aligned}\rho &= 1 - \frac{6 \sum d_i^2}{n\,(n^2 - 1)} \\ &= 1 - \frac{6((1-1)^2 + (3-2)^2 + (2-3)^2)}{3\,(3^2 - 1)} \\ &= 0.5\end{aligned}$$

   In this equation n is the number of training mappings and d_i is the difference between the estimated and the real rank of training mapping i.

   The Spearman's rank correlation is always in the range between -1 and 1. A correlation of 1.0 corresponds to a perfect match between the ranking of the real and estimated fitness, whereas a correlation of 0.0 denotes that there is absolutely no relation between both rankings. Correlation can also be negative. In this case, the estimated fitness gives exactly the opposite mapping ordering

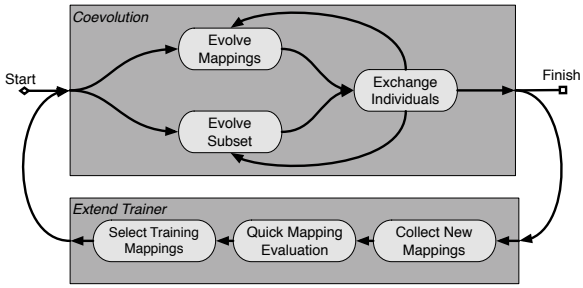


Figure 4.5: *The trainer selection during the coevolution.*

than the real fitness. As this is undesirable, the rank correlation must be as close to 1.0 as possible. Therefore, we subtract the Spearman's rank correlation from 1.0 in order to transform it into a minimization problem.

The final fitness of our example subset is the weighted average of the two aforementioned metrics: 0.37.

4.3.2 Trainer Selection

In the previous subsection, we discussed how the fitness of an individual subset is obtained. The fitness calculation of a scenario subset, however, is completely dependent on the trainer that is used. If the trainer is not representative, the obtained subset cannot be representative either. Therefore, it is crucial to obtain a representative trainer.

While selecting a trainer, there are two conflicting requirements: 1) *size* and 2) *quality*. On one hand, the trainer must be as compact as possible. The fewer mappings that there are in the trainer, the less exhaustive evaluations that are required. On the other hand, the quality of the trainer must be as high as possible. The more mappings there are in the trainer, the higher the probability that the trainer reflects the current population of the solution space (i.e., the set of mappings that is currently used to find the optimal mappings).

In order to keep a modestly sized trainer that reflects the current state of the solution space, we have chosen to dynamically update the trainer during the coevolution. Periodically, the trainer is updated with a small number of mappings that is sampled from the current population in the solution space. In this way, the trainer can reflect the part of the solution space that is searched at that specific point in time.

The procedure is illustrated in Figure 4.5. The core of the procedure is the coevolutionary process. During this process both spaces (solution and problem space) are coevolved. After each generation, some of the individuals are exchanged. For the solution part, this means that a new representative subset is read to predict the fitness of a new set of mappings. The problem part, on the other hand, will collect the

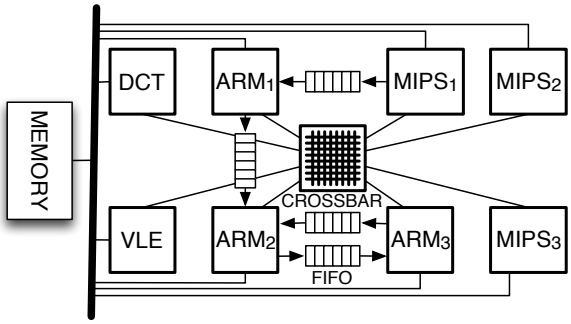


Figure 4.6: *The used architecture model for the experiments in this chapter.*

sampled mappings for later use. Periodically, the coevolutionary process is halted. At this halting point the continuation depends on the user-defined exploration time (i.e., the total time the DSE is running). In case all the exploration time has passed, the DSE is finished after the exchange of individuals. Otherwise, the trainer is updated before the coevolutionary process is continued.

For the trainer extension, the collected mappings in the problem space are used. First, these mappings are filtered such that no duplicate mappings will be added to the trainer. Next, a quick evaluation is done for the potential new trainer mappings. For this quick evaluation, a subset of scenarios is used that is obtained by selecting a limited number of the mostly used scenarios in the complete set of scenario subsets. Finally, the new trainer mappings are selected based on the standard deviation of the mapping fitness on the different scenarios that are evaluated. The mappings with the highest diversity are added to the trainer. As the diversity is high, the real average fitness is harder to predict. Therefore, the trainer uses these mappings to improve the fitness prediction in the solution space. Before the selected mappings are added to the trainer, they are evaluated exhaustively.

As the size of the trainer effects to execution time of the DSE, we have built in several measures to control the execution time and how this time is spent. As the Sesame simulation calls account for the majority of the execution time, the designer can specify the maximal number of Sesame invocations. This is done both for mapping evaluation in the solution space and the trainer extension for the problem space. Given the number of remaining Sesame invocations, it is decided how many training mappings can be added to the trainer. The more Sesame invocations are assigned to the problem space, the higher the potential quality of the trainer.

4.4 Case Studies

In order to evaluate our coevolutionary DSE approach, the DSE of two types of multi-application workloads are studied. The first workload is composed of two real world applications, whereas the second workload only contains synthetic applica-

Table 4.1: *General settings for the (coevolutionary) GA*

Solution Space		Problem Space	
Population size	200	Population size	150
Offspring size	30	Offspring size	25
$P_{crossover}$	0.8	$P_{crossover}$	0.8
P_{mutate}	0.01	P_{mutate}	0.02
Generations	600	Gen-train	10

tions. By also using synthetic applications, the experiment has more control on the characteristics of the multi-application workload. This control is used to create a more workload that is more dynamic than our real world multi-application workload. For each of the two multi-application workloads a DSE is performed that entails the search for optimal design instances with respect to cost and execution time. The used heterogeneous MPSoC platform that is shown in Figure 4.6 contains eight processing elements: three MIPS processors, three ARM processors and two dedicated ASICs for DCT and VLE operations. For communication the platform architecture contains a crossbar with private memory buffers, four dedicated point-to-point FIFOs and a bus connected to a shared memory. As we are exploring mappings, not all of these components need to be allocated in the final design.

To illustrate the added value of scenario-based DSE, we compare it against two other approaches: 1) a static approach and 2) an exhaustive approach. The static approach randomly selects the representative subset of scenarios before the DSE is started. This scenario subset will be subsequently be used to evaluate the mappings in a single GA for searching the design space. The exhaustive approach, on the other hand, uses the complete scenario database to evaluate each individual mapping.

General settings of the GAs of the solution and problem spaces are given in Table 4.1. The settings of the solution space are used for each of the approaches, whereas the settings of the problem space are only used for the coevolutionary approach. For both spaces, the first four parameters are traditional GA parameters: the size of the population, the number of offspring per generation and the probability for mutation and crossover. Based on the convergence of the initial experiments, the parameters are chosen and there is decided to run the exploration for 600 generations. During the exploration, the trainer is updated every ten generations. This gives a clear view of the complete convergence of the DSE. To show the variance of the different runs of the DSEs (the GA has stochastic nature), each experiment of the realistic workload is repeated twelve times, whereas each experiment of the stochastic workload is repeated six times.

Finally, each experiment will result in a Pareto front. These Pareto fronts can

be compared using the hypervolume indicator (see Section 2.3). Before the hypervolume can be calculated, the resulting mapping fitness values must be exhaustively evaluated. Remember that both the coevolutionary and the static approaches use a representative subset of scenarios to estimate the fitness. Most likely both approaches will use different subsets and thus the mappings must be evaluated using the same scenarios before they can be compared. Moreover, for each 15th generation the current Pareto front is logged such that it can be evaluated afterwards. In this way, the convergence can be observed.

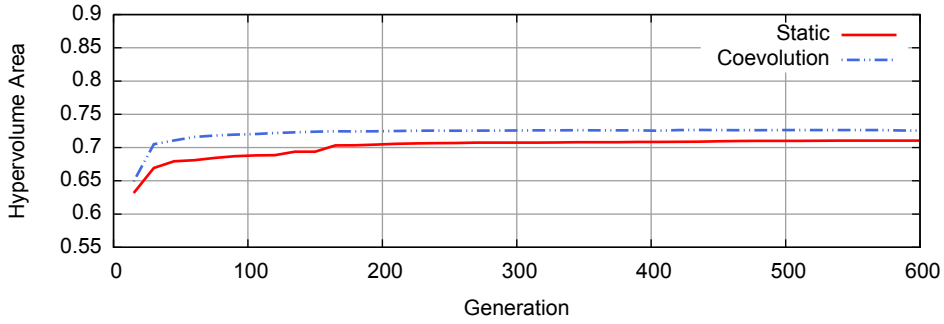
4.4.1 Real Workload

For the real multi-application workload, two applications are used: a MJPEG encoder and a MPEG4 decoder (only for the simple profile, so without any frame reordering). The workload has a total of 131 scenarios: the MJPEG encoder has eleven intra-application scenarios, whereas the MPEG4 decoder has ten scenarios. As all the possible inter-application scenarios are valid (the applications can both run individually and simultaneously) the total number of scenarios is $10 + 11 + 10 * 11 = 131$.

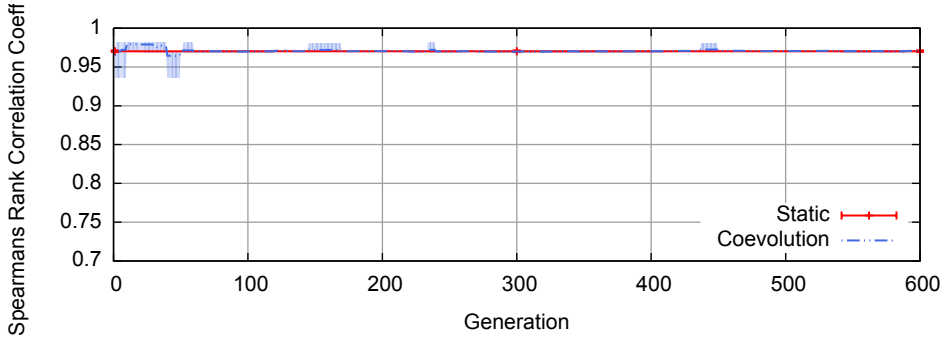
The statistically selected subset contains five scenarios, whereas the dynamically selected subset, on the other hand, may maximally contain four scenarios. This distinction is done deliberately as in this way the average number of Sesame invocations per generations is exactly the same for the static and dynamic selection. The additional effort that the coevolutionary approach requires for evaluating the trainer mappings is compensated in the static approach by using a larger subset of scenarios. In this manner, a fair comparison can be made as both approaches have a similar running time.

Figure 4.7 shows the results of the experiments. One of the important concerns of a GA is the time it takes until the resulting Pareto front stabilizes (i.e., convergence). For this purpose, we have analyzed the hypervolume over time. The results of this analysis are shown in Figure 4.7a. For both the static and the coevolutionary approach the average hypervolume is shown (averaged over the different repetitive experiments). Both of the approaches converge relatively quickly. Within 100 generations the Pareto front is more or less stabilized. The coevolutionary approach, however, converges to a better Pareto front. On average, the hypervolume is 0.02 larger. With respect to the found mappings, this means that the mappings found by the coevolutionary approach have an execution time that is on average 2 percent smaller.

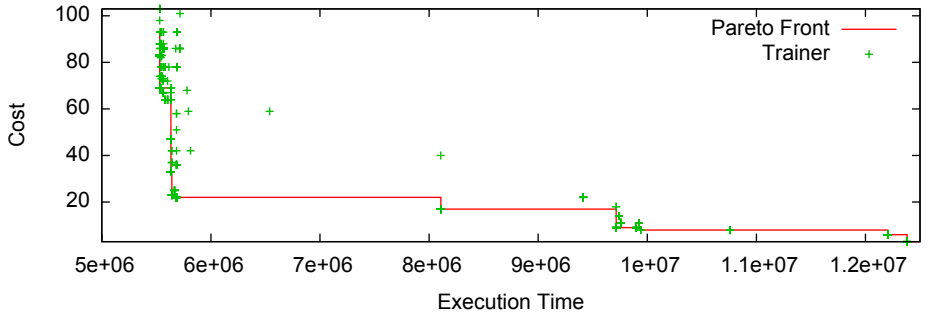
This is not such a large difference. Most likely, the statically selected subset is good enough to correctly predict the ranking of the solution individuals. In order to verify this, we have calculated the Spearman's rank correlation of all the representative subsets that are obtained. For this purpose, a large set (more than 14K) of exhaustively evaluated solution individuals is used. This set is unequal to the trainer and, therefore, the representative subset is not specifically trained on this set of solution individuals. Recall that when the Spearman's rank correlation is closer



(a) *Convergence of Pareto front*



(b) *Quality of representative subset*



(c) *Used training mappings*

Figure 4.7: *Experimental result of DSE using a real multi-application workload*

to 1.0, the ranking of the representative subset provides a better resemblance of the real ranking. Results in Figure 4.7b shows the average ranking correlation over the different runs. It is indeed clearly visible that the quality of the subsets for the static and coevolutionary approach is comparable. As the static approach selects the representative subset before the DSE is performed, the quality of the subset does not change over time. Therefore, the correlation of the static approach (the horizontal red line) remains fixed at approximately 0.97. The scenarios in the coevolutionary

subset change over time, but its quality still fluctuates around 0.97 (the blue dashed line). The range of values for the coevolutionary approach is slightly better as shown by the blue shaded region. The best subset found by the coevolutionary approach has a ranking correlation of 0.98, whereas the best statically selected subset has a ranking correlation of 0.97.

A conclusion that can be drawn from this experiment that with this real multi-application workload the coevolutionary approach does not significantly benefit with respect to the static approach. The reason is the lack of diversity in the application workload. As the diversity is relatively small, it is quite likely that a randomly selected subset is already representative for the complete scenario database.

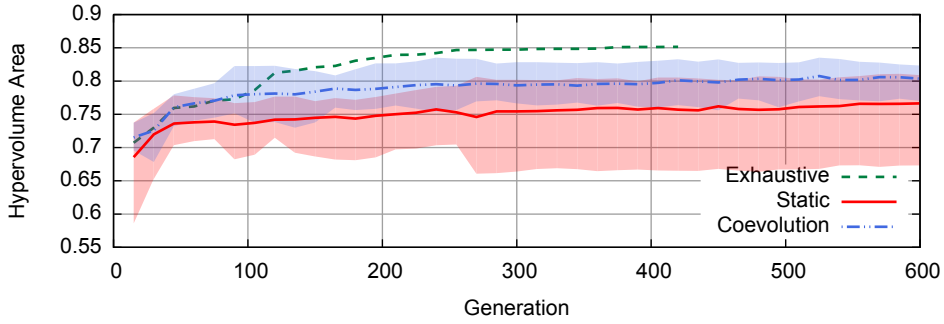
4.4.2 Synthetic Workload

To assess the merit of the coevolutionary approach for multi-application workloads with a higher diversity, we also preformed our previous experiment using a synthetic workload. The synthetic workload contains 18 applications with a total of 72 application tasks. With 27 inter-application scenarios and 51 intra-application scenarios, the workload now has a total of 514 scenarios. The static scenario subset uses a subset of ten scenarios and the coevolutionary approach has a maximal subset size of six scenarios. As discussed in the previous section, this makes sure that the number of Sesame invocations is the same for the static and the coevolutionary approaches. The consequence is that the coevolutionary approach must put additional effort to let its representative subset outweigh the random scenario subset.

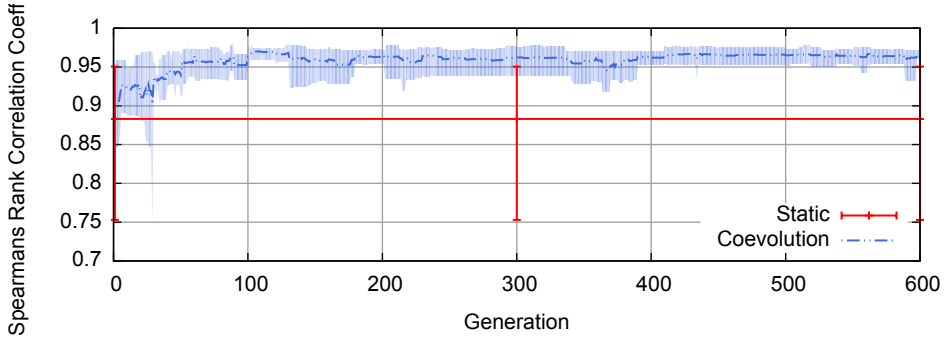
In contrast to the previous section, we also included a DSE where the mappings are evaluated exhaustively. The exhaustive approach is only run once, as its running time is an order of magnitude larger than the static and coevolutionary approaches that use fitness prediction. This is clearly visible in Figure 4.9. The 420 generations of the exhaustive approach already require 43 million Sesame invocations, whereas the static and coevolutionary approaches only require a million Sesame invocations for 600 generations. For a Sun Fire X4440 with four quad-core AMD Opteron 8356 processors running at 2.3GHz, this means that the exhaustive approach has a running time of 23 minutes per generation. The static and the coevolutionary approaches, on the other hand, only take 36 seconds per generation.

Figure 4.8 shows the results of the experiment. In Figure 4.8a the Pareto front convergence for the different approaches is shown. Evidently, the exhaustive approach outperforms the other approaches. At each generation the average hypervolume of the exhaustive approach is higher (i.e., better) than the other approaches. As the SPEA2 based GA uses elitism and the exhaustive approach evaluates the mapping using the real fitness, the strong individuals that are found will always be kept in the population. This leads to a hypervolume that is monotonically increasing. However, the exhaustive approach is much more expensive. As discussed earlier, it is more than 30 times slower than the static and coevolutionary approaches.

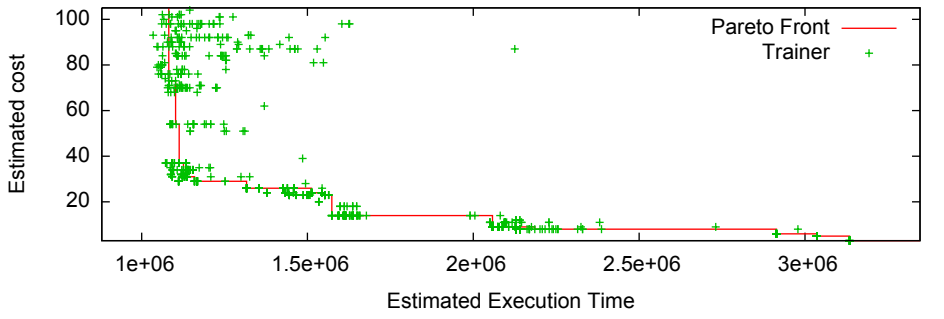
The static and the coevolutionary approaches have a similar running time. With



(a) *Convergence of Pareto front*



(b) *Quality of representative subset*



(c) *Used training mappings*

Figure 4.8: *Experimental result of DSE using a real multi-application workload*

respect to convergence, the coevolutionary approach has better results than the static approach. More precisely, the average hypervolume (shown with a line within Figure 4.8a) of the coevolutionary approach is better starting from the first measuring point. Where the hypervolume of the coevolutionary approach increases almost monotonically, the average hypervolume of the static approach even drops between the 200th and the 300th generation. In this case, the fitness prediction fails to predict the correct ordering. For the coevolutionary approach, however, the representative subset

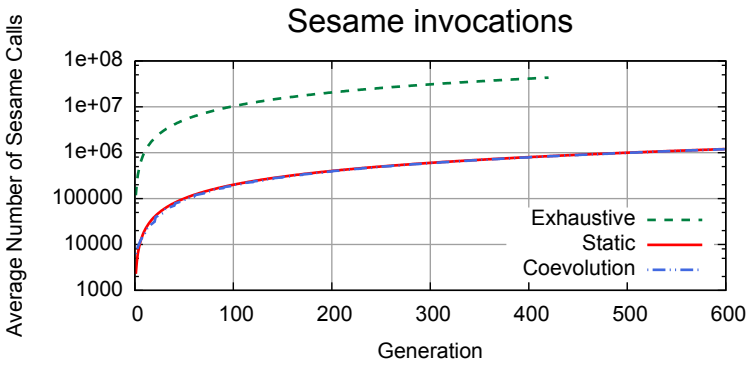


Figure 4.9: *A comparison of the number of Sesame invocations for the different approaches. The horizontal axis shows the number of generations, while the logarithmic vertical axis shows the total number of Sesame invocations made during the earlier generations.*

is adequate for keeping the good individuals in the population. Hence, the selection manages to get a good representative subset with only 1.2 percent of the total number of scenarios.

More importantly, the coevolutionary approach does not only obtain better results, but the outcome of the DSE is also more stable. In Figure 4.8a the variance of the static and coevolutionary approaches is highlighted using a shaded region. The variance of the coevolutionary approach is shown with blue and for the static approach it is shown using a red region. These regions can also overlap, which is shown using the somewhat darker region. In the initial phases, not only the hypervolume of the coevolutionary and the static approaches is similar, but also the variance. Over time, the coevolutionary approach is improving the subset. Therefore, the subset prediction can be improved. As a result, the hypervolume improves and the variance becomes smaller. For the static approach, the subset remains the same. In some cases, the initial subset is adequate for identifying decent Pareto fronts. Due to the randomness of the static approach, however, some poor subset will also lead to poor Pareto fronts. A poor subset will be uncertain about the mapping ordering and will make incorrect conclusions with respect to Pareto dominance. On top of that, the longer the exploration takes, the more variance there will be between the quality of the obtained Pareto fronts of a decent and a poor scenario subset.

The poor subsets can also clearly be seen when the subset quality is measured. Just as in the previous experiment, a large set of exhaustively evaluated individuals is used to obtain the ranking correlation of the used scenario subset with the real ranking. For the results of the static and the coevolutionary approach, as shown in Figure 4.8b, different visualizations are used. As the static subset does not change over time, the subset quality is also constant over time. Hence, a straight line is used

with for error line (shown at generation 0, 300 and 600) that shows the extremes of the subset quality. On average, the ranking correlation is 0.88. The quality variance, however, is quite large: the subset of the best run has a ranking correlation of 0.95, whereas the subset of the worst run has a ranking correlation of 0.75. In contrast to the static approach, the quality of the coevolutionary approach changes over time. That is why a line is used for the average quality and a shaded region to show the variance. Initially, the subset quality of the coevolutionary approach is similar to the static approach (between the 0.85 and 0.95). However, the quality of the subset quickly becomes better. Eventually, the quality of the scenario subset stabilizes between 0.96 and 0.97.

So, for a scenario database with more diversity the coevolutionary approach clearly outperforms the static approach. Even the subset of the worst run of the coevolutionary approach is better than the subset of the best run of the static approach. The higher diversity in the scenario subset makes it hard to quickly select a good scenario subset.

4.4.3 Trainers

As a final experiment, we have visualized both a trainer from the real and the synthetic workload. The trainers are shown in Figure 4.7c and 4.8c. The line shows the Pareto front that is found in the specific DSE. For embedded system designers this is the most important outcome of the DSE that shows the trade-off between performance and cost. Within the graph also all the final trainer mappings are shown using markers. It can be seen that most of the selected training mappings are close to the Pareto front. The GA in the solution space mostly focuses on this area as it tries to improve the Pareto front. In general, the differences between these mappings are small and, therefore, the mappings are hard to differentiate. As a result, imprecise fitness prediction is more likely to give a better fitness to the inferior mappings. Due to this incorrect fitness, the SPEA2 based genetic algorithm may create offspring mappings of a lower quality as the wrong parents are selected.

Also observe that in Figure 4.8c some of the training mappings dominate the mappings that are found in the final Pareto front of the solution space. This issue will be resolved in the next chapter, where the final outcome of the DSE will also take the training mappings into account.

4.5 Related Work

High-level modeling, simulation for MPSoC performance evaluation and GA based DSE [25, 15] are relatively mature research areas. The majority of the research used to focus on the system exploration under a single (fixed) workload. Recently, research has been initiated on making the DSE scenario aware [67, 59, 52, 16].

One of the techniques is by using the scenario-aware dataflow model [71, 72].

The scenario-aware dataflow model allows the modeling of dynamic applications using application scenarios. Multiple analysis tools are available to obtain metrics like throughput and latency. However, the disadvantage of analysis in comparison to simulation is that analytical models quickly become very complex. Hence, there are two options: 1) the architecture is modeled in an abstract manner (e.g., properties like resource contention are not taken into account) or 2) the architecture is completely modeled. In the first case analysis is fast but it may be imprecise. For the second case, the analytical model quickly becomes too complicated or too computationally expensive.

Another type of scenario is the use-case scenario. A use case can be compared with what we call inter-application scenario: it describes which applications can run concurrently. Examples of frameworks deploying such use cases are MAMPS [40] and a framework of Benini [4]. MAMPS is a system level synthesis tool for mapping multiple applications on an FPGA. The framework of Benini, on the other hand, reconfigures the embedded system whenever a new use case is detected. A notion of multiple applications can also be incorporated using a multimode multimedia terminal [26] that captures the inter-application scenario behavior in a single, heterogeneous model of computation. In this model an application only has fixed behavior that is described using a dataflow model, whereas a finite state machine describes the transition between the different inter-application scenarios. Where scenario-aware dataflow uses the finite state machine to switch between the operation modes of a single application, the multimode multimedia terminal switches between different sets of active tasks. The work of [64] also uses a finite state machine to model the dynamism within a multi-application workload. In this case, each application can be running or is paused. A scenario transition is explicitly done by starting, stopping, pausing or resuming an application.

None of these approaches describe both the intra-application scenarios and the inter-application scenarios. To the best of our knowledge, our scenario-based DSE is the first to address both inter- and intra-application scenarios during the DSE of MPSoCs with multi-application workloads.

On top of that, to the best of our knowledge, coevolution is also not yet used in the field of DSE of MPSoCs. Other research areas, however, have applied coevolution to reduce the number of scenarios or test cases for the efficiency improvements of GAs. Branke and Rosenbusch [8] use coevolution to improve the efficiency of their worst-case optimization problem. This problem tries to optimize the variable s of the mathematical function $F(s,t)$ such that the worst-case value for all possible test cases t is as high as possible. The interesting property for the worst case value is that as long as the worst-case test is in the population, the predicted fitness matches the real fitness. Therefore, no exhaustively evaluated trainer is required (in contrast to the average case optimization in our DSE problem).

Schmidt and Lipson [63] apply coevolution using an exhaustively evaluated trainer. Their optimization problem is to increase the efficiency of a GA by using a fitness

predictor. The fitness predictor is coevolved with the GA and consists of a subset of all the samples in the problem definition. A set of exhaustively evaluated individuals is used to judge the quality of the fitness predictor based on the absolute prediction error. This is one of the major differences with our work. Not only do we normalize the prediction error (in this way each objective is weighted equally), but also the relative ordering is taken into account. After all, the fitness predictor is only meant to predict which individual is better than the other. In this way, the DSE ends with best set of individuals. The fitness values of these individuals may have a systematic (absolute) error, but that does not affect the identified Pareto front.

4.6 Conclusion

This chapter introduced the use of application scenarios during the system-level DSE of MPSoC based embedded systems. More specifically, a novel scenario-based DSE is proposed based on a coevolutionary genetic algorithm. This approach searches simultaneously for optimal design instances and representative subsets of workload scenarios to efficiently evaluate these instances.

Our use cases showed that, as long as the diversity in the scenario database is large enough, the coevolutionary DSE clearly outperforms the DSE where a representative subset is selected statically. Additionally, it is an order of magnitude faster than a traditional DSE where the design instances are evaluated exhaustively using the complete scenario database. In case the diversity in the scenario database is low, the approach still can provide similar results than the static subset selection. As the static subset can compensate the overhead of the dynamic subset selection by using a larger scenario subset, this means that the computational overhead of the dynamic subset selection pays off.

The improvement of scenario-based DSE is twofold. First, the obtained Pareto fronts have a higher hypervolume. This means that the design instances that are found are faster and cheaper. Secondly, the dynamic subset selection of the scenario-based DSE also has a higher probability on a good outcome than a static subset selection. The variance of the hypervolume over the different runs for the scenario-based DSE decreases over time, whereas the variance in hypervolume for the different runs only increases over time. This means that the scenario-based DSE is less dependent on the stochastic nature of the GA leading to a more reliable exploration framework.

In the next chapter we will refine the scenario-based DSE by formalizing the solution and the problem space. Additionally, the problem space is completely revised.

5 Fitness Prediction Techniques

This chapter is based on:

- P. van Stralen and A. D. Pimentel. 'Fast Scenario-Based Design Space Exploration using Feature Selection'. In: *Proc. of the Int. Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA'12).* Feb. 2012
- P. van Stralen and A. D. Pimentel. 'Fitness Prediction Techniques for Scenario-based Design Space Exploration'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 32.8 (Aug. 2013), pp. 1240–1253

Earlier, we have introduced scenario-based DSE. Scenario-based DSE rapidly evaluates mappings for multi-application workloads during the search through the MPSoC design space. The multi-application workload is described using application scenarios. Hence, the scenario-based DSE explicitly takes the existence of multiple applications into account. Without the explicit modeling of the multi-application workloads there are only two alternatives for designing an embedded system with multiple applications: 1) Isolate all the applications on the architecture or 2) design for the case where all the applications are active. Both of the approaches have their own potential drawbacks. The isolation of the applications completely disregards resource sharing. One of the consequences of the lack of resource sharing is that communication structures like *Time Division Multiple Access (TDMA)* are required. The design for the case where all applications are active may take resource sharing into account, but it still results in an over-designed system.

In order to rapidly evaluate the mappings, the scenario-based DSE explores the optimal mappings in the solution space simultaneously with the optimal fitness predictor in the problem space. This fitness predictor is a representative subset of scenarios that is used to predict the quality of a mapping. Such a fitness predictor is of utmost importance for the overall quality of the DSE. In the previous chapter (Section 4.3), the problem space used a genetic algorithm to identify the optimal

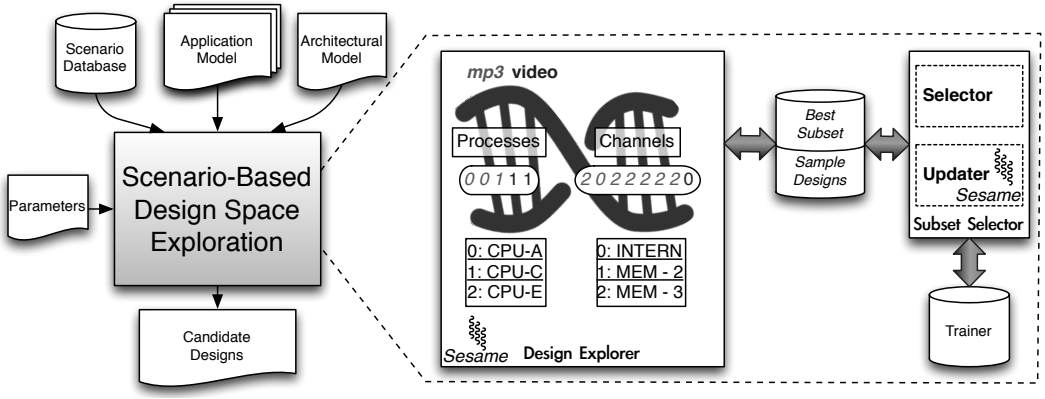


Figure 5.1: *The refined exploration framework for scenario-based DSE.*

scenario subsets. Although experiments showed the effectiveness of the fitness prediction by means of a representative subset of application scenarios, there are more potential methods to select a representative subset of scenarios. In this chapter, we will investigate two alternatives.

The chapter is organized as follows. At first, Section 5.1 will give a refined design of the scenario-based DSE. Based on this refined design, Section 5.2 gives a formal definition of the mapping procedure of an MPSoC design instance. Next, Section 5.3 describes the fitness prediction techniques that are used to select the representative subset of scenarios. The fitness prediction techniques are compared and validated in Section 5.4. At the end of the chapter, a short conclusion is given.

5.1 Refined DSE Framework

Conceptually, scenario-based DSE is an exploration technique for embedded systems with a dynamic multi-application workload. In this section, an exploration framework for scenario-based DSE is presented that aims to provide a *static mapping* of a multi-application workload onto a MPSoC. The mapping is to be used during the entire lifetime of an embedded system. Consequently, the average behavior of the designed MPSoC must be as good as possible for all the different application scenarios. Currently, we assume an equal likelihood for each application scenario. The approach, however, can easily be generalized to different probability distributions.

Figure 5.1 shows the exploration framework. The left part of the picture provides a general flow, whereas the right part illustrates the scenario-based DSE in more detail. As input, the scenario-based DSE requires a scenario database, application models and an architecture model. As a mapping specifies both the allocation and binding (see Section 2.1.4), the architectural model typically contains more architectural components than actually fit on the final MPSoC. The allocation will only select a subset of the architectural components that is used for the final design.

Binding is performed for a multi-application workload and the description of this workload is split into two parts: 1) the structure and 2) the behavior. The structure of the applications is described using the KPN models of the application. This model describes all the processes and communication channels that need to be mapped. Next to the KPN models, the scenario database is used to describe the behavior of the multi-application workload. This behavior is not relevant for the mapping, but it is required for the evaluation of each mapping.

Ideally, all the mappings are exhaustively evaluated for all of the application scenarios in the database. Practically, however, this is infeasible. Therefore, fitness prediction is applied using a *representative subset of scenarios*. As motivated in the previous chapter, this scenario subset needs to be selected dynamically by co-exploring both the set of optimal mappings and the representative subset of scenarios. In the refined framework (Figure 5.1), two separate components are shown that simultaneously performs these tasks. At first, the design explorer searches for the set of optimal mappings. Secondly, the subset selector tries to select a representative subset of scenarios. As these components are running asynchronously, a shared memory is present that is used to exchange data. For the design explorer, a sample of the current mapping population is stored in the shared memory, whereas the subset selector makes the most representative subset available for the fitness prediction in the design explorer. One of the main advantages of the strict separation of the execution of the design explorer and the subset selector is that the running time of the design explorer becomes more transparent. From a user perspective, this is the most important component, as it will identify the set of optimal mappings.

Finally, the speed of the scenario-based DSE also depends on the machine that is used to run the framework. Therefore, we have also ported our scenario-based DSE to a compute server with a high degree of chip multithreading and analyzed its performance behavior. For the results, please refer to [84, 92].

5.2 Design Explorer

In this section, the design explorer, the component that is responsible for identifying promising mappings, is specified. First, the used system model is described. This system model formally describes both the applications and the architecture. Next, the system model is used to describe the complete mapping procedure that is used during the search of the GA.

5.2.1 System Model

Figure 5.2 shows the Sesame model that was introduced in Chapter 2 with the three conceptual layers in Sesame: 1) the application model, 2) the mapping layer and 3) the architecture model. The system model formalizes each of these layers:

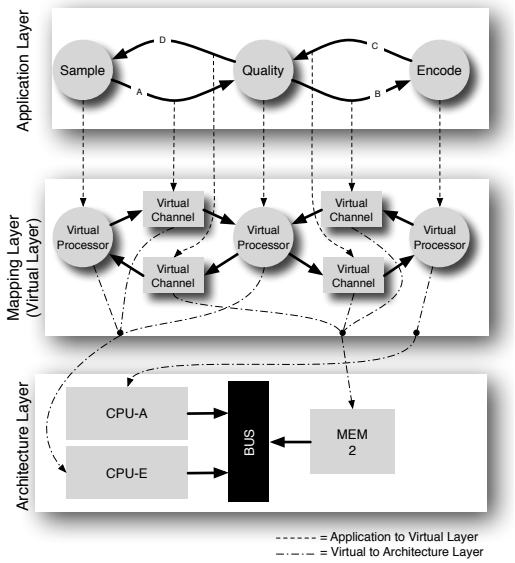


Figure 5.2: *The different layers in the Sesame model and their connections.*

Application layer The application model describes each individual application using a KPN. A KPN is formally defined as a directed graph $G_K(V, E_K)$. The vertexes V are the Kahn process nodes. For the example in Figure 5.2, V is equal to {SAMPLE, ENCODE, QUALITY}. The communication channels of the application are represented by directed edges $E_K = V \times V$. If, for example, (SAMPLE, QUALITY) is defined in E_K, it means that there is a communication channel from SAMPLE to QUALITY.

Architecture layer The architecture is represented by a directed graph $G_R(R, E_R)$. In this case, the set R contains the architectural components like processors, communication buses, crossbars, FIFOs and memories. Edges $E_R = R \times R$, on the other hand, describe the communication links in the architecture.

There are three types of architectural elements: 1) processors, 2) buffers and 3) interconnects. The processors $R_P \subset R$ are architectural elements that are capable of running processes. Buffers $R_B \subset R$ are the FIFO buffers used for the communication between the different processors. If two communicating processes are mapped onto the same processor, the communication may also be done internally. This is only possible when a processor supports internal communication. If a processor $p \in R_P$ supports internal communication, a buffer $b \in R_B$ is added to the architecture. Additionally, the buffer is connected to enable reading and writing of data: $(p, b) \in E_R \wedge (b, p) \in E_R$. Finally, the set of interconnects $R_I \subset R$ is purely meant to connect the various system components.

In the example of Figure 5.2, the architectural processors E_P consist of CPU-A

and CPU-E. Next, the FIFO buffers in the architecture are MEM-2 and CPU-E. This means that CPU-E supports internal communication. Finally, the BUS belongs to the set of interconnects. There are 8 edges in the architecture. Six of these links are connected to the bus (for reading and writing). Additionally, the internal communication buffer of CPU-E is connected to the processor for reading and writing. In this example, all communication links are bidirectional. An example of unidirectional communicational links is the FIFO queue in Figure 4.6 on page 73.

Mapping Layer The mapping layer connects the application layer to the architecture layer. Hence, it contains only edges: computation edges and communicational edges. Computation mapping edges E_X assign Kahn process nodes to the architectural resources. To be precise, the edge $(v, p) \in E_X$ assigns Kahn process $v \in V$ to processor $p \in R_P$. A Kahn process can only be mapped on processing elements that are feasible of running that task:

$$(v, p) \in E_X \iff \text{Feasible}(p, v) \tag{5.1}$$

An example of a processor that only is able to perform a limited set of tasks is shown in Figure 4.6. In this architecture two ASICs are present: one can only execute DCT transformations and one can only perform VLE encodings.

The communication is mapped using communication edges E_C. A communication edge (c, b) maps FIFO channel $c \in E_K$ to FIFO buffer $b \in R_B$.

5.2.2 Mapping Procedure

Where the application and the architecture layers are predefined before a DSE is started, the mapping layer is the part of the MPSoC design that is optimized. As discussed before (see Figure 2.5 on page 19), the mapping is split into two steps: allocation and binding. Allocation can reduce the resource usage of the MPSoC design, whereas the binding maps all processes and channels on the allocated resources. The procedure is as follows:

Allocation First, the architectural resources are selected to use in the allocation α. All types of architectural resources are selected at once: $\alpha_P = \alpha \cap R_P$, $\alpha_B = \alpha \cap R_B$ and $\alpha_I = \alpha \cap R_I$. More precisely, the allocation α contains a subset of resources such that $\alpha \subseteq R$:

$$\left(\sum_{r \in \alpha} \text{area}(r) \right) \leq \text{MAX\_AREA} \tag{5.2}$$

Hence, Equation 5.2 says that the total area of the allocated resources may not be larger than the maximal area on the chip. Part of the system model is also

the feasibility of the mapping: for each of the processes there must be at least one processor that is feasible of running the specific process. This is defined as follows:

$$\forall v \in V : |p \in \alpha_P : \text{Feasible}(p, v)| \geq 1 \tag{5.3}$$

Once the allocation α is known, a set of potential communication paths $\psi = (\alpha_P \times \alpha_B \times \alpha_P)$ can be defined:

$$\psi = \{(p_1, b, p_2) : \text{PATH}_\alpha(p_1, b) \wedge \text{PATH}_\alpha(b, p_2)\} \tag{5.4}$$

The set of communication paths ψ is the set of paths such that (:) there is a path from processor p_1 to buffer b and a path from buffer b to processor p_2 (Equation 5.4). This path may span multiple resources as long as they are allocated:

$$\text{PATH}_\alpha(r_1, r_2) := (r_1, r_2) \in E_R \vee \tag{5.5a}$$

$$\exists r_i \in \alpha_I : (r_1, r_i) \in E_R \wedge \text{PATH}_\alpha(r_i, r_2) \tag{5.5b}$$

The PATH function is recursively defined. There is a path between resources r_1, r_2 if there is a direct connection between them (Equation 5.5a). An interconnect r_i can also be used as part of the path. In this case, there must be a direct connection between resource r_1 and interconnect r_i and a path between interconnect r_i and resource r_2 (Equation 5.5b). An allocation is only valid if there is at least one communication path between each set of processors:

$$\forall p_1, p_2 \in \alpha_P : \exists (p_1, b, p_2) \in \psi \tag{5.6}$$

By enforcing at least a single communication path between each set of processors, the automatic exploration of mappings is guaranteed to find at least one valid mapping. As will be explained later, the procedure randomly picks the processors after which one of the communicational paths is selected.

Binding Binding maps all the Kahn process nodes onto the allocated resources. There are two steps: 1) computational binding and 2) communication binding. Computational binding β_X maps the processes onto the processors such that $\beta_X \in E_X$:

$$\forall v \in V : |\{p : (v, p) \in \beta_X \wedge p \in \alpha_P\}| = 1 \tag{5.7}$$

Equation 5.7 enforces that each process v is mapped on exactly one allocated processor p. After the computational binding, the communication binding can be done. Recall from Equation 5.6 that we have enforced that between each set of processors at least one communication path is present in ψ. Therefore, for each communication channel in the application, a communication path in the allocated architecture can

be selected. More strictly, for each communication channel in the application, an architectural buffer is selected such that $\beta_C \in E_C$:

$$\forall (v_1, v_2), b \in \beta_C : (v_1, p_1) \in \beta_X \wedge \tag{5.8a}$$

$$(v_2, p_2) \in \beta_X \wedge \tag{5.8b}$$

$$(p_1, b, p_2) \in \psi \tag{5.8c}$$

$$\forall c \in E_K : |\{b : (c, b) \in \beta_C\}| = 1 \tag{5.9}$$

Multiple conditions must be enforced. First, the architectural buffer b on which the communication channel (v_1, v_2) of the application is mapped must be a valid communication path. This means that processes v_1 and v_2 must be mapped on processors p_1, p_2 (Equation 5.8a and 5.8b) and that (p_1, b, p_2) is within the set of communication paths ψ (Equation 5.8c). Processor p_1 and processor p_2 does not necessarily need to be different as both of the processes in the communication link may be mapped onto the same processor. Next, all communication channels c (which is a tuple of the two communication processes) must be mapped on exactly one buffer (Equation 5.9).

A mapping m is the combination of an allocation α and the bindings β_X and β_C. It is only valid if all the preceding constraints (Equations 5.2, 5.3, 5.6, 5.7 and 5.9) are fulfilled.

5.2.3 Genetic Algorithm

Our aim is to optimize the mapping of an embedded system. Hence, the space of possible mappings must be explored as efficiently as possible. For this purpose, an NSGA-II based multi-objective GA [12] is used. NSGA-II is an elitist selection algorithm that applies non-dominated sorting to select the offspring individuals. Non-dominated sorting ranks all the design points based on their dominance depth [10]. Conceptually, the dominance depth is obtained by iteratively removing the Pareto front from a set of individuals. After each iteration the rank is incremented. An example can be seen in Figure 5.5c.

In the previous chapter, a SPEA2 based GA was used. NSGA-II has a similar performance (with respect to diversity and convergence) than SPEA2 [37]. The main reason of choosing the NSGA-II based GA is because of its use of the dominance depth for optimization. As will be discussed in Section 5.3.2, the dominance depth can easily be used for rating the quality of the representative subset.

The dominance of the individuals is based on their fitness. As discussed before, the predicted fitness is used instead of the real fitness. Let S be the total set of scenarios and $\tilde{S}_j$ the representative subset of scenarios at time step j. The fitness

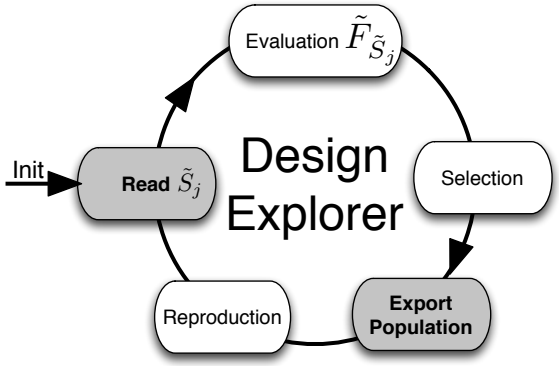


Figure 5.3: *The genetic algorithm for the design explorer extended with the required steps to communicate with the subset selector. The steps that are emphasized involve communication.*

objectives of a mapping m are as follows:

$$F(m) = \frac{1}{|S|} \sum_{s \in S} (\text{time}(m,s), \text{energy}(m,s), \text{cost}(m)) \tag{5.10}$$

$$\tilde{F}_{\tilde{S}_j}(m) = \frac{1}{|\tilde{S}_j|} \sum_{s \in \tilde{S}_j} (\text{time}(m,s), \text{energy}(m,s), \text{cost}(m)) \tag{5.11}$$

The functions $\text{time}(m,s)$ and $\text{energy}(m,s)$ are evaluated using Sesame. Given the mapping m and the application scenario s, the execution time and used energy are determined by means of simulation. The cost of a mapping is independent of the scenario and can be determined statically by adding up the costs of the individual elements within the allocation α. There is an important difference between the *real* fitness F and the *estimated* fitness $\tilde{F}_{\tilde{S}_j}$. The real fitness uses all possible scenarios to determine the fitness, whereas the estimated fitness only uses a subset of the scenarios ($\tilde{S}_j \subseteq S$). As a result, the real fitness is independent of the current generation. The predicted fitness, on the other hand, may vary over the different generations. The fitness $\tilde{F}_{\tilde{S}_j}$ is only valid for generation j as the representative subset $\tilde{S}_{j+1}$ may change over time.

In order to update the representative subset of scenarios in between the generations, the GA of the design explorer must be extended to support the communication between the design explorer and the subset selector. This extension is shown in Figure 5.3. Before any individual can be evaluated, the currently most representative subset of scenarios $\tilde{S}_j$ must be acquired. Using the representative subset of scenarios, the design explorer can quickly predict the fitness of all the individuals in the population. This means that, depending on the number of changed scenarios in the representative subset of scenarios since the previous generation, the parent population also must be partially reevaluated. This predicted fitness is used to select the

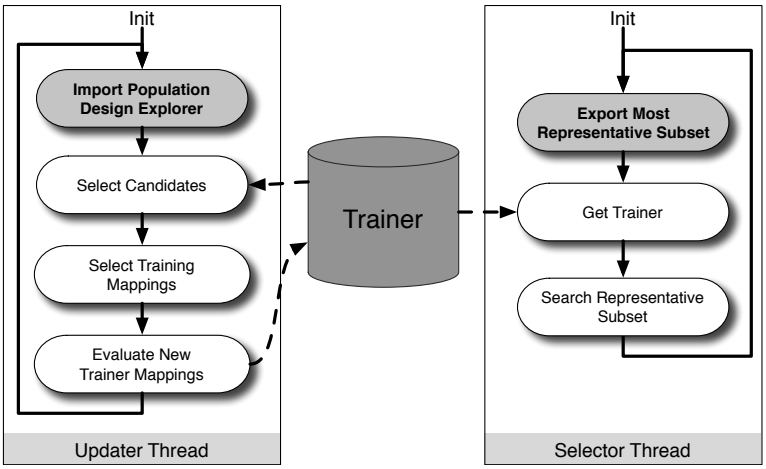


Figure 5.4: *The design of the subset selector.*

individuals for the next generation. In case the scenario subset is representative, the decisions made by the NSGA-II selector are similar to those where the real fitness would have been used. If this is not the case, the scenario subset should be improved. For this purpose, the selected population is exported to the subset selector. Finally, reproduction is performed with the selected individuals. During reproduction, a new population of individuals is generated for usage in the next generation.

5.3 Subset Selector

To work properly, the design explorer requires a representative subset of scenarios. The better the fitness prediction in the design explorer, the better the outcome of the scenario-based DSE. Therefore, the subset selector is responsible for selecting a subset of scenarios. This subset selection is not trivial. At first, there are a potentially large number of scenarios to pick from. This leads to a huge amount of possible scenario subsets. On top of that, the scenario subset cannot be selected statically as the representativeness of the scenario subset is dependent of the current set of mappings. This set of mappings is only available at runtime. Therefore, the scenario subset is selected dynamically.

At the end of the previous section, it already became clear that the design explorer communicates its current mapping population to the subset selector. This set of mappings can be used to train the scenario subset such that it is representative for the current population in the design explorer. As the population of the design explorer slowly changes over time, the subset will change accordingly. The overview of the scenario-based DSE (Figure 5.1) showed that the subset selector contains two threads of execution: the selector thread and the updater thread. Figure 5.4 shows these threads in more detail. The updater thread obtains the mapping individuals from

the design explorer and updates the training set T_i of application mappings. This set of training mappings is used by the selector thread for selecting a representative subset of scenarios. The most representative subset of scenarios is exported to the design explorer.

In the remainder of this section we provide a detailed view of the subset selector. Before doing so, however, we will first describe the updater thread that is responsible for updating the trainer. Next, the metric is described that is used to judge the quality of the scenario subsets. The final subsection will show how the subset quality metric is used within the selector thread to select scenario subsets.

5.3.1 Updater Thread

During the search of a representative subset of scenarios it is crucial to have a set of training mappings T_i. Without a set of exhaustively evaluated mappings, one cannot judge if the predicted fitness of a scenario subset makes a correct prediction. As a training mapping is evaluated for all scenarios, it is relatively expensive to evaluate a mapping that needs to be added to the trainer. Therefore, it is important that the training mappings are selected carefully. Figure 5.4 illustrates the trainer update from T_i to T_{i+1}. The steps are as follows:

Import Population Design Explorer: To keep the training set T_i up to date with the mapping population in the design explorer, the current design explorer population g_j is imported.

Select Candidates: The current population is used to update the list of candidate mappings C_{i+1}:

$$\begin{aligned} &\underset{C_{i+1}}{\text{maximize}} \sum_{m \in C_{i+1}} \text{last\_gen}(m) \\ &\text{subject to } (1)\ C_{i+1} \subseteq C_i \cup g_j \\ &\qquad\qquad (2)\ C_{i+1} \cap T_i = \emptyset \\ &\qquad\qquad (3)\ |C_{i+1}| \leq \text{C\_SIZE} \end{aligned}$$

While updating the candidate mappings there are three conditions. First, the new set of candidate mappings is the union of the previous set of candidate mappings and the population g_j that is just received from the design explorer. Secondly, condition two makes sure that all the candidate mappings are not yet in the trainer. Using these two conditions, the procedure selects a set of candidate mappings that is new to the trainer. Still, the first two conditions do not provide any control on the size of the set of candidate mappings. As the selection of training mappings involves computational overhead, the size of the set of candidate mappings must be limited as well. Therefore, condition three makes sure that the size of the set of candidate mappings is not larger than the predefined constant C_SIZE. As the optimization

goal is to optimize the sum of the last generation that each of the training mapping is used (as returned by the function last_gen), the most recently used mappings will be kept in the set of candidate mappings (these have the highest value for last used generation). Additionally, the optimization of the total sum tries to get the number of candidate mappings as large as possible: the least recently used candidate mappings will be removed until the set of candidate mappings is smaller or equal to C_SIZE. In this way, the representative subset of scenarios can be optimized to predict the fitness of the current population of the design explorer.

Select Training Mappings: For each of the candidate mappings the predicted fitness using the currently most representative subset of scenarios is obtained. For the candidate mappings that are just imported from the design explorer, this should not require any new Sesame simulations (the population is quite likely evaluated using the same representative subset). For older candidate mappings some computational overhead may be required for the partial reevaluation of the mapping fitness. Together with the mappings in the current trainer T_i, an estimated Pareto front $\tilde{P}_j$ is obtained.

The main goal of the representative subset is to correctly identify most optimal mappings. Therefore, the trainer will focus on the mappings that are the closest to the Pareto front. Any mapping may have a fitness that is hard to predict (a mapping with a high quality or a mapping with a poor quality), but the scenario-based DSE only suffers from high quality mappings that have an incorrectly predicted fitness. As long as both the real and predicted fitness of a mapping is bad, it does not really matter how bad the predicted fitness is. However, it is still an issue if the predicted quality of a mapping is poor, whereas the real quality is good. In this case, the mapping will not be added to the trainer. Although this is undesirable, without exhaustively evaluating the candidate mappings these kind of incorrect predictions cannot be detected. As the exhaustive evaluation is expensive, the gain in the trainer quality does not outweigh the additional computational overhead that is required to identify the high quality mappings where the predicted mapping quality is low. Over time the predicted ordering of the mappings near the predicted Pareto front will be improved. Likely, this will also improve the prediction of other mapping individuals.

Therefore, the k new training mappings M_c are selected from the set of candidate mappings C_{i+1} by optimizing the distance to the estimated Pareto front:

$$\begin{aligned} \underset{M_c}{\text{minimize}} \quad & \sum_{m \in M_c} \min_{m_p \in \tilde{P}_j} \left(\overline{d} \left(\tilde{F}_{\tilde{S}_j}(m_p), \tilde{F}_{\tilde{S}_j}(m) \right) \right) \\ \text{subject to } & (1)\ M_c \subseteq C_{i+1} \\ & (2) |M_c| = \min(|C_{i+1}|, k) \end{aligned}$$

The mappings are ordered on their normalized Euclidean distance (see Definition 5 on page 28) to the closest mapping in the estimated Pareto front $\tilde{P}_j$. From the

candidate mappings (condition 1) the k mappings are selected (condition 2) that are the closest to the estimated Pareto front.

Evaluate New Training Mappings: The mappings that are selected are exhaustively evaluated using Sesame. For this purpose, a separate pool of Sesame workers is used (just as the design explorer has a pool of Sesame workers). Once the real fitness is known, the mappings can be used to generate trainer T_{i+1} out of trainer T_i. Before the new training mappings are added, the trainer is truncated to fit the new training mappings. This is done in such a way that the trainer always contains real Pareto front P:

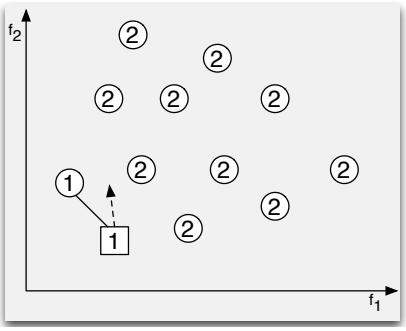
$$\begin{aligned} \underset{T_{i+1}}{\text{minimize}} \quad & \sum_{m \in T_{i+1}} \min_{m_p \in P} \left(\overline{d} \left(F(m_p), F(m) \right) \right) \\ \text{subject to } & (1)\ T_{i+1} \subseteq T_i \\ & (2)\ |T_{i+1}| = \min(|T_i|, \text{T\_SIZE} - |M_c|) \end{aligned}$$

The truncated new trainer is a subset of the old trainer (condition 1) and it does not exceed the predefined trainer size. If trainer mappings must be discarded, the mappings that are the furthest from the real Pareto front are removed. This is done because of the second purpose of the trainer: at the end of the scenario-based DSE it contains the best mappings that are found over time with their real fitness. Hence, we assume that the maximal trainer size is picked in such a way that it is significantly larger than the size of the Pareto front P. After truncation, the next trainer can be finalized: $T_{i+1} = T_{i+1} \cup M_c$.

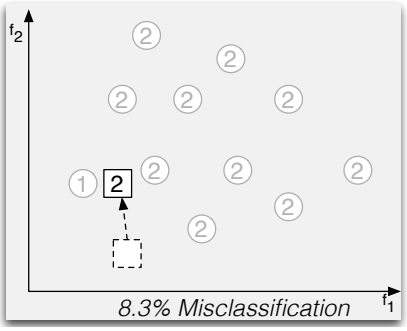
5.3.2 Subset Quality Metric

Having a set of training mappings is not sufficient for judging the quality of the scenario subsets. In the previous chapter (Section 4.3.1) the fitness deviation and the Spearman's rank correlation was used for the subset fitness. For a correct prediction, the Spearman's rank correlation of a scenario subset is the most important. A problem is, however, that whenever the Spearman's rank correlation is less than one a higher rank correlation does not necessarily corresponds to a better fitness prediction of a scenario subset. For this purpose, the fitness deviation was added: the average difference between the real and the estimated fitness.

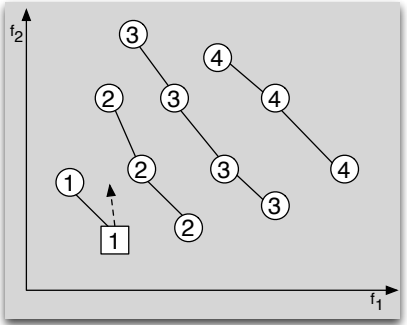
In retrospective, this was not the best approach. The main purpose of the scenario-based DSE is to identify the best set of mappings. Hence, a deviation in fitness or lower rank correlation may not affect the identified set of best mappings. Additionally, the fitness of a scenario subset was defined to be multi-objective: the fitness deviation and the rank correlation were defined per individual optimization objective for the MPSoC mapping. Although this does not prevent us from finding the best subset, the subset search becomes easier once the quality of a subset is independent of the number of mapping optimization objectives. In this way it is always



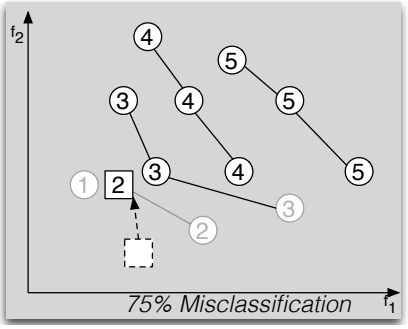
(a) *Boolean ranking, real fitness*



(b) *Boolean ranking, predicted fitness*



(c) *Goldberg's ranking, real fitness*



(d) *Goldberg's ranking, predicted fitness*

Figure 5.5: *A set of Pareto fronts showing the effect of a small error in the prediction (as shown with the dashed arrow) on the misclassification rate using the boolean and Goldberg's ranking scheme.*

clear which scenario subset is the best (with multiple objectives there may be more Pareto optimal subsets).

That is why the *misclassification rate* is used as a subset quality metric. The misclassification rate counts the number of ranks that are predicted incorrectly. Before we go into the definition of the misclassification rate, we first take a look into the Pareto ranking [77]. There are several approaches to rank individuals using the Pareto dominance relations, but in this thesis we only focus on two of those: Boolean ranking and Goldberg's ranking (also called non-dominated sorting).

The ranking schemes are visualized in Figure 5.5. Goldberg's ranking approach uses the dominance depth of the individuals. This is the same approach as the NSGA-II selector. Boolean ranking, on the other hand, follows a more simple approach: if the solution is non-dominated the rank is one, otherwise the rank is two. As the design explorer uses an NSGA-II based GA, it may be straightforward to use Goldberg's ranking scheme for the misclassification rate. The boolean ranking, however, can be obtained more efficiently than the Goldberg's ranking. On top of that, the misclassification rate may be deceiving when Goldberg's ranking is used.

In Figure 5.5 an example of such a deceiving case is given. The used trainer consists of 12 training mappings. Figures 5.5a and 5.5c show the exact fitness of the training mappings, whereas Figures 5.5b and 5.5d show the predicted fitness of a specific scenario subset. This scenario subset provides a relatively good prediction: eleven out of the twelve training mappings are predicted correctly (the circular mappings). The incorrectly predicted training mapping (the square mapping) is slightly off as shown by a dashed arrow. Due to the incorrect prediction, the square mapping seems to be dominated by the leftmost training mapping. For both ranking schemes the rank of the square mapping becomes ranked second instead of first. In case of the boolean ranking this is the only rank that is incorrect. For Goldberg's ranking, however, all the training mappings that are dominated by the square mapping are also incremented by one. As a result, the Goldberg's ranking has a misclassification rate of $\frac{3}{4}$, whereas the boolean ranking has a misclassification rate of $\frac{1}{12}$. Our example clearly shows that a high quality mapping that is incorrectly ranked can affect all of its dominated solutions. However, as the main purpose of scenario-based DSE is that the Pareto front is predicted correctly, it is not a problem that the poor mappings are incorrectly ranked.

This is exactly what is determined with boolean ranking. Each rank is based on the correct prediction of a non-dominated individual. A formal definition of the boolean ranking can be seen in the following equation:

$$\mathrm{rel}_{F'}(m_1, m_2) := \begin{cases} 1 & F'(m_1) \text{ dominates } F'(m_2) \\ -1 & F'(m_2) \text{ dominates } F'(m_1) \\ 0 & else \end{cases} \tag{5.12}$$

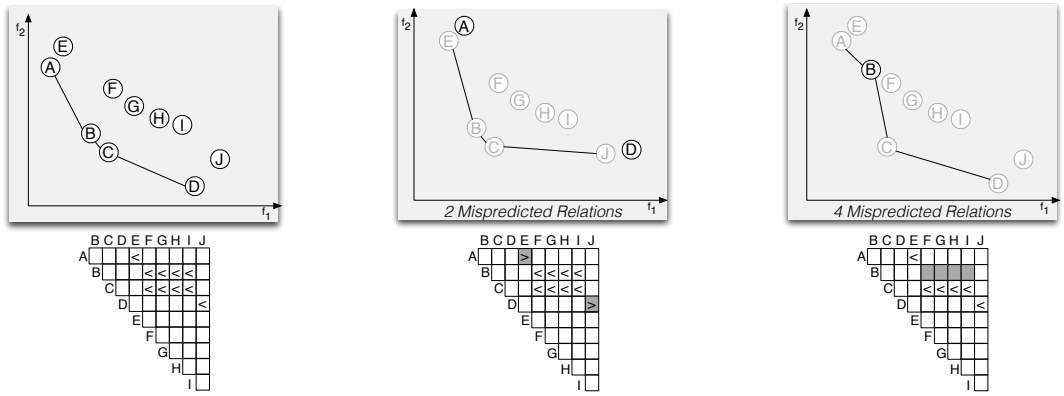
$$\mathrm{rank}_{F'}(m, T) := \begin{cases} 2 & \exists m' \in T \left(\mathrm{rel}_{F'}(m', m) = 1\right) \\ 1 & else \end{cases} \tag{5.13}$$

At first, Equation 5.12 formally defines the Pareto dominance between two mappings m_1 and m_2. The mappings are evaluated using fitness function F'. This can be the real fitness F, but also the predicted fitness $\tilde{F}_{\tilde{S}}$. In this case the scenario subset $\tilde{S}$ is used to predict the fitness of the mappings.

Based on the $\mathrm{rel}_{F'}$ function the $\mathrm{rank}_{F'}$ is defined. This function ranks mapping m given trainer T (see Equation 5.13). In case any of the mappings in the trainer dominates the mapping, the rank is equal to two. Otherwise, the mapping is Pareto optimal and the rank is equal to one. Given the ranking function the misclassification rate can be defined:

$$\mathrm{r}_{\mathrm{rank}}(\tilde{S}, T) := \frac{|\{m \in T : \mathrm{rank}_F(m, T) \neq \mathrm{rank}_{\tilde{F}_{\tilde{S}}}(m, T)\}|}{|T|} \tag{5.14}$$

The rate of misclassified boolean ranks is too coarse-grained to be used in isolation. In contrast to Spearman's rank correlation, a lower misclassification rate is



(a) *Real* **(b)** 50% *error ratio* **(c)** 0% *error ratio*

Figure 5.6: *A larger number of misclassified relations does not strictly correlate with the Pareto front quality. The Pareto front in (c) has more mispredicted relations than the front in (b), but the error ratio with respect to the real front is better.*

always better (the more non-dominated individuals that are correctly identified, the better). However, the probability of an equal misclassification rate is quite likely.

In this case the *number of misclassified relations* is used as a tiebreaker. The number of misclassified relations can be defined quite straightforwardly:

$$\mathrm{r}_{\mathrm{rel}}(\tilde{S}, T) := \frac{|\{m_1, m_2 \in T : \mathrm{rel}_F(m_1, m_2) \neq \mathrm{rel}_{\tilde{F}_{\tilde{S}}}(m_1, m_2)\}|}{|T|^2} \tag{5.15}$$

By definition when the number of misclassified relations is zero, the number of misclassified ranks is also zero. For the other cases, the number of misclassified relations can suffer from the same problem as we showed with Goldberg's ranking. An example is shown in Figure 5.6. Figure 5.6a shows the real Pareto front, where the fronts of Figure 5.6b and 5.6c are obtained using a predicted fitness. The first prediction (Figure 5.6b) only has two mispredicted relations (A $\leftrightarrow$ E and D $\leftrightarrow$ J), whereas the second prediction (Figure 5.6c) has four mispredicted relations. Still, the Pareto front of the first prediction is only correct for 50% (E and J are not Pareto optimal). The second prediction, which is worse according to the number of misclassified relations, correctly identifies the Pareto front. As we are using the number misclassified relations as a subordinate metric, and not as a main metric, this is no problem in our case. Figure 5.6b has a misclassification rate of 20% that is worse than the misclassification rate of 0% in Figure 5.6c.

5.3.3 Selector Thread

The selector thread uses the subset quality metrics to select the representative subset of scenarios. More specifically, the goal of the selector thread is as follows:

$$\underset{\tilde{S}}{\text{minimize}}\ \mathrm{r}_{\text{rank}}(\tilde{S},T):\ \underset{\tilde{S}}{\text{minimize}}\ \mathrm{r}_{\text{rel}}(\tilde{S},T) \tag{5.16}$$

As discussed in the previous subsection, the main goal is to optimize the quality of the predicted ranking. In the case of ties, the number of mispredicted relations will determine which of the scenario subsets is the best. Whenever a better representative subset is found, the subset is shared with the design explorer in order to improve its fitness prediction. The subset may be of any size, as long as it does not exceed a user-defined limit. This means that a smaller subset that has a better or equal representativeness is preferable to a larger counterpart (the smaller the subset is, the faster the fitness prediction).

As Chapter 4 showed, a random pick of scenarios does not result in a representative subset of scenarios. Hence, the subset of scenarios is selected dynamically. In the previous chapter, a GA was used to select the scenario subset. In this chapter, we explore three different techniques: 1) a genetic algorithm, 2) a feature selection algorithm and 3) a hybrid method.

Genetic Algorithm (GA)

Our first subset selection technique uses a GA to select the representative scenario subsets. The GA of the subset selector is somewhat similar to the GA in the design explorer: a population of individuals is evolved over time in order to find the individual with the highest fitness. In order to describe the individual, a chromosome is used that enumerates the scenarios that are contained in the scenario subset. This chromosome, as illustrated in Figure 4.2 on page 68, is simply a sequence of integers. As the length of the chromosome is equal to the limit of the scenario subset size, the scenario subset can never become too large. Smaller scenario subset sizes are achieved in two ways: 1) scenarios may be used more than once within the same chromosome and 2) there is a special value for an unused slot in the scenario subset.

Scenario subsets can change size as an effect of the mutation or crossover that is applied during the search of a GA. The evolutionary process uses mutation and crossover to prepare individuals for the next generation of scenario subsets. Where the mutation replaces the scenarios in the subset one by one with other scenarios, the crossover partly exchanges the scenarios of two subsets (see the illustration in Figure 2.8 on page 25). Only the successful modifications will make it to the next generation, leading to a gradual improvement of the representative subset of scenarios.

This approach has several benefits. First, the computational overhead is relatively small. Most time is spent in judging the quality of the scenario subsets and modifying the population of scenario subsets. Additionally, selecting scenario subsets for the next generation is relatively cheap. Apart from the low computational overhead, the

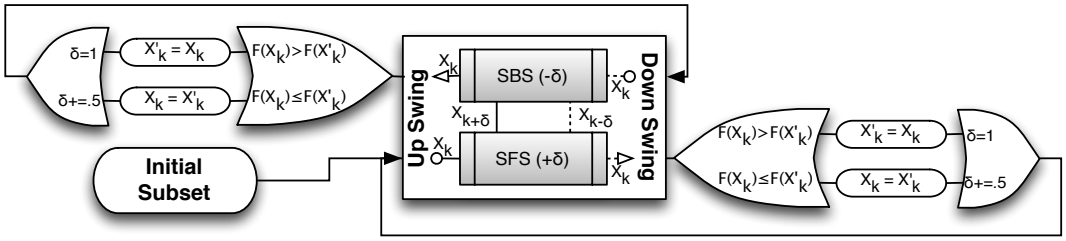


Figure 5.7: *An illustration of feature selection by means of the dynamic oscillation search procedure.*

search can also quickly move through the total space of scenario subsets. Due to crossover and mutation, the scenario subset can quickly change and the alternatives can be evaluated and explored. This also means that local optima can easily be avoided. A local optimum is a solution that is clearly better than all closely related solutions. A small change on a local optimum does not result in a better subset, but the local optimum may still be much worse than the global optimal solution. As a genetic algorithm always has a small probability that the scenario subsets are changed significantly in the next generation, there is always a probability that the search will move out of the local optimum.

Unfortunately, this is also the downside of the approach. Just when the search comes close to the optimal solution, the mutation can quickly move the search to a completely different direction. Although the likeliness of this all depends on the choice of the parameters like mutation and crossover probability, it may be quite hard to pick the parameters in such a way that search space is completely explored in the promising regions. Elitism in GAs assure that the points close to the local optimum are remained, but not that the neighborhood of each solution is carefully explored.

Feature Selection (FS)

It would be better, if the approach was less dependent on the choice of the parameters of the search algorithm. The *Feature Selection (FS)* technique has less parameters and it basically performs a guided search that tries to improve a single scenario subset step by step. There are many different feature selection techniques, each giving a different trade-off between computational overhead and its quality. In fact, the feature selection techniques with the lowest computational overhead actually use a GA. In our case, we have chosen to use the dynamic sequential oscillating search [68] as, in general, it provides better classifiers (i.e., the scenario subset that classifies the non-dominated mapping individuals), with a moderate computational overhead.

Figure 5.7 illustrates the dynamic oscillating search. The most fundamental part of the algorithm is the up and down swing. These swings are named according to their

effect on the size of the scenario subset. Where the upswing will modify the subset by first adding a number of scenarios to the subset and then removing the same number of scenarios, the downswing will first remove scenarios before new scenarios are added again. This explains the name of the upswing and downswing: in case of the upswing, the size of the subset swings upward and for a downswing it swings downward. For adding and removing scenarios the *Sequential Forward Selection (SFS)* and *Sequential Backward Selection (SBS)* are used. These techniques will iteratively add or remove a scenario in the most optimal way. This means that SFS will increase the scenario subset size by iteratively adding the most optimal scenario. This most optimal scenario is determined by trying out all possible scenarios from the scenario database and the scenario that results in the best scenario subset will be added to the larger subset. Similarly, the SBS will iteratively choose the optimal scenario to remove from the scenario subset. This means that all the scenarios that can be removed from the scenario subset are tried and the scenario removal that results in the best scenario subset will be applied.

As simple as it sounds, it makes the computational overhead of the swings largely dependent on the number of scenarios that are added or removed. The number of possibilities will grow linearly with respect with the number of scenarios that are added and removed during the swing. For each scenario that is added, all scenarios in the scenario database must be analyzed. Since this leads to a quick increase of computational overhead once the swings become larger, the size of the swing (or δ as used in Figure 5.7) is initialized to one and slowly increased. During the search, the up and down swings are alternated and whenever both the up and downswing do not result in a better scenario subset the size of the swing is incremented by one. This can be seen in Figure 5.7, at the cases where $F(X_k) \leq F(X'_k)$. The subset X'_k is the current best subset and the subset X_k is the subset that is obtained after the up or downswing. As a higher value for the function F means a better scenario subset, the case where $F(X_k) \leq F(X'_k)$ is an unsuccessful attempt to improve the scenario subset by a swing. Therefore, the currently best subset is restored and the size of the swing is increased by 0.5. The 0.5 is just a trick to increment the swing by one after two unsuccessful swings: the number of scenarios that are added is the truncated integer value of δ. Of course, the swing can also be successful: in that case the current best subset X'_k is updated and the swing size is reset to one.

In a sense, the dynamic oscillating search is a kind of hill climbing technique. It oscillates the size of the scenario subset by exhaustively exploring all possibilities to change the scenario subset. Whenever a better subset is found, the current best representative subset is updated. Important to realize is that the current best subset can also be updated during a swing. As SFS and SBS analyze the quality of the scenario subset for each scenario that is added, it can be the case that a better representative subset is found during the swing. If the size of this subset is smaller than the maximal size, the currently most representative subset is updated and sent to the design explorer.

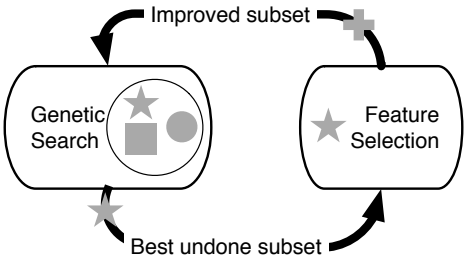


Figure 5.8: *The hybrid subset selection approach that alternates between a GA and a FS algorithm.*

The FS method is more directed than the GA and, therefore, it will only move closer to the optimal scenario subset. Unfortunately, this comes at a price: the FS is much more sensitive to local optima than the GA approach.

Hybrid Approach (HYB)

Ideally, we want to combine the strengths of the GA and the FS approaches. The hybrid approach (as shown in Figure 5.8) tries to achieve this by alternating the GA and the FS methods. During the search of the representative subset a GA will quickly prune the design space of potential scenario subsets, whereas the FS will thoroughly search the small neighborhood around the high quality scenario subsets that are found by the GA. The tricky point is the moment of alternation. When one of the methods starts to converge, the other method should be activated.

At first sight, the feature selection may be interpreted as a custom variation operator for the GA, but this is absolutely not the case. Both the GA and the FS will keep state over time and, thus, if the same subset is sent to the FS more than once the oscillating search will be continued where it stopped in the previous invocation.

As the GA keeps a population of scenario subsets and the FS only works on a single scenario subset, there must be determined which scenario subset from the GA population is sent to the FS selection method. The most obvious method is to send the most representative subset from the GA to the FS. This can, however, not be done indefinitely. If the same subset is sent to the FS too often, the hybrid will be susceptible again for getting stuck in local optima as all the effort of the FS will be spent on the same subset. Therefore, the amount of effort spent by the FS to improve a single scenario subset is limited. If the FS has spent sufficient time on the same scenario subset (this time can be spread over multiple invocations of the FS), the subset is done and it will not be sent to the FS anymore. So, the subset is only sent if it is "undone": the size of the swing in the oscillating search does not exceed a predefined maximal margin. This margin is chosen in such a way that the computational overhead of a single swing is still acceptable.

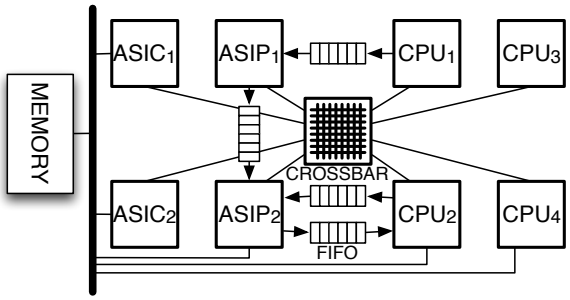


Figure 5.9: *The used MPSoC architecture components used for the experiments*

5.4 Experiments

Three different fitness prediction techniques have been introduced in this chapter: *Genetic Algorithm (GA)*, *Feature Selection (FS)* and a *Hybrid Approach (HYB)*. In this section, we will preform some experiments to compare the different techniques and to analyze their behavior. The multi-application workload that is used in this set of experiments is generated stochastically using a tool based on [51]. As discussed in the previous chapter, this gives us full control on the behavior of the application and the diversity within the corresponding scenario database by specifying constraints on the amount of computation and communication per scenario. Our used multi-application workload has 4607 scenarios that describe the behavior of 10 applications with a total of 58 processes and 75 communication channels.

The multi-application workload has to be mapped on the architecture that is shown in Figure 5.9. For the processes, there are eight potential processors: four processors are able to execute all the different processes (CPU), the two ASIPs are supporting 11 out of the 58 processes and the two ASICs can only run 4 out of the 58 processes. The ASICs and ASIPs are more specialized than the CPUs and can, therefore, run the processes faster, but the result of this gain is that not all the processes can be mapped on it. Looking back to our mapping procedure (Equation 5.3 to be precise), a valid allocation requires at least one of CPUs. For communication, there are multiple options: a crossbar with buffers to temporarily store messages [30], a memory connected by a shared bus and four dedicated FIFO communication channels. Although each of the architectural components has a specific cost, we have chosen not to specify a maximal area for the set of allocation components (Equation 5.2), but to treat the cost of the embedded system as an additional objective.

For this MPSoC design problem, it is clearly infeasible to exhaustively search for the Pareto front of non-dominated mappings. Not only are there $6.8 * 10^{58}$ possible mappings, but also the number of 4607 scenarios is too large to quickly evaluate the real fitness of each separate mapping. Therefore, it is crucial to use scenario-based DSE to obtain different Pareto fronts. To be able to obtain the results of multiple experiments quickly, all the experiments were performed on a dedicated node (e.g.,

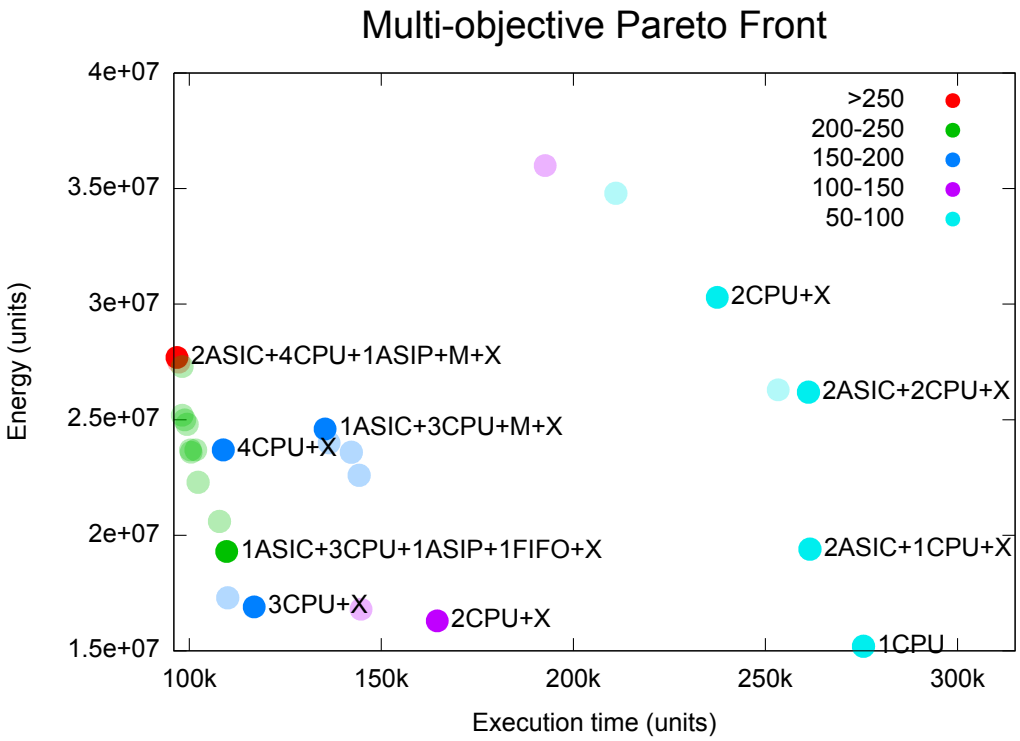


Figure 5.10: *An example Pareto front for the stochastic multi-application workload for three objectives: time (X-axis), energy (Y-axis) and the cost (point color). For the non-transparent points the allocated components are given. The potential component types are: ASIC, ASIP, CPU, memory (M) and crossbar (X).*

no other user processes are running) of the LISA cluster [42] that is part of the Dutch SARA Computing and Networking Services. The selected nodes for our experiments contain two Intel quad-core Xeon L5520 processors with a total of eight cores running at 2.26 GHz.

Next to the cost of the embedded system, execution time and the energy consumption are also objectives for the optimized mapping that we are looking for. The execution time and the energy consumption are determined with the use of the Sesame simulator. As was discussed in Section 2.1.5, the energy is estimated via an activity-based power model that uses the busy and idle times of each individual architectural component. As a result of these three objectives, the scenario-based DSE will end up with a Pareto front that gives a trade-off between time, energy and cost.

Figure 5.10 shows an example of such a Pareto front. It shows the time and energy objectives on the X and Y-axis and the cost objective is shown using color. Each color stands for a different cost range, where light blue refers to the cheapest

mappings and red to the most expensive mappings. The cheap mappings can use only a small set of allocated components, like the single core solution that is the most energy efficient solution. This can easily be explained: the CPU is fully utilized and no energy is spent on communication between different components. Obviously, adding more processors improves the execution time due to the ability to spread the computational load over the different processors. Not surprisingly, the fastest solution uses almost all of the available architectural resources: two ASICs, four CPUs, one ASIP, the shared memory and the crossbar. For most of the mappings, the fast crossbar switch is the most obvious choice for communication, but the fastest solution uses both a shared memory and a crossbar. It is a quite expensive solution, but it increases the available communication bandwidth. When looking at low-energy mappings, the tendency is non-trivial: in general adding more resources increases the peak power of the system, but as the execution time may become smaller the energy potentially drops. This is perfectly shown by the 1CPU, 2CPU+X and the 2CPU+2ASIC+X mappings. The mapping with only two CPUs has one of the highest energy consumptions. Other solutions as the 3CPU+X mapping has a better exploitation of the processors that leads to a lower execution time. The gain in execution time is large enough to compensate for the higher power usage of the additional processor. Due to communicational overhead, adding two ASICs in the 2CPU+2ASIC+X mapping decreases the performance with respect to the 2CPU+X mapping, but the tasks are executed in a more energy efficient manner.

In the following subsections a couple of experiments will be discussed. In general these will focus on two aspects: 1) how does the subset selection behave and 2) what is the effect of the subset selection on the DSE itself. Section 5.4.1 will compare the different selection techniques in isolation: what is the effect of the subset size on its quality, how does the misclassification look like and what is the success rate of each of the methods? As the subset selector is analyzed in isolation, the design explorer is not taken into account. This design explorer will be taken into account in the next experiment in Section 5.4.2 where the effect of the subset selection method and the subset size on the DSE results of the design explorer is investigated. After analyzing the subset selection in isolation and its effect on the DSE outcome, the quality of the subset during the DSE is analyzed in Section 5.4.3. The last subsection in this section compares the required effort for the different subset selection techniques to identify mappings that match certain requirements.

5.4.1 Subset Selection Efficiency

In the first experiment of this chapter, we will focus on the subset selector. This means that from the scenario-based DSE (as shown in Figure 5.1), only the subset selector is taken into account and not the design explorer. The only way to achieve this is to statically fill the trainer of the subset selector and to keep this trainer fixed over time. Two different sets of training mappings will be used: the first contains 518 mappings, whereas the second contains 498 mappings. The difference in effort

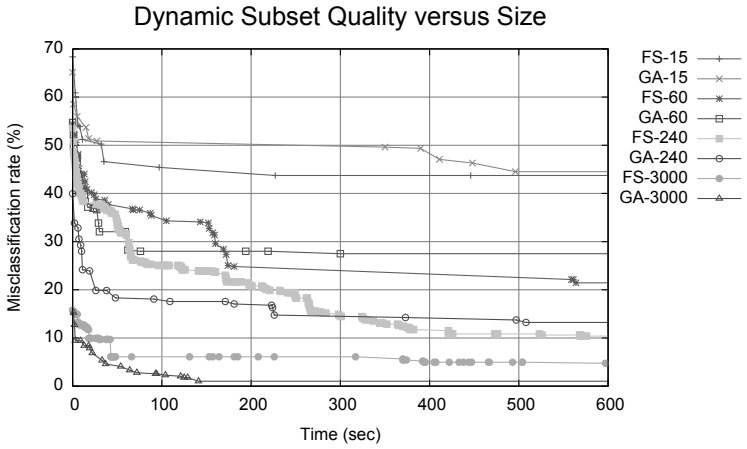


Figure 5.11: *The relation between quality and size for the GA and FS subset selection approach over time.*

that is required to find a representative subset for these trainers is shown in Figure 5.13c, which will be discussed later.

Quality versus size

One of the parameters affecting the quality of a representative subset of scenarios is its size. When the allowed subset size is larger the subset selector has a higher potential to obtain a representative subset. To quantify the effect of the subset size our first experiment compares the quality of different sized scenario subsets over time. Additionally, this experiment is also used to highlight the differences between the GA and the FS approach for subset selection. Figure 5.11 shows the results of this experiment, where the horizontal axis shows the elapsed wall clock time and the vertical axis shows the quality using the misclassification rate. For this experiment the first trainer of 518 training mappings is used. In contrast to our earlier definition of Equation 5.14, the misclassification rate is based on Goldberg's ranking (see Figure 5.5c) and not on the boolean ranking. Although Goldberg's ranking may be somewhat deceiving, the differences between the GA and the FS approaches are more clear when Goldberg's ranking is used. The lowest rank for the boolean ranking and Goldberg's ranking are the same, but for higher ranks Goldberg's ranking has much more different ranks (boolean ranking ranks all the dominated individuals with the same rank). This increases potential number of ways that the trainer can be ranked. The parameters for the GA subset selection method are determined experimentally and are shown in Table 5.1a. For the FS approach no additional parameters are required.

As both approaches start with random subsets, the initial quality of the GA and FS approaches are quite similar. This initial quality is clearly dependent on the size

of the scenario subset: a subset of size 60 starts with a misclassification rate of 54 percent, whereas a subset of 15 scenarios misclassifies 68 percent of the individuals. During the startup phase of the subset search (the first 30 seconds), the GA based selector clearly has better subsets than the FS based selector due to the ability of the GA based selector to quickly explore the complete design space of potential scenario subsets. The modifications of the FS to the subsets are done more thoroughly and are subtle and, therefore, it takes somewhat more time to find a representative scenario subset. However, after sufficient time, the more thorough FS based selector starts to outperform the GA based selector. For a subset of 15 scenarios (where there are $6.6 * 10^{42}$ potential subsets) it takes only 30 seconds to outperform the GA based selector. When the number of scenarios in a subset increases, the number of potential scenario subsets also grows exponentially. Hence, the FS based selector requires 170 seconds to outperform the GA based selector for 60 scenarios in a subset and 300 seconds for 240 scenarios in a subset. On a further growth of the scenario subset size, the time for the FS based selector to outperform the GA based selector will only grow. Within 600 seconds, the FS based selector does not yet outperform the GA based selector for a scenario subset of 3000 scenarios. Still, it is most likely that eventually FS will outperform the GA based selector.

The startup phase of the search can be seen as a warm up period. As long as this warm up period is short enough, it does not necessarily negatively affect the scenario-based DSE. As the design explorer also has a warm up period, there is a phase where the differences between the mappings are still significantly large and, therefore, it is still relatively easy to discriminate between good and bad MPSoC mappings. Later in the DSE, the differences between the mappings become smaller and at that point it is important to have a good representative subset of scenarios. For the smaller subset sizes, the FS based selector will have a more representative scenario subset than the GA based selector after the warm-up period of the design explorer. This is not the case for the larger subset sizes, but for these cases the GA is required to quickly prune the design space of potential subsets. To automatically make these considerations, we have introduced the hybrid selector method.

Misclassification: how does it look like

Up to now, the term misclassification rate was used quite often, but it was not really shown what it looks like. In the experiment illustrated in Figure 5.12 an example is shown of the relationship between the real and the predicted dominance depths. In total, 1933 mappings are used to generate the density graph. These mappings do not overlap with the mappings that are used to train the scenario subset of 240 scenarios using the FS based subset selector. The horizontal and vertical axes show the predicted and real dominance depth. For each combination of a predicted and real dominance depth the density is given using different gray tones. The darker the region the more often the combination occurs.

Ideally, the diagonal (real dominance depth is equal to predicted dominance

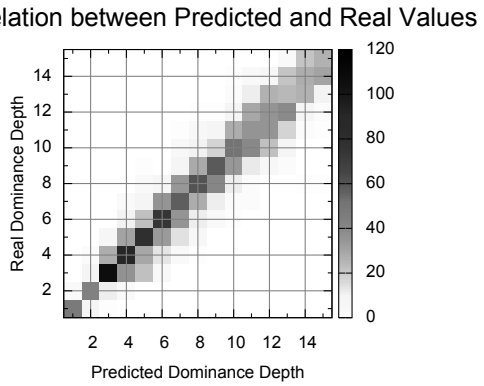
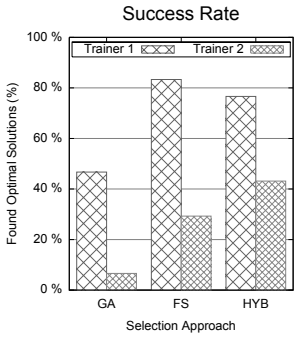


Figure 5.12: *A density graph showing the relationship between the predicted and the real dominance depth.*

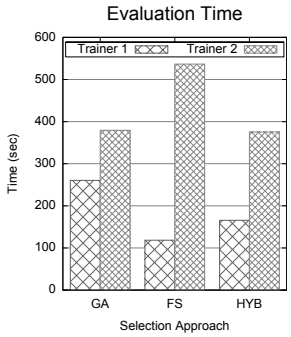
depth) should be gray and the rest of the density graph should be white. This is indeed the case for dominance depths 1 and 2. As the DSE is only interested in identifying the Pareto front of non-dominated mappings, the results for the real dominance depth of 1 are quite good. For the mappings with a real dominance depth of 1, 84 percent of the mappings have a correctly predicted dominance depth. The fraction of non-dominated mappings is hardly visible in the density graph (which is a good sign): 14 percent of the non-dominated mappings have a predicted dominance depth of 2. For higher dominance depths, however, the diagonal becomes wider (i.e., the error in the predicted dominance depth becomes larger) and the uncertainty about the dominance depth grows. As discussed earlier in this chapter, a behavior like this is the motivation for using the boolean ranking where the dominance depth of two and higher all have the same rank.

Comparing the different methods

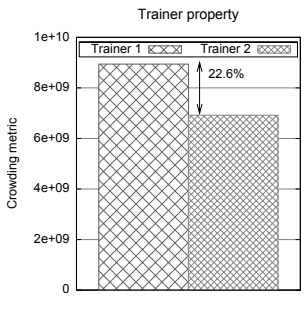
After illustrating the need of boolean ranking (it exactly corresponds to the aim of the scenario-based DSE: identify the Pareto front) and the hybrid subset selection method (the GA reduces the warm-up period of the subset selection procedure, whereas the FS closely explores the local neighborhood of a subset), the next step is to quantify the difference between the three different subset selection methods. As in the previous experiments, the subset selector is run in isolation without the use of the design explorer. Therefore, instead of dynamically updating the trainer, a fixed trainer was used. The trainers, which are shown in Figure 5.13c have 518 and 498 training mappings and they differ in their average crowding distance [12] between the training mappings. On average, the training mappings in the second trainer are 22.6 percent closer to each other than the training mappings of the first



(a) *The success rate for finding the optimal subset.*



(b) *Required time to find the optimal subset.*



(c) *The differences between the two trainers.*

Figure 5.13: *A comparison of the three approaches for the subset selector: a genetic algorithm, a feature selection algorithm and a hybrid approach of both.*

trainer. Consequently, for the second trainer it is harder to predict the ranking of the different training mappings.

For each subset selection methods, 180 different runs are performed where the size of the subset is varied between 20 and 200 (i.e., 0.4% to 4.3% of the total set of scenarios). The subset of 20 scenarios is the smallest subset size for which an optimal subset is found that correctly predicts the ranking for the trainer. For the subset of 200 scenarios, on the other hand, the subset is not yet too large to prevent a quick identification of the optimal subset.

The aim of the experiment is to search for an optimal subset and find it within 15 minutes. In case an optimal subset is found, the search has succeeded. Figure 5.13 shows the results of this experiment. First, Figure 5.13a shows the average success rate for each subset selection method. Recall that success means that a scenario subset is found that is able to perfectly predict the Pareto front. This scenario subset does not need to be unique; there may be more scenario subsets that are capable of performing a perfect prediction. In Figure 5.13b the average evaluation time is shown for each of the cases that the subset selector method found an optimal scenario subset. The results clearly show that it is harder to find an optimal scenario subset for the second trainer: the success rate for all the selection methods is lower and the average evaluation time in case of success is higher.

When comparing the GA and the FS approaches, the FS approach is clearly better with respect to success rate. For the evaluation time it depends on the trainer: in case of the first (and simpler) trainer, the FS approach is able to find a solution more quickly. In this case, relatively few modifications are required to a random initial subset to converge it into an optimal subset. The optimal subset for the second trainer is harder to find and, therefore, requires more modifications to transform an initial subset into a representative subset. As the GA is able to quickly prune the

design space of potential scenario subsets, the GA can potentially find the optimal scenario subsets much earlier than the FS approach. However, the success rate of the GA is still more than four times as low as the FS method (6.6% versus 29%).

In order to combine the strengths of both the GA and FS, the hybrid approach is introduced. For the first trainer, the gain of the hybrid approach seems to be small. It has a slightly lower success rate than FS (77% versus 83%) and it also requires some additional time to find the optimal solutions. This result is not really surprising, the optimal subset of the first trainer is easier to find and, therefore, the pool of scenario subsets that is maintained in the hybrid approach only results in additional overhead. Instead of focusing on fully optimizing a single scenario subset, effort is spent to keep multiple subsets simultaneously. This effort, however, is well spent when it becomes harder to find the optimal subset of scenarios. For the second trainer, the hybrid approach has a success rate of 43 percent, where the FS approach only has a success rate of 29 percent. Due to the quick pruning by the GA part of the hybrid approach, the average evaluation time of the hybrid approach is the fastest in case of the second trainer.

It is not implied that in the remaining 57 percent of the cases the subset cannot be found: after some additional time the GA within the hybrid approach will likely steer the search to the correct direction. Whenever a near-optimal scenario subset is found, the hybrid approach can (in contrast to the GA approach) use its FS to fully search the neighborhood of the near-optimal scenario subset. One could argue that a FS method can, just as the hybrid approach, can be extended to circumvent local optima. As an example, the FS algorithm can be restarted with different initial subsets. However, if we would apply this technique, the resulting FS approach would be slower than the hybrid method. Even if we would start the FS approach only twice, it would be clearly slower than the hybrid approach.

A final advantage of the GA and the hybrid methods are that they do not have a single good subset of scenarios. At all times, they will have a population full of good scenario subsets. This is useful when the trainer is updated since it is more likely that there is already a scenario subset in the population that is representative for that specific trainer. The FS approach has only one subset and this leads to a slower adaption to a changing trainer.

5.4.2 The Effect of the Subset Size on the DSE

Up to now, we have only looked at the subset selector in isolation. As the subset selector is only meant to support the fitness prediction of the design explorer, the next step is to investigate the effect of the subset selection techniques on the ability of the design explorer to identify good MPSoC mappings. Since the term "good mappings" sounds rather vague, we have chosen to search mappings that, given a maximal cost, have an execution time that is as low as possible. This is also easier

Table 5.1: *Experimental settings*

Population size	20
Offspring size	40
$P_{crossover}$	0.7
P_{mutate}	0.02

(a) *Subset GA*

Population size	100
Offspring size	200
$P_{crossover}$	0.9
P_{mutate}	0.01

(b) *Design Explorer*

to visualize. For experiment A the objective is:

$$\begin{aligned} &\underset{m}{\text{minimize}} \quad \text{time}(m) \\ &\text{subject to } \text{cost}(m) \leq 100 \end{aligned}$$

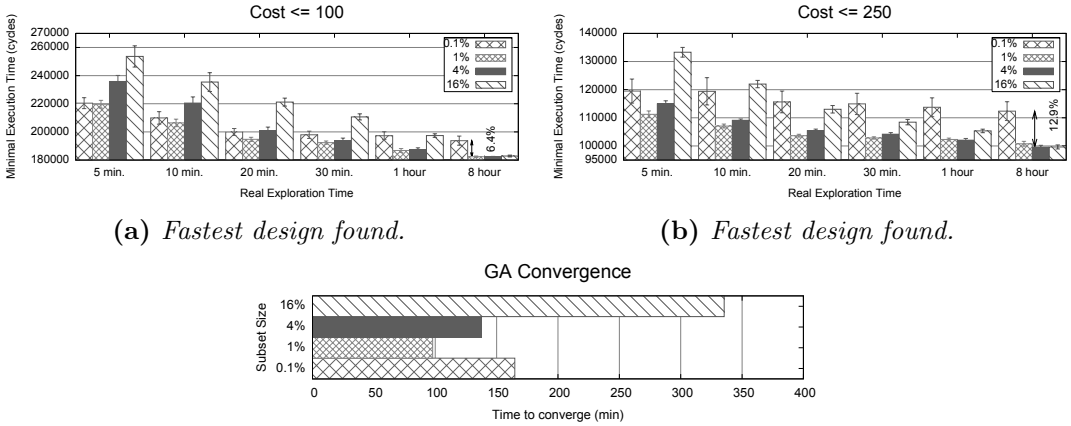
Experiment A has a maximal cost of 100. In the example Pareto front given earlier (Figure 5.10), there can be seen that for this cost limitation the number of allocated CPUs is at most two. For communication, only the crossbar or the memory can be used (not both). The FIFO is not available in experiment A. Experiment B allows for more resources that are used to execute the multi-application workload as fast as possible. With a maximal cost of 250 almost all components can be allocated:

$$\begin{aligned} &\underset{m}{\text{minimize}} \quad \text{time}(m) \\ &\text{subject to } \text{cost}(m) \leq 250 \end{aligned}$$

For this experiment, we performed a DSE of eight hours with the hybrid subset selector approach. From the eight available cores on the dedicated machine, two cores were assigned to the subset selector and six cores were assigned to the design explorer. The GA parameters are given in Table 5.1 and the experiments are done for four subset sizes: 0.1, 1, 4 and 16 percent of the total number of application scenarios. Each of the runs is repeated nine times to take the stochastic nature of the GA into account. At six points in wall-clock time the current estimated Pareto front is logged, which is exhaustively evaluated after the DSE to obtain the real fitness values of the non-dominated MPSoC mappings.

Figure 5.14 shows the real execution (cycle) time of the fastest mappings that are found by the DSE for each of the time steps. The subset sizes are shown with separate bars from small to large. In general, there are two trends when the number of scenarios in the subset is increased:

- **Accuracy:** A larger subset has will most likely lead to a more accurate fitness prediction in the design explorer. As a result of this increased accuracy, better decisions are made with respect to the discrimination between good and bad



(a) *Fastest design found.* **(b)** *Fastest design found.*

(c) *Time to converge.*

Figure 5.14: *The effect of the subset size on the perceived result of the DSE.*

MPSoC mappings. Therefore, the GA in the design explorer will require fewer generations to identify the mappings with a low execution time.

- **Overhead:** A larger subset may result in fewer generations to identify good MPSoC mappings, but the wall clock time of a single generation will also increase. The higher the number of scenarios in the subset, the longer it takes to evaluate a single mapping. This leads to a less effective evolutionary process as fewer generations are performed within the same time frame.

For a short period (i.e., five minutes), the overhead effect is the most significant in case of the 1%, 4% and the 16% scenario subsets. A larger scenario subset clearly leads to an increase of the minimal execution time in the experiment of Figure 5.14a and the experiment of Figure 5.14b. The only exception is the 0.1% scenario subset which is too inaccurate to correctly predict the fitness of a mapping. Hence, its best solution after five minutes is already worse than the one from the 1% subset. For the 16% scenario subset the overhead effect is still too large and as a result it still provide worse DSE results than the 0.1% subset.

The (in)accuracy effect of the 0.1% subset can be observed both by the low quality of the final outcome and the high diversity over the different experiments. After eight hours, the final result is 6.4 percent (Figure 5.14a) to 12.9 percent (Figure 5.14b) slower than the most optimal mappings found by the larger scenario subsets. Next, the standard deviation (as shown by the error lines on top of the bar) of the 0.1% scenario subset is also relatively large. On top of that, the standard deviations remain the same over time, whereas the large subsets have a decreasing standard deviation over time. This clearly shows that the larger subsets become more accurate over time, in contrast to the small and imprecise 0.1% scenario subset.

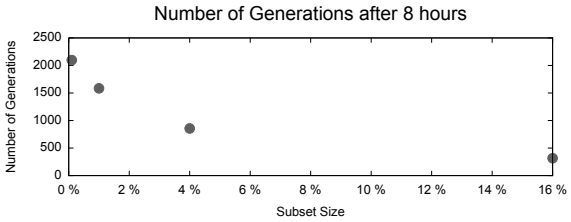


Figure 5.15: *The number of generations in* 8 *hours.*

Figure 5.15 makes the overhead effect more clear. The 1% subset size is capable of doing 1585 generations in eight hours, whereas the 16% subset can only do 316 generations. Obviously, the number of generations indeed decreases with a larger scenario subset. Less obvious is the non-linear relation. This is because a lower evaluation time per generation leads to a larger sequential portion within the design explorer. Amdahl's law teaches us that this leads to a less efficient usage of the multi-core machine, as the selection and the variation of the GA always needs to be done sequentially.

After a longer period of time, the overhead effect is not relevant anymore. At the end of the 8 hours the 1%, 4% and 16% subset sizes have a similar minimal execution time as a final searching result. This is also the case for the standard deviation between the different repetitive experiments. As long as the subset is large enough to be accurate, the overhead only determines the convergence time of the DSE. Figure 5.14c shows the convergence (i.e., how long does it take until the DSE is within 1% of the final result) of experiment A is given. The convergence time of the 1%, 4% and 16% scenario subset sizes increases monotonically. Only, the inaccurate 0.1% subset is an exception to this rule as its convergence time is larger than the 1% scenario subset. Additionally, the slower convergence of the 16% scenario subset can also be seen by the steady decrease of the minimal execution time of the found solutions over the different time steps in Figure 5.14a.

To conclude, we can speak of an *accuracy threshold*. Once the accuracy of a subset is above a certain threshold, the final GA results of the design explorer are not significantly affected by the subset size. Above the accuracy threshold, a larger scenario subset size will only negatively affect the convergence time.

5.4.3 Subset Quality during DSE

In the experiment of Section 5.4.1 we have shown the quality of a scenario subset of a fixed trainer. This experiment leads to the conclusion that the hybrid subset selector was better in identifying optimal subsets than the other approaches (especially when the trainer becomes more crowded), but it does not tell us what happens with the subset quality when the trainer dynamically changes during the DSE. This is what we try to show with the experiment in this section, where we monitor the subset

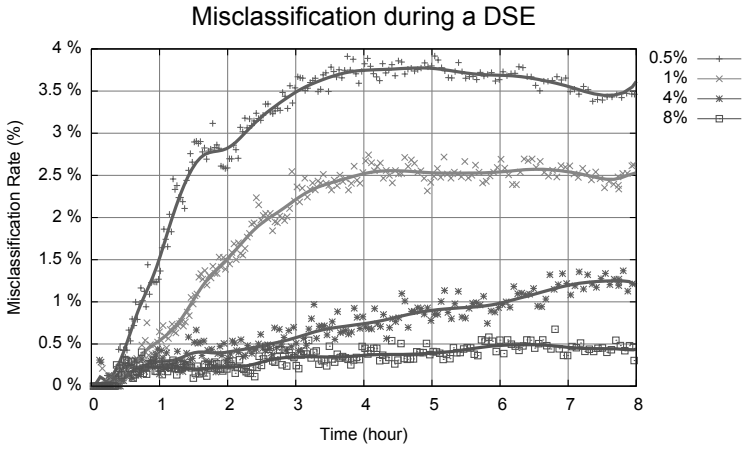


Figure 5.16: *The subset quality during the DSE with respect to the misclassification rate to the current training sets. Both the average measurements (the points) and the smoothed trend (the line) is shown.*

quality over time during a scenario-based DSE. For this purpose, we have taken the same experimental settings as in the previous subsection (see Table 5.1) and we logged the quality of the most representative scenario subset just before each update of the trainer.

The results are shown in Figure 5.16. Just as before, an eight-hour DSE is performed where each experiment is repeated nine times. Based on the conclusions of the previous experiment, we have chosen for the scenario subset sizes of 0.5%, 1%, 4% and 8%. The points show the average misclassification rate, whereas the line is an approximation of the data points with a Bézier curve.

In this experiment there are two opposing forces. At first, the subset selector dynamically improves the scenario subset over time. As a consequence, the quality will gradually become better. At the same time, however, the trainer is changed dynamically by adding mappings originating from the design explorer. After such a change it is harder to still classify the mappings in the trainer correctly and the subset scenario quality decreases.

Not surprisingly, a larger scenario subset size results to a lower misclassification rate. Initially, the misclassification may be zero, but at this point the trainer is still empty. In the first hour, the trainer is slowly filled and transforms from an easily predicted set of training mappings into a set of training mappings after which the Pareto ranks are hard to predict. As a result, the misclassification rate of the smaller subsets of 0.5% and 1% quickly increase to approximately 3.5 and 2.5 percent. After a four-hour search, the trainer starts to stabilize. Hence, the misclassification rate of the scenario subsets becomes more stable. The larger the scenario subsets are, the less sensitive they are to a changing trainer. For the 8% scenario subset, the

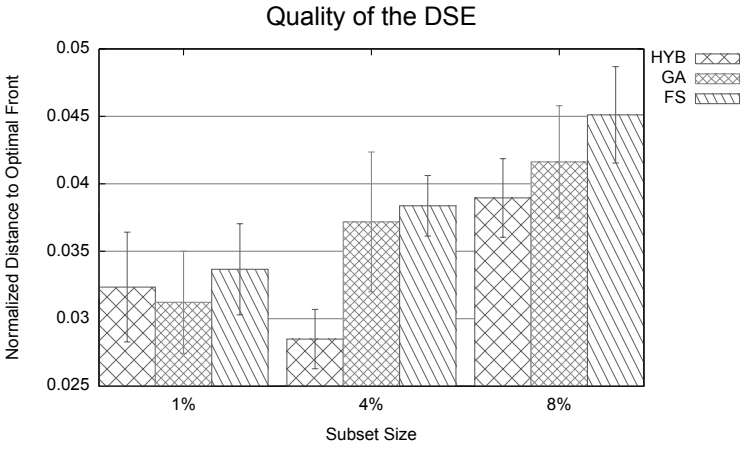


Figure 5.17: *The average quality of the final DSE result for the different subset selection techniques.*

misclassification rate remains around 0.5 percent irrespective of the warming up of the training set of mappings. This is mainly due to a better accuracy of larger scenario subsets.

Hence, the trainer requires a short warm up period before the dynamic quality of the subset stabilizes. Initially, the average crowding in the trainer is very low and, without any exception, all of the scenario subsets provide a perfect prediction of the Pareto ranks of the trainer. Over time, the design explorer starts to converge and the trainer becomes more crowded (i.e., the mappings become more similar). As a result, it becomes harder to correctly predict the Pareto ranks. This improved training set may hurt the perceived misclassification rate of the scenario subset in the subset selector, but it is beneficial for the design explorer that continuously encounters unknown mappings. Therefore, the decrease in perceived misclassification rate does not imply that the scenario subset is worse. It is more a sign of a more challenging trainer. This is also seen by the stabilizing misclassification rate after a certain period. At this point in time, the scenario subset is good enough to also predict the fitness of the new mappings that are added to the trainer.

5.4.4 Selection Methods and DSE Efficiency

Our final experiment will focus on the different subset selection techniques and their effect on the efficiency of the DSE. For this purpose, we compare the generational distance (see Section 2.3.2) of the final Pareto front to the optimal front. As the optimal front is not known, it is estimated by combining the Pareto fronts of all the experiments that we have performed. Figure 5.17 shows the normalized generational distance after 100 minutes of exploration time. For this purpose, all the objectives time, energy and cost are normalized to a range from 0 to 1. The experiment is

averaged over nine runs and done for three different subset sizes: 1%, 4% and 8%.

Recall from Section 5.4.2 that an increasing subset size results in two effects: 1) a higher accuracy and 2) a lower convergence rate. The lower convergence rate manifests itself for the GA and FS with a higher generational distance to the optimal Pareto front (i.e., the Pareto front is worse) after 100 minutes when varying the size of the scenario subset. The hybrid approach, however, performs best with a subset size of 4%. This subset is more accurate than the 1% subset. The slower convergence becomes only visible at the 8% subset.

When comparing all the subset selection methods, the hybrid approach provides the best results. The only exception is the 1% scenario subset where the design space of potential scenario subsets is still small enough for the GA based selector to quickly identify a good representative subset. When the design space of potential subsets becomes larger (i.e., the scenario subset is larger), the hybrid method can exploit the benefits of both the GA and the FS based selection methods. Still, a larger subset size does not necessarily result in a larger gain of the hybrid approach (with respect to the normalized distances of the other approaches). A larger subset size is also potentially more accurate than a smaller one. Therefore, a larger subset can also ease the search to a scenario subset. This can be observed when comparing the gain of the hybrid approach at a 4% subset size and a 8% subset size.

5.5 Conclusion

Scenario-based DSE efficiently explores the mapping of dynamic multi-application workloads onto an MPSoC. Crucial for the efficiency is the subset selector that dynamically selects the fitness predictor for the design explorer. This fitness predictor is a subset of application scenarios that is used by the design explorer to quickly identify the non-dominated set of MPSoC mappings. In this chapter, we have given a detailed description of how the representativeness of a scenario subset can be calculated and which techniques can be used to select the fitness predictor (i.e., the subset of scenarios).

The three different fitness prediction techniques that were presented are: 1) a genetic algorithm, 2) a feature selection algorithm and 3) a hybrid method combining the two aforementioned approaches. A genetic algorithm is capable of quickly exploring the space of potential scenario subsets, but due to its stochastic nature it is susceptible for missing the optimal scenario subsets. This is not the case with the feature selection algorithm as it more systematically explores the local neighborhood of a scenario subset. Unfortunately, this approach is relatively slow and can suffer from local optima. The solution is to combine these approaches in the hybrid approach, leading to a fitness prediction technique that can quickly prune the design space, can thoroughly search the local neighborhood of scenario subsets and is less susceptible to local optima. Experiments showed that the hybrid approach provides similar or better results than the other fitness prediction techniques.

Additionally, this chapter investigated the effect of the scenario subset size during the DSE. Generally, there is an accuracy / overhead trade-off. Potentially, a larger the subset results in a more accurate fitness prediction. A more accurate fitness prediction may reduce the number of required generations for the DSE. However, the time a generation takes is affected by the overhead effect. This effect states that the larger a subset, the longer a single mapping evaluation takes, and the longer it takes to perform a single generation. There is a limited interaction between the two effects. If the accuracy is below a certain threshold, the DSE will not be able to find a representative subset of scenarios. As a result, the scenario-based DSE will provide poor results. Above the accuracy threshold, the overhead effect will only negatively impact the time before the DSE is converged. Still, as long as the subset size is above the accuracy threshold, the scenario-based DSE will result in a non-dominated set of MPSoC mappings for a dynamic multi-application workload.

Part III

Architecture Scenarios

6 Sesame Automated Fault-tolerant Explorer

This chapter is based on:

- P. van Stralen and A. D. Pimentel. 'A SAFE approach towards early design space exploration of fault-tolerant multimedia MPSoCs'. In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '12)*. Tampere, Finland, Oct. 2012, pp. 393–402

This thesis discusses the application of scenarios during the early embedded system design. In Part II of this thesis, we have looked at application scenarios that model the dynamism in the functional behavior of a multi-application workload. These application scenarios were utilized in order to obtain a static mapping of the multi-application workload onto the MPSoC such that the average behavior was as good as possible. The applications, however, are not the only dynamic part of the MPSoC. Another dynamic part is the MPSoC architecture where, for example, the architecture may be susceptible to faults or it allows for a reconfigurable set-up. A result of this dynamism in the architecture is that the non-functional properties (like performance and power) of an application mapping may change over time.

In this thesis, we will use the occurrences of faults in the architecture as an example of the dynamism in the architecture. Due to these faults, the output of applications that are mapped onto the architecture is not always correct. Take, for example, the binary sum $0001 + 0001$. Normally, the solution of this sum would be 0010, but upon the occurrence of a fault one of the bits may flip. Therefore, on the flipping of the rightmost bit, the result can suddenly be 0011 instead of 0010. The potential presence of faults has a major effect on the embedded system design. As none of the outputs can be fully trusted, the reliability must be part of the design.

Next to the traditional objectives (e.g., performance, power and cost), reliability can be seen as an additional objective for an MPSoC design. To be precise, the reliability of an embedded system is the ability of the system to cope with errors. A higher reliability implies that the system has a greater ability to cope with the faults that are occurring in the architecture. These faults in the architecture can be of two

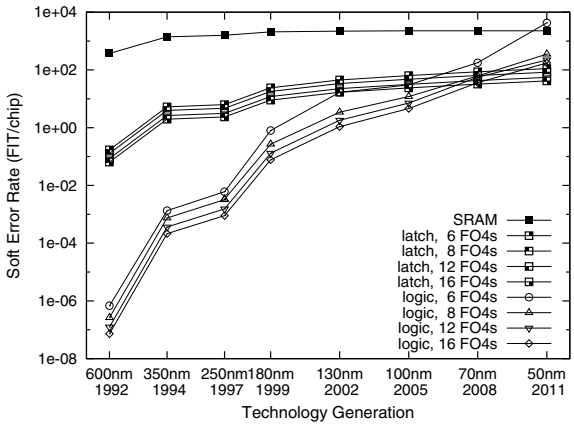


Figure 6.1: *The effect of of technology trends on the soft error rate of combinatorial logic [66].*

types: hard and soft errors. Hard errors (also called permanent faults) are permanent and once they have occurred, the affected architecture component becomes unusable. Soft errors (also called transient faults), however, are only causing temporal malfunction of the system. Examples of soft errors are *Single Upset Event (SUE)* [66] and *Negative Bias Temperature Instability (NBTI)* [1]. A SUE is a soft error where the electrical system is influenced by high-energy neutrons originating from cosmic rays that collide with particles in the atmosphere. NBTI, on the other hand, is caused by a degradation of the electrical circuit during its lifetime. Generally speaking, a soft error is a fault in one of the circuits (like processors, memory, communication, etc.) due to electrical noise or external radiation.

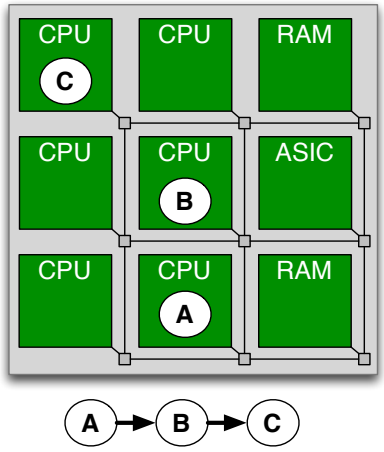
There are important difference between faults, failures and errors. At first, a fault is something like an incorrect step or data item in a process that leads to an unintended or unanticipated execution result of the process. A fault is a weakness in the process that might lead to a failure. This completely depends on the situation and how tolerant the application is to faults. Secondly, a failure, on the other hand, is the inability of the system to perform its task within the required performance requirements. This is undesirable and, unlike a fault, it unavoidably negatively affects the *Quality-of-Service (QoS)* of the system. Finally, an error is a discrepancy between the real and the computed output of a process. An error is caused by a fault and it may potentially be propagated in the process to result into a failure. Therefore, soft errors are, by definition, noticed.

It used to be the case that soft errors were not an issue for normal embedded systems, but only for avionics (electronic systems that are used in aircrafts, satellites and spacecrafts). With the improvements in technology, however, it has become also an issue on the ground level. This is illustrated in the graph of Figure 6.1 that shows the modeled number of faults for the different technology scales [66, 65]. On the

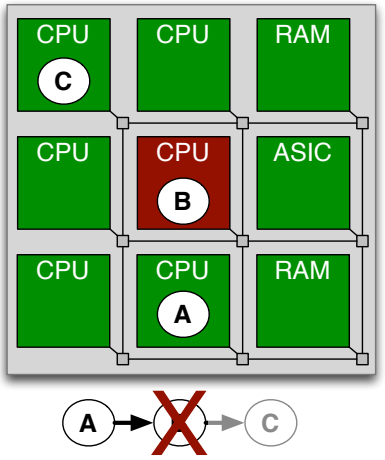
horizontal axis the technology is shown, whereas the vertical axis shows the soft error rate. This soft error rate is described using the number of *Failures In Time (FIT)*. In a technology of 600nm a single SRAM chip (the memory part of a system) has roughly a FIT of 10^3 (i.e., 3000 faults per 1 billion hours or a mean time to failure of 10^6), whereas the logic (the computational part of the system) only has a FIT of 10^{-6}. Therefore, for this technology scale it is quite obvious to make the memory storage reliable, but to ignore the soft errors that could occur during computation. However, with the decreasing technology scales, this relationship between memory and computation changed. When alternating the technology scales, the FIT rate is affected by a trade-off between a lower critical charge for the high-energy neutrons and the smaller device size. The critical charge is the minimal charge a neutron must have to be able to trigger a soft error. A smaller technology scale leads to a lower critical charge and, therefore, the FIT rate of the devices increase. On the other hand, a smaller technology scale leads to a smaller area of the device. As a result, the FIT rate decreases. For a memory the effect on the critical charge is relatively small and, thus, the memory benefits from the smaller area: from a technology scale of 600nm to 50nm the FIT rate stabilizes around 10^{-3}. In case of a logic circuit, however, the critical charge drops much faster: the FIT rate grows from 10^{-6} to a FIT rate between 10^2 and 10^3. As a consequence, the fault rate in the logic computations becomes higher than the fault rate in the memory. Therefore, reliability becomes a first class citizen of the embedded system design and should definitely be taken into account when an embedded system is designed.

Figure 6.2 shows an example of how a single fault in the architecture can affect the output of a mapped application. The left column shows a situation where no fault is present and all the architectural components work flawlessly. In the right column, however, one of the CPUs is affected by a fault. Depending on the way the application is implemented, this may have a significant effect on the correctness of the behavior of the application. The first row shows an unmodified application. It is a pipeline of three processes (A, B and C) that operate on a single stream of data. In case the architecture is free of any faults, the output will be correct and in time (Figure 6.2a). The situation in the second architecture scenario is somewhat different (Figure 6.2b): the CPU on which process B is mapped is affected by a fault. A CPU that is in a faulty state cannot be trusted anymore. Therefore, the outcome of process B may be: 1) correct, 2) incorrect, 3) it may arrive later than usual or it may even never arrive at all. Depending on the situation, process C gets unreliable data or no data at all. As a result, the complete application becomes unreliable since it is not known if the output stream is correct or not.

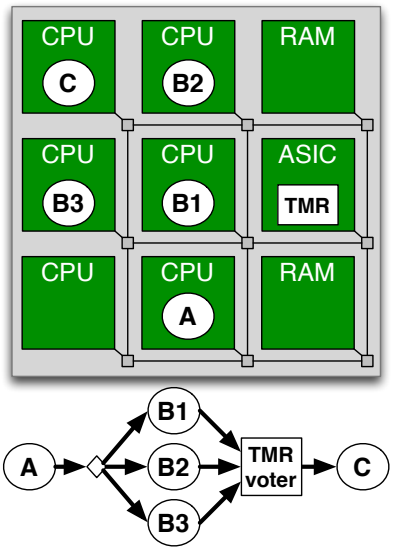
In order to make an embedded system reliable, both hardware- and software-based techniques can be used to improve the reliability. Hardware-based techniques design an architecture component in such a way that it is more resilient to soft errors by, for example, using replication of architecture components [20]. The software-based techniques apply similar techniques purely in software. An example of a



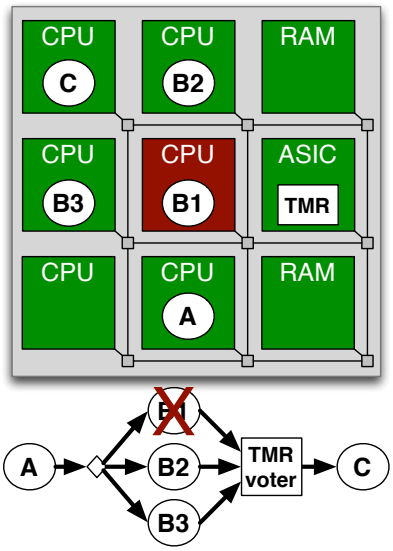
(a) *Normal, without faults.*



(b) *Normal, single fault.*



(c) *TMR, without faults.*



(d) *TMR, single fault.*

Figure 6.2: *An example of how a single fault affects a normal application and a fault-tolerant application using triple modular redundancy (TMR).*

software-based technique is active redundancy in space or time. Active redundancy in space uses different resources to run the same task. After running the tasks, their outputs are compared using a majority voter. In case the value of one of the outputs deviates from the value of the majority of the outputs, there is assumed that the majority has the correct output value. In Figure 6.2c an example of active redundancy in space is given: *Triple Modular Redundancy (TMR)*. TMR uses three different replicas to execute the same task. When there are no faults, the behavior of the application is almost similar as the normal application. The only difference is the overhead of the majority voter in the TMR. In contrast to the normal application, however, it can tolerate a single fault. This can be seen in Figure 6.2d. The processor on which the first replica of process B is mapped has a fault. Since the other two replicas are mapped onto correctly functioning processors, the TMR voter observes the same outputs for processes B2 and B3. Assuming that the processes have a deterministic behavior, the voter observes a majority for the correct output and this output can be streamed to process C. Active redundancy in time is similar, only in this case the replicas are running on the same architectural resources and are scheduled in a sequential fashion.

Implementing software-based reliability techniques has its pros and cons. As discussed earlier, a technique like TMR already adds some voting overhead to the execution time of a single application. Apart from that, it also requires more architectural resources than a normal application without replicas. Therefore, the non-functional properties (like execution time, energy and cost) of a normal mapping are highly affected by software-based reliability techniques to make an embedded system reliable. A software-based reliability technique may boost the reliability of the system, but other metrics like performance and power may be affected negatively. Therefore, the reliability of a system must be taken into account in the early phases of the design space exploration. It makes no sense to apply software-based techniques after the mapping has been optimized (hierarchical design). The non-functional properties of such an optimized mapping will be invalidated and better mappings may be identified when the reliable application is taken into account from the start on.

In this chapter, we introduce *Sesame Automated Fault-tolerant Explorer (SAFE)*. SAFE is an extension of Sesame to explicitly model the appliance of software-based reliability techniques to MPSoC based embedded systems. The simulation of SAFE will not only results in the evaluation of traditional non-functional properties of an embedded system like execution time, energy and cost, but also metrics like frame miss rate. In Section 6.1, a detailed description is given of the application model that is required as an input for SAFE. Next, Section 6.2 discusses the way fault tolerance is modeled within the KPN, followed by Section 6.3 that introduces the fault-tolerant mapping. The extensions to the simulation model of Sesame are given in Section 6.4. Finally, the chapter is concluded by a set of experiments, related work and a short conclusion.

6.1 IO Modeling

Up to now, the MPSoC designs were optimized to be as fast as possible. However, regularly the perceived output by the user is of more importance than the total running time of an application. In case of a multimedia application, like a video player for example, the user wants a timely display of the frames. This means not too late, but also not too soon. Take, for example, a frame buffer of a computer monitor. A frame buffer of a computer monitor should not be overwritten before the frame is shown. For the fault-tolerant optimization it is even more important to be able to reason about timely delivery of frames. While searching the design space of fault-tolerant embedded systems, we are actually trying to find out how many and which reliability techniques can be applied without significantly affecting the user's experience. Therefore, in order to be able to model frames and to capture frame rate in scenario-aware Sesame, we have made two extensions: 1) Frame barriers and 2) IO modeling.

Explicit frame barriers can be modeled by using an intra-application scenario as a single frame. As was discussed in Section 3.1, an intra-application scenario is encapsulated between a StartScenario and an EndScenario event. The discrete event simulator of the MPSoC architecture will process these events and, as a result, the architecture is able to get statistics on frame level. Examples are: the frame rate of an application or the number of missed deadlines (given a predefined frame rate). Obtaining these metrics does not mean that we are able to provide any guarantees about real time behavior. The aim of Sesame is to prune the design space and to identify interesting mappings with respect to several objectives like time and energy. This is also the case for the frame rates that are obtained from our simulator. Good mappings will be identified, but in order to provide any guarantees a more detailed (and probably more computationally expensive) analysis must be done on the pruned design space.

Another aspect required by SAFE is IO modeling. Where the frame barriers are purely meant to be able to reason about a good mapping with a timely delivery of frames, IO modeling makes a clear distinction between data that is visible to the user and data that is invisible to the user. For this purpose, the notion of an *Outside World Process (OWP)* is introduced that provides an interface between the application and its outside world. All output that is visible to another user must be communicated to an OWP process. Similarly, all external input is obtained by reading from an OWP process. The separation between an OWP process and a normal process also makes the fault tolerance of the application more transparent. Data coming in from a OWP process is guaranteed to be correct (as result of the fault-tolerant communication network), but the data that is sent to an OWP process for output must be verified to make the application fault tolerant. Any other communication between normal processes does not need to be verified since it is not visible to the user.

The OWP process also provides explicit knowledge of the moment that the data is sent to the user, which allows for exactly obtaining the frame rate. A frame is started

(a) *Application Layer.* **(b)** *Architecture Layer.*

Figure 6.3: *An example MPSoC model extended with IO support.*

after the first external data is read using an OWP process and it is finished after the last block of data is written to an OWP process. Combined with the workload description that was introduced in Chapter 3 (see Figure 3.3 on page 39), it can be determined how long it takes to process a frame and if the given deadline was met.

Such an outside world can manifest itself in many ways. It can be a storage device that is read by another system, but it can also be a monitor. For the application layer, these different possibilities have a similar interface, but there may be different architectural components to model the IO for storage devices or monitors. Therefore, a part of the mapping is the binding of the IO components to a suitable architectural component.

Figure 6.3 shows an example of a Sesame model that is extended with IO support. It is a *Motion-JPEG (MJPEG)* decoder (Figure 6.3a) with a four-processor architecture that uses a shared memory (Figure 6.3b). At first, the OWP0 process reads the encoded frames from an input source like a disk or an external connection. Depending on the XML workload description, all frames can be available at once (i.e., the rate of incoming frames is only bound by the buffer sizes of the communication channels) or the frames can arrive with a certain interval. Secondly, the OWP1 process will write the decoded frames to the output device. All the processes of the application need to be mapped onto the architecture, both the normal and the OWP processes. The normal processes can be mapped on any of the processors, but OWP0 can only be mapped on the CAMERA component. Similarly, OWP1 must be mapped on DISK that is an IO component capable of modeling the writing to memory and to analyze if all the frames are delivered in time.

Although it is easy to obtain the frame rate of the MJPEG example, one must be careful with the used application scenario workload. In contrast to an application with a single process, a KPN has exposed parallelism and, therefore, the KPN potentially supports pipelining. That means that different parts of the application may work on different frames simultaneously. For the example MJPEG application, this means that while the OWP1 is still working on the first frame, the OWP0 process may already have been started with processing the third frame. In order to illustrate this, Figure 6.4 shows experimental results from the MJPEG application where the mapping utilizes all of the four processors and all frames are immediately

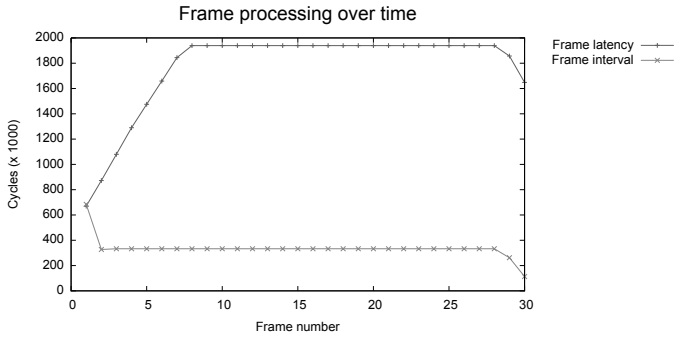


Figure 6.4: *The frame processing time of a MJPEG application.*

available. The horizontal axis shows the number of the processed frames and the vertical axis shows the corresponding cycle time for both the frame latency and the frame interval.

The frame latency is the number of cycles that it takes to process the complete frame from the start in the OWP0 process until the end in the OWP1 process. Until the eighth frame, the frame latency increases when the number of processed frames increases. In this warm up phase the system makes a cold start where the pipeline is empty and, over time, the pipeline becomes saturated. In case all of the processes were mapped to different processors, the frame latency would be less dependent on the saturation of the pipeline (there still is a shared communication network), but in our example there are only two processors for eight processes. As a result, the processors and the communication link are shared between the different processes and multiple frames may be competing for the same resources. After eight frames, the frame latency stabilizes until no more new frames are read in from disk. During this cool down period, the pipeline empties and the resource contention on the processors and the shared bus drops.

To determine the frame rate the frame interval is used. This is the difference between the arrival time of the frame that was processed previously and the arrival time of the current frame. When determining the frame rate only the saturated case is realistic. For the first frame the frame interval is still relatively large, but once the system is heated up every $333k$ cycles a new frame is finished. It is not realistic to use the interval of the last frame as a frame rate. The latency of $113k$ would result in a frame rate that is only valid during the cooling down of the application. To obtain realistic frame rates, an automatic warm-up procedure is used. As long as the processing time of a frame keeps increasing the frames are discarded for the frame rate calculation. After the first frame with a non-increasing processing time the frame rate calculation is started. To prevent the cooling down effect, frames keep being added until the simulation is halted.

(a) *Transformed Application.* **(b)** *Architecture Layer.*

Figure 6.5: *An example MPSoC model extended for active redundancy.*

6.2 Fault-Tolerant KPN Model

The experiment of the previous section showed a perfectly constant frame latency during the lifetime of the MJPEG application. This may well be the case for reliable architectures, but once the architecture is considered to be unreliable, the latency may fluctuate over time. In order to minimize these fluctuations, the application could be made fault tolerant.

This fault tolerance is important as faults may arise any time and at any place. As a starting point, we assume that the communication network is already fault tolerant. This assumption is necessary to provide a base for the software-based reliability techniques. Without it, we can, for example, not even be sure if the data that is sent from the IO component to the source node of the application is still valid. Apart from that, it is perfectly doable to implement the communication network in a fault-tolerant manner [54].

SAFE will automatically search for a good fault-tolerant implementation of the computation of the application. For this purpose, fault tolerance patterns are used to make the application reliable. A fault tolerance pattern is a software or hardware-based technique to make the application fault tolerant. Earlier we already have seen the software-based technique *Triple Modular Redundancy (TMR). Double Modular Redundancy (DMR)* is related to TMR, only DMR uses two replicas instead of three. Hardware-based fault tolerance patterns do not require any modification to the application; they only affect the architectural component(s) on which the application processes are mapped. Software-based techniques, on the other hand, may require some transformations to the application. This modification is done automatically by SAFE.

As an example, take our example of Figure 6.3. We could make a fault-tolerant implementation by applying the active redundancy technique DMR as shown in Figure 6.5. DMR performs the computation twice and compares the outcome. In this case, the application transformation (Figure 6.5a) needs to introduce two additional components: a splitter to distribute the incoming data from the OWP process to all the replicas and a majority voter to combine the results of the different replicas such

that verified data can be sent to the OWP process. Additionally, the application needs to be duplicated such that two replicas will be available. The usage of majority voting implies that only deterministic processes are supported. That means that, given error-free execution, that the same input will always generate one unique outcome. Majority voting also assumes that an error is always random. The probability that exactly the same error is made by the majority of the components should be almost zero.

The DMR also requires additional architectural components. These components are shown in Figure 6.5b and are emphasized using a gray fill. At first, an architectural component is required that is capable of providing support for the splitter and the voter. The voter must compare the outputs of the different replicas. If there is already a majority for a certain output before all different outputs are in, the resulting output can already be forwarded. In this way, the application can also tolerate if a single output does not arrive at the voter at all. The component that both provide the splitting and the voting is the NMR (an abbreviation of N-modular redundancy) component. As the architecture is unreliable, this NMR component must already be implemented in a fault-tolerant manner.

Having a NMR component that provides facilities for splitting and voting is insufficient. Whenever a voter does not observe a majority on the obtained outputs (either by corrupted data or a time-out on one or more ports), the faults that are detected cannot be masked and in this situation there are basically two options: restarting and skipping. Skipping will discard the rest of the frame and the computation is resumed with the next frame. As SAFE focuses on soft real time systems, a low number of skipped frames can still lead to a decent QoS. When too many frames are skipped, however, the QoS quickly drops. In this case, restarting may help. Restarting will retry the decoding of the frame, but to assure that the input data is still present (it may be IO that is only temporarily available) the input data must be cached. As the architecture is unreliable, this is done in *Stable Storage (SS)*. Not only the splitter has stable storage, but the voter and processors may also have stable storage. The presence of stable storage for the voter and the processors may allow the use of checkpoints as will be discussed later on.

Although this discussion will focus on active redundancy as a software-based reliability technique, other techniques are also possible. As checkpoints are modeled in SAFE (as will be discussed later), a technique like *Roll Forward Checkpointing Scheme (RFCS)* is also possible. RFCS is similar to DMR, only RFCS activates a spare replica to break the tie when a majority is absent. Other techniques can easily be added as fault tolerance patterns are defined separately as transformations on an existing application. Therefore, on the introduction of a new fault tolerance pattern it can, without further modifications, immediately be applied to all the existing applications.

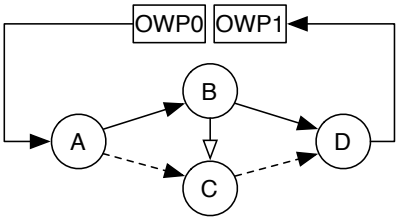
6.2.1 Patternization

A fault tolerance pattern does not necessarily need to be applied to a complete application. This technique is just one extreme in the potential ways an application can be made fault tolerant. Another extreme is to apply fault tolerance patterns individually to the application processes. If, for example, on each individual process DMR is applied then each process has its own splitter and voter. In comparison to a voter that is used for a complete application, this approach may be able to intercept a fault earlier, but it also involves significantly more overhead. Therefore, a DMR per process is not necessarily beneficial for the quality of an application. In between these solutions, there are many more solutions that leverage the overhead by dividing an application network into an arbitrary number of subnetworks on which fault tolerance patterns are applied.

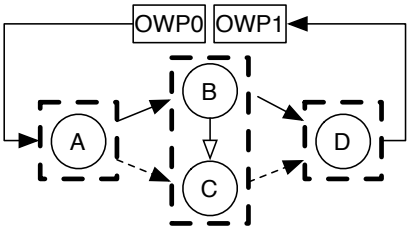
Hence, to make an application fault tolerant, the application is segregated into a number of subnetworks. On each of these subnetworks, a fault tolerance pattern is applied. The transformed application in Figure 6.5a, for example, has only a single subnetwork that is equal to the complete KPN. If more than one subnetwork is used, there are some requirements on the subnetworks. First, the complete set of subnetworks must cover the complete application and none of the subnetworks may overlap. It is absolutely crucial that the complete application is covered. When in the transformed application in Figure 6.5a, for example, the DMUX would not part of a subnetwork, the data that is processed by the rest of the application cannot be trusted anymore. A fault of the DMUX cannot be detected and, therefore, other processes can potentially operate on faulty data. Consequently, the input(s) of a subnetwork must always be verified.

Figure 6.6 gives an example of the segregation of an application. The application, as shown in Figure 6.6a, can be segregated in 15 different ways. Not all of these segregated networks, however, are valid. In our example, a subnetwork that is only consists process A and D is invalid. Since these two processes are not connected, a subnetwork like this can be considered as being two separate subnetworks. If, for example, a soft error results into a fault of process A, it may be decided to restart the frame. As process D is not directly connected to process A, it is useless to restart D as it is not dependent on process A, nor does it provide input to process A. Therefore, the subnetwork must be a weakly connected graph (i.e., if the directed edges are replaced by undirected edges there is a path between each pair of vertexes). In Figure 6.6b, a segregation is given where there are three subnetworks: A, BC and D. The idea of segregating the graph implies that for each individual subnetwork a fault tolerance pattern is applied. Based on the pick of the fault tolerance patterns, the application can be transformed into a fault-tolerant application. For our example, one of the potential fault-tolerant graphs is shown in Figure 6.6c. In this case, DMR is applied on subnetwork BC and TMR is applied to the subnetworks A and D.

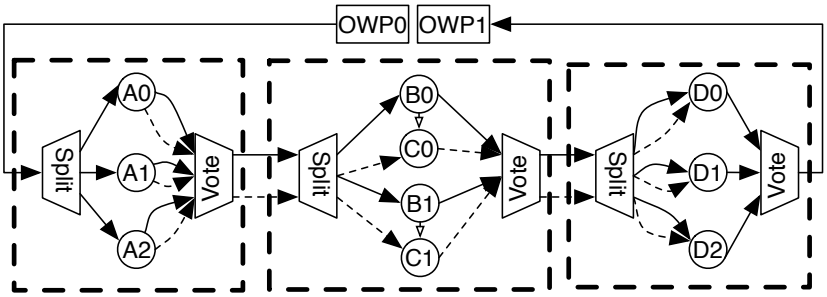
The example illustrates that there are not only many ways to segregate the application, but there are also many ways to apply fault tolerance patterns to a segregated



(a) *Original Application*



(b) *Segregated Application*



(c) *Transformed Fault-tolerant Application Graph*

Figure 6.6: *An example of application segregation in order to apply active redundancy on parts of the application instead of applying it to the complete application.*

application. If only DMR and TMR would be used, there are not less than 82 different fault-tolerant graphs for our toy application. For a real application with more processes, the search space of potential fault-tolerant application graphs even grows exponentially with respect to the number of processes and fault tolerance patterns. The combination of segregating an application and assigning fault tolerance patterns to the individual subnetworks is called patternization.

By exploring the design space of potential patternizations, a trade-off can be made between the overhead of fault tolerance and the QoS of the MPSoC design. The granularity of the subnetworks can be manipulated, where a single process per subnetwork is the highest granularity and a single subnetwork per complete application is the lowest granularity. Apart from that, the used fault tolerance pattern can be varied. The more extensive fault tolerance patterns check for the validity of the computed outputs, the more reliable the design becomes. However, this likely comes at a price of a higher overhead like computation and / or communication overhead that needs to be performed to implement the fault tolerance patterns. To automatically explore these options an automated way must be developed to describe the patternizations of an application.

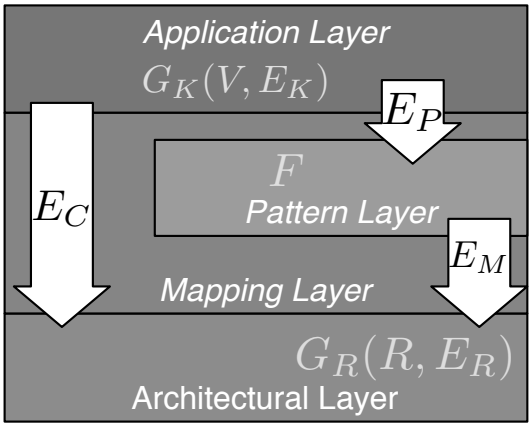


Figure 6.7: *The SAFE model with the additional pattern layer.*

6.3 Fault-Tolerant Mapping

SAFE extends the Sesame model (see Section 2.1) to support patternization by adding an additional layer: the pattern layer. To make a clear distinction between the functional behavior of the application (i.e., the application layer) and the fault tolerance behavior, the pattern layer describes the fault tolerance patterns that can be used to make the application fault tolerant. These patterns are described using an application transformation such that they are completely independent of the used application. In this way, a database of possible fault tolerance patterns can be maintained for usage in different projects.

The mapping layer extends the mapping of the original Sesame model. Before the application tasks are mapped to the architecture, the fault-tolerant mapping first performs the complete patternization by segregating the applications and transforming their application graphs using the selected fault tolerance patterns. All the tasks in the *transformed* application(s) can then be mapped onto the architecture.

In this section a formal description is given of the fault-tolerant mapping. The implementation details of SAFE will be discussed in later sections. At first, Subsection 6.3.1 formally describes the SAFE system model, after which the next subsection will formally describe the fault-tolerant mapping. To illustrate the fault-tolerant mapping, the MJPEG example of Figure 6.5 is used.

6.3.1 System Model

The main purpose of the SAFE system model is to transparently integrate the fault tolerance into the early DSE of MPSoC based embedded systems. In this subsection the SAFE model, as shown in Figure 6.7, will be described formally. The most

important mathematical definitions used in this subsection are also shown in their corresponding layer in Figure 6.7.

Application layer The application layer describes the functional behavior of the application. This is the pure behavior, not the extensions that are made to make the application fault tolerant. As an application within SAFE is represented by a KPN, the applications are modeled by a directed graph $G_K(V, E_k)$. The vertexes V represent the processes and the edges $E_k = V \times V$ represent the communication channels. To be able to make the application fault tolerant, a distinction is made between two types of processes: the normal processes V_{N} and the outside world processes V_{OWP}. A process can only be one of the types: $V_{\mathrm{N}} \cap V_{\mathrm{OWP}} = \emptyset$. From these types of processes, the OWP processes are the only processes that are allowed to interact with the world that is external to the application (like reading from a disk that is writable by other systems or displaying an image on a monitor). They are not allowed to perform any computation. Only the normal processes can do computation. For this reason, it is not allowed for OWP processes to communicate directly with each other:

$$\forall v_1, v_2 \in V_{\mathrm{OWP}} : (v_1, v_2) \notin E_k \tag{6.1}$$

Architecture layer In the architecture layer the structure of the architecture, as well as the non-functional behavior, is described using a directed graph $G_R(R, E_R)$. In this case, R represents the available architectural resources of the MPSoC. Among these resources, there are multiple dedicated resources like processors $R_P \subset R$, IO elements $R_I \subset R$ and reliability elements $R_R \subset R$. The way in which the different architectural elements are connected is described using the edges in $E_R = R \times R$.

Processors R_P are components that can be used to map normal process nodes. The IO elements R_I, however, are only meant for OWP processes. This means that, in contrast to processors, they do not perform any processing, but they only perform input and output tasks for the mapped applications. Next to the processors and IO elements, there are also reliability elements R_R. These are the main addition to the architecture that allows us to automatically add support for fault tolerance in MPSoC design. Conceptually, each reliability element consists of a splitter and a voter. The splitter takes care that the incoming data is available to all the processes in the fault-tolerant subnetwork. Not only does this involves sending data to different replicas, but it also may be required to cache the input data such that the processes can be re-executed. Another part of a reliability element is the voter that is responsible for detection of faults and handling them. Every data item that passes the voter is verified by the corresponding fault tolerance pattern. Therefore, each of the reliability elements is linked to one or more fault tolerance patterns. An example is a fault tolerance pattern where reliable computation is used. Reliable computation can only be done using a fault-tolerant processor that has built in support for detecting and handling faults. It does not need a splitter or voter and, therefore, conceptually the

splitter and voter of this reliable component just forward the data. DMR, on the other hand, has a splitter that sends the data to the two replicas and a voter to do a majority vote on the outputs of these replicas.

Referring to our example of Figure 6.5b, all of the shown resources are part of the architectural resources R. From the architectural resources, PROC-1, PROC-2 and PROC-FT are part of the processors R_P, whereas DISK and DISPLAY belong to the IO elements R_I. In the architecture there are two reliable elements (R_R): the NMR that implements active redundancy and PROC-FT that is an implementation of a fault-tolerant processor. This example also shows that it is perfectly possible that the same component belongs to two types of processors: PROC-FT both provides the ability the execute processes as well as it provides reliability.

Pattern layer The pattern layer describes the transformations that need to be done to make a subnetwork fault tolerant and to which architectural resources the fault-tolerant subnetwork can be mapped on. All the possible fault tolerance patterns are gathered in collection F. An example of fault tolerance pattern $f \in F$ is active redundancy [13] where the processes are executed multiple times and the outcome is compared. Such a fault tolerance pattern does not only describe the reliability technique. In case of the active redundancy, for example, action must be taken in case a fault is detected. Next to that, a fault tolerance pattern may involve taking checkpoints to be able to re-execute the code at arbitrary points in time. Therefore, parameters like the action to be taken on the detection of faults and the frequency of checkpoints (see Section 6.4.3) are also part of a fault tolerance pattern.

As an example take the three fault tolerance patterns DMR, TMR and FPROC. The DMR and TMR are active redundancy techniques, whereas the FPROC is a fault-tolerant processor. Each of the fault tolerance patterns uses a different number of replicas. Therefore, a function $n_{\text{proc}}(f)$ is defined that describes the number of processors that are required for fault tolerance pattern $f \in F$. In our MJPEG example, the function $n_{\text{proc}}(\text{FPROC})$ returns 1, whereas $n_{\text{proc}}(\text{DMR})$ and $n_{\text{proc}}(\text{TMR})$ are, respectively, 2 and 3.

Mapping layer Finally, the mapping layer glues the application, pattern and architecture layers together. For this purpose, three types of edges are used: patternization edges, mapping edges and IO edges.

Patternization edges E_P describe both the segregation of the application and the selection of the fault tolerance patterns for each subnetwork. Each edge $(v, f) \in E_P$ represents a possible appliance of the fault tolerance pattern $f \in F$ for the process $v \in V_N$. That means that all the processes that are connected to the same fault tolerance pattern f belong to the same subnetwork. A consequence of this approach is that the same pattern must be present multiple times in the collection F in case the same pattern must be available for multiple subnetworks. This duplication is done automatically during the initialization of SAFE such that each individual process

feasible(e, r)	e = DMR	e = FPROC	e =OWP0	e =OWP1
r = PROC-1	✓	–	–	–
r = PROC-2	✓	–	–	–
r = PROC-FT	–	✓	–	–
r = NMR	✓	–	–	–
r = CAMERA	–	–	✓	–
r = DISK	–	–	–	✓

Table 6.1: *The values for the feasible function for the MJPEG example in Figure 6.5. In this table '✓' is* `True` *and '–' corresponds to* `False`*.*

can use the same pattern in its own subnetwork. The user, however, may limit the number of duplications of a fault tolerant pattern.

Mapping edges E_M assign the architectural resources to the application-level components that will be used by a fault tolerance pattern. More precisely, the edge $(f, r) \in E_M$ assigns architectural element $r \in R_P \cup R_R$ to pattern $f \in F$. Not all mapping edges are valid:

$$(f, r) \in E_M \iff \text{feasible}(f, r) \tag{6.2a}$$

$$\forall f \in F : |\{r|(f, r) \in E_M \wedge r \in R_R\}| \geq 1 \tag{6.2b}$$

$$\forall f \in F : |\{r|(f, r) \in E_M \wedge r \in R_P\}| \geq 1 \tag{6.2c}$$

Depending on the fault tolerance pattern, other types of architectural resources may be used for the mapping of a transformed fault-tolerant application. For our MJPEG example, the feasible function is shown in Table 6.1. The first two columns show the two fault tolerance patterns DMR and FPROC. A DMR makes the execution reliable by voting on the results using the NMR component. Therefore, its replicas can be mapped onto the generic processors PROC-1 and PROC-2. For the FPROC pattern, on the other hand, only a special fault-tolerant processor PROC-FT can be used. In general, each fault tolerance pattern must have at least one voter component on which it can be mapped (Equation 6.2b) and one processor component (Equation 6.2c).

Next, the *IO edges* (E_{IO}) bind the OWP processes to an IO component in the architecture. To be precise, an IO edge $c = (v, i)$ assigns the OWP process $v \in V_{\text{OWP}}$ to an IO component $i \in R_I$ in the architecture. Similarly to the mapping edges, the IO edges also must comply with the feasibility requirements. As there are many types of IO, there also are a large potential number of architectural components to map onto. The feasibility requirement is as follows:

$$(v, r) \in E_{IO} \iff \text{feasible}(i, r) \tag{6.3a}$$

$$\forall v \in V_{\text{OWP}} : |\{r|(v, r) \in E_{IO}\}| \geq 1 \tag{6.3b}$$

This requirement enforces that there is at least one potential mapping for any of the OWP processes. Without a potential IO edge there would not be a valid fault-tolerant mapping. The feasible IO edges of the MJPEG example are shown in Table 6.1. The last two columns show the OWP processes OWP0 and OWP1. Each of them complies with the requirement that there is at least one IO edge (Equation 6.3b), namely, CAMERA for OWP0 and DISK for OWP1.

Mapping the transformed application graph and the OWP processes is not sufficient. Due to adding processes and replicating processes while transforming the application graph, there will be more messages in the embedded system than in the original application. All of these messages need to be dispatched, where special care must be taken if a message need to be sent to multiple destinations (the splitter) or when multiple messages needs to be sent to the same destination (the voter). Therefore, a final step in the mapping layer is the message dispatch that determines the destination of individual messages that are sent.

Let $Q = V_{\mathrm{N}} \times V$ be the set of requests. A request is done by a process node and is an act to ask for reading or writing data to another process node. Only after the destination confirms the request, the read or write can be handled. Read requests can only be handled if the data is present, whereas write request require available buffer space. The normal process node v_s initiates the request $(v_s, v_d) \in Q$ (An OWP process is passive and can therefore not initiate a request) and sends it to the other process v_d. There are two types of requests: one set of read requests $Q_{\mathrm{READ}} \subseteq Q$ and one set of write requests $Q_{\mathrm{WRITE}} \subseteq Q$. These sets of requests need to comply with the following requirements:

$$Q = Q_{\mathrm{READ}} \cup Q_{\mathrm{WRITE}} \tag{6.4a}$$

$$Q_{\mathrm{READ}} = \{(v_s, v_d) | (v_d, v_s) \in E_k \wedge v_s \in V_N\} \tag{6.4b}$$

$$Q_{\mathrm{WRITE}} = \{(v_s, v_d) | (v_s, v_d) \in E_k \wedge v_s \in V_N\} \tag{6.4c}$$

Per communication link a single request is added, where Equation 6.4b is for a read request and Equation 6.4c is for a write request. Process v_s is the process node that initiates the request and process v_d is the node where the request is sent. Both of the read and write request must be done from a normal process node ($v_s \in V_N$). The difference between the requirement of the read request (Equation 6.4b) and the write request (Equation 6.4c) is the direction of the communication link. For a write request, the communication link starts at the requesting nodes and ends at the destination node ($(v_s, v_d) \in E_k$), whereas for a read request the communication link is reversed: it starts at the destination process and it ends at the requesting node ($(v_d, v_s) \in E_k$).

As SAFE deals with fault-tolerant application networks, a request is not sufficient to exclusively define a single message. A node may be replicated and, hence, the same requests may be initiated from different locations. For that reason, a message is the

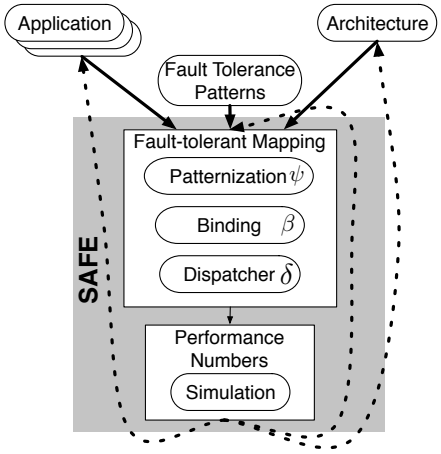


Figure 6.8: *The modified Y-chart for synthesis with fault tolerance support.*

combination of a request and a location: $M = Q \times R$. Based on the request and the location, the *dispatch edges* E_D determine the target architectural element for the request. When $(m, r) \in E_D$, message $m \in M$ needs to be sent to architectural resource $r \in R$.

6.3.2 Mapping Procedure

The previous section introduced the system model of SAFE. A next step is to use this system model to map the applications onto the architecture. During such a mapping the fault tolerance must be taken into account. For this purpose, the Y-chart approach [38] that is used by Sesame must be extended to incorporate the fault tolerance patterns. This modified approach, as shown in Figure 6.8, has three inputs: 1) the applications, 2) the fault tolerance patterns and 3) the architecture. Due to its three inputs it looks more a Ψ-chart than a Y-chart, but the concept is the same. With the application, fault tolerance patterns and the architecture, a fault-tolerant mapping is made. Using SAFE, the performance of the fault-tolerant mapping is obtained using simulation and based on these results the fault-tolerant mapping can be optimized. For our Ψ-chart, this means that the applications, the fault tolerance patterns and the architecture can be changed, but also that a different fault-tolerant mapping is used.

This subsection focuses on the fault-tolerant mapping that not only performs the binding of the applications onto the architecture, but before binding it also transforms the applications into fault-tolerant applications. More specifically, there are three steps: 1) patternization (ψ), 2) binding (β) and 3) message dispatch (δ). First, the patternization splits the application into subnetworks that each gets their own fault tolerance pattern. Next, the binding binds the components of the fault-

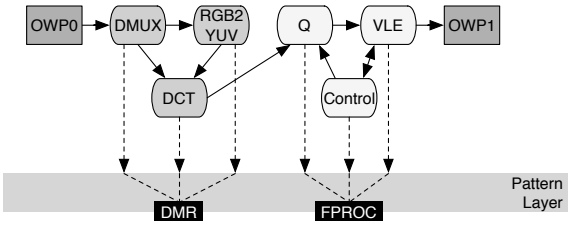


Figure 6.9: *Potential patternization for the MJPEG example.*

tolerant applications onto the architecture after which the message dispatch will specify the destination of all the messages that are sent between the architectural resources. In the following, a detailed description of each step will be given.

Patternization A fault-tolerant mapping starts with a patternization that divides the applications into subnetworks by selecting fault tolerance patterns for each individual process. Formally, the patternization ψ contains one pattern edge for each normal node in V_{N} such that $\psi \subseteq E_P$:

$$\forall v \in V_{\mathrm{N}} : |\{f|(v,f) \in \psi\}| = 1 \tag{6.5}$$

Only normal nodes need to be taken into account during patternization. OWP processes do not perform any processing, but only communication. Since, we assume that the communication is already fault tolerant, OWP processes do not need to be guarded. Implicitly, this assignment of fault tolerance patterns $f \in F$ divides each application into subnetworks that use the same architectural support for their fault tolerance. The definition of a *fault-tolerant subnetwork* (G_f) with fault tolerance pattern $f \in F$ is as follows:

$$G_f := \{v|(v,f) \in \psi\} \tag{6.6}$$

$$\forall v_1, v_2 \in G_f : \mathrm{weakly\_connected}(v_1, v_2, E_k) \tag{6.7}$$

As discussed before, performance wise it is important that the processes in the subnetworks are weakly connected (which is enforced by Equation 6.7). If this would not be the case, processes may, for example, be unnecessarily restarted after a transient fault of an unrelated process in the subnetwork.

Figure 6.9 shows one of the potential patternizations of the MJPEG application. Basically, the six normal processes are split into two subnetworks G_{DMR} and G_{FPROC}. The first subnetwork G_{DMR} uses the DMR technique that makes two replicas of the subnetwork $\{\mathrm{DMUX}, \mathrm{RGB2YUV}, \mathrm{DCT}\}$. For the other part of the application, the subnetwork G_{FPROC} uses a specialized fault-tolerant processor to execute the processes Q, Control and VLE in a reliable manner. If the complete patternization

ψ would be written down, the result would be as follows:

$$\begin{aligned}\psi = \{&(\text{Control}, \text{FPROC}), (\text{DCT}, \text{DMR}),\\ &(\text{DMUX}, \text{DMR}), (\text{RGB2YUV}, \text{DMR}),\\ &(\text{Q}, \text{FPROC}), (\text{VLE}, \text{FPROC})\}\end{aligned}$$

Computational Binding Conceptually, the computational binding maps the transformed fault-tolerant application graphs onto the architecture. However, this mapping is not done per individual process. The reason for mapping one fault-tolerant subnetwork at the time is that some fault tolerance patterns require the different processes to be on the same processor to work efficiently. Take, for example, explicit checkpoints (see Section 6.4.3) where all the processes of a subnetwork must be halted before the checkpoint can be taken. If the replica processes are on the same processor (the other replicas may still be mapped on a different processor), the local checkpoint procedure will not require any external communication.

The computational binding β_x will map the fault-tolerant subnetworks onto two types of architecture components. At first, a voter is selected for splitting and combining the in and outgoing data. Recall that for some fault tolerance patterns these components will simply forward the data (like the fault-tolerant processor that does not need to split and combine the data). Voters and splitters are, just like replicas, provided as a general description of a fault tolerance pattern. Based on this general description of a fault tolerance pattern, a general mapping procedure can be defined. The real implementation of voters and splitters, however, may be an empty function. Secondly, a fault-tolerant mapping requires the selection of a set of processors that is used to run the individual processes. To be precise, the computational binding $\beta_x \subseteq E_M$ uses mapping edges to select architecture components for the subnetworks:

$$\forall f \in F : G_f = \emptyset \ \vee (\text{HAS\_VOTER}(f) \wedge \text{HAS\_REPLICA}(f)) \tag{6.8a}$$

$$\text{HAS\_VOTER}(f) := |\{r | (f, r) \in \beta_x \wedge r \in R_R\}| = 1 \tag{6.8b}$$

$$\text{HAS\_REPLICA}(f) := |\{p | (f, p) \in \beta_x \wedge p \in R_P\}| = n_{\text{proc}}(f) \tag{6.8c}$$

These equations make certain that architectural resources are selected for processing the tasks and a voter element is selected to make the outcome reliable. The capacities of these components, however, are not taken into account during the creation of the fault-tolerant mapping. A fault-tolerant mapping can, for example, map ten fault-tolerant subnetworks onto the same NMR component. It depends on the NMR component if it really has the capabilities to support ten fault-tolerant subnetworks. The simulation verifies if a fault-tolerant mapping adheres to the maximal capabilities of the architectural resources. As simulation knows exactly the data requirements of the applications, it can analyze if the voter capacity is exceeded.

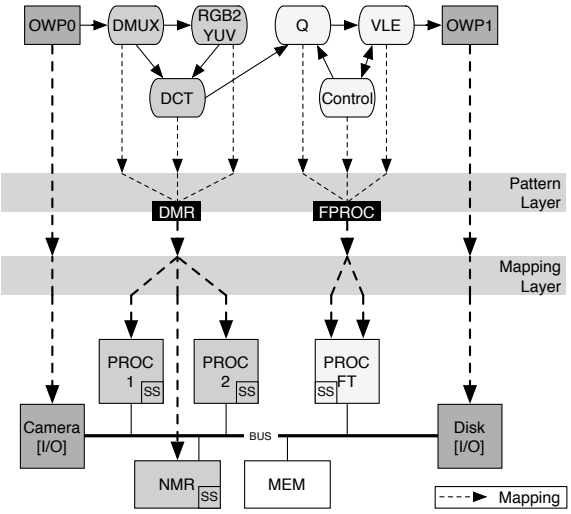


Figure 6.10: *Potential binding for the MJPEG example.*

An example of a computational binding is shown in Figure 6.10. As a computational binding cannot be defined without a patternization of the application, this example binding is based on the patternization in Figure 6.9. There are two subnetworks that need to be mapped: G_{DMR} and G_{FPROC}. The first subnetwork G_{DMR} uses the NMR component as a voter component and as replicas the processors PROC-1 and PROC-2 are used. Looking to Table 6.1, NMR is in this architecture the only potential voter that can be chosen for the DMR pattern. For the processors, however, there are multiple options. Where this example chooses to map each replica to a different processor, there can also be chosen to map the replicas onto the same processor. This leads to replication in time instead of in space. The mapping of the FPROC is even more restricted than the DMR pattern. Both the voter and the single replica (which is strictly speaking not a replica, but the original application subnetwork) can only be mapped onto one architectural component: the PROC-FT. This means that the same mapping edge is added twice to the computational binding: once for the voter and once for the application processes. The mapping of the voter is purely for defining a formal and generic model of a fault-tolerant mapping. The real implementation will not contain the voter (only a fault-tolerant processor that runs the original processes). Hence, the complete computational binding becomes:

$$\beta_x = \{(\text{DMR}, \text{PROC-1}), (\text{DMR}, \text{PROC-2}), (\text{DMR}, \text{NMR}), \\ (\text{FPROC}, \text{PROC-FT}), (\text{FPROC}, \text{PROC-FT})\}$$

IO binding Binding the transformed application graph is not sufficient since the OWP processes must also be bound onto the architecture. As can be expected, IO

binding β_{io} defines exactly one IO edge for each of the OWP processes such that $\beta_{io} \subseteq E_{IO}$:

$$\forall v \in V_{\text{OWP}} : |\{i|(v,i) \in \beta_{io}\}| = 1 \tag{6.9}$$

The IO binding of the MJPEG application is, just as the computational binding, shown in Figure 6.10. There are two OWP processes (OWP0 and OWP1) that need to be mapped onto the architecture. Due to the feasibility rules (see Table 6.1), there is only one possible IO binding:

$$\beta_{io} = \{(\text{OWP0}, \text{CAMERA}), (\text{OWP1}, \text{DISK})\}$$

Message Dispatch After determining the computational and the IO binding, the only remaining aspect is the message dispatch δ that deals with the communication between the processes. The message dispatch $\delta \subseteq E_D$ can be seen as the routing tables for the complete MPSoC. If $((q, r_s), r_d) \in \delta$, then the routing table at architectural element r_s will contain the destination element r_d for the request q. For the routing, there may be, depending on the communication architecture, different possibilities for the routing. Although this is also a potential design decision that can be explored, we have chosen to keep the routing fixed. Therefore, the architectures that we explore in our experiments only have one routing possibility between the different architectural components.

In order to verify if the dispatch is valid and complete, the set of messages in the mapped system needs to be determined. As discussed before, a message is the combination of a request (read or write) and the architectural resource the request is sent from. Therefore, the set $M_\delta \subseteq M$ contains the requests and the architecture resource of all the processes in the transformed application graph. The validity of the set of messages and the dispatch can be verified as follows:

$$\forall((v_s, v_d), r) \in M_\delta : \exists f \in F \mid ((v_s, f) \in \psi \wedge (f, r) \in \beta_x) \tag{6.10a}$$

$$\forall m \in M_\delta : |\{r|(m, r) \in \delta\}| \leq 1 \tag{6.10b}$$

The first requirement (Equation 6.10a) checks if the source node of the request is really mapped onto the resource that is specified in the message. Next, Equation 6.10b ensures that each message can have at most one destination.

These requirements, however, only verify that each message that is listed in the dispatch contains correct information. Another requirement is that it is complete. For each request $q \in Q$ that may be encountered during the lifetime of the embedded system, the dispatch should have the correct destination node for the communication:

$$\forall (v_s, v_d) \in Q : \begin{cases} \text{Internal}(v_s, v_d, f) & \exists f \in F : v_s, v_d \in G_f \\ \text{External}(v_s, v_d) & \nexists f \in F : v_s, v_d \in G_f \end{cases} \tag{6.11}$$

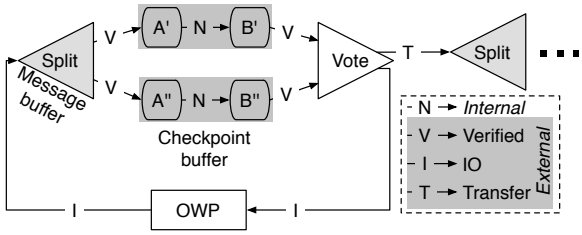


Figure 6.11: *The different types of communication and the required buffers to enabling restarting.*

There are two types of communication: internal (the process nodes are in the same subnetwork) and external communication (the other process is an OWP process or in a different subnetwork). Figure 6.11 shows the communication types. *Internal communication* is communication between two process nodes in the same fault-tolerant subnetwork (Figure 6.11 shows an example of such communication between processes A and B). Since the communication is within a single subnetwork, the data messages do not need to be verified. Similarly, it is known that internal communication is done within one and the same processor (as a complete replica of a subnetwork is mapped to a single processor). Therefore, the entry in the message dispatch δ is as follows (notice that the message is sent to the same resource r from which it is initiated):

$$\text{Internal}(v_s, v_d, f) := |\{r|(((v_s, v_d), r), r) \in \delta \wedge r \in R_P\}| = n_{\text{proc}}(f) \tag{6.12}$$

In case the processes of the link are not in the same subnetwork, the communication is external. *External communication* passes the voter and consists of two steps:

$$\text{External}(v_s, v_d) := \text{V}(v_s, v_d) \wedge \begin{cases} \text{IO}(v_s, v_d) & v_d \in V_{\text{OWP}} \\ \text{Transfer}(v_s, v_d) & v_d \in V_N \end{cases} \tag{6.13}$$

First, *verified communication* (V) takes care of the communication from the replicated processes in subnetwork G_f to the voter r_d. The voter r_d (Equation 6.14a) will verify the contents of the messages of the different replicas (Equation 6.14b) of v_s:

$$V(v_s, v_d) := (v_s, f) \in \psi \wedge (f, r_d) \in \beta_X \wedge r_d \in R_R \wedge \tag{6.14a}$$

$$|\{r|(((v_s, v_d), r), r_d) \in \delta \wedge r \in R_P \wedge r_d \in R_R\}| = n_{\text{proc}}(f) \tag{6.14b}$$

Equation 6.14b ensures that the number of replicas is exactly the same as the fault tolerance pattern defines (by means of $n_{\text{proc}}(f)$).

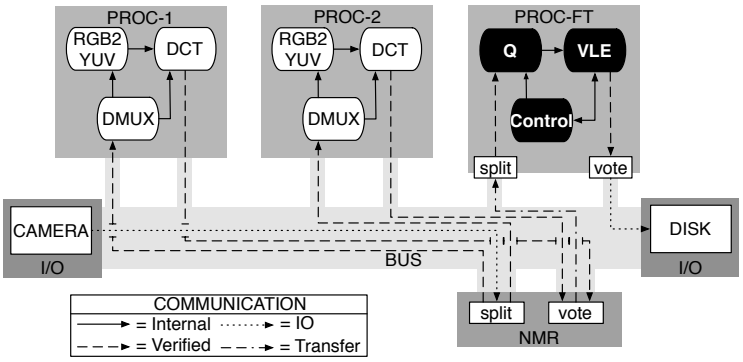


Figure 6.12: *The synthesized design for the mapped MJPEG example in Figure 6.10.*

The second step of external communication depends on the type of the process node v_d. If the destination process is a normal process node ($v_d \in V_{\mathrm{N}}$), there will be a *transfer* (T) of the message to the subnetwork of v_d. In case of an OWP process ($v_d \in V_{\mathrm{OWP}}$), *IO* is performed:

$$\begin{aligned} \mathrm{IO}(v_s, v_d) := & |\{r|((v_s, v_d, r), r_d) \in \delta \wedge r \in R_R \\ & \wedge (v_d, r_d) \in \beta_{io} \wedge r_d \in R_I\}| = 1 \end{aligned} \tag{6.15}$$

$$\begin{aligned} \mathrm{Transfer}(v_s, v_d) := & |\{r|((v_s, v_d, r), r_d) \in \delta \wedge r, r_d \in R_R \\ & \wedge (v_d, f_d) \in \psi \wedge (f_d, r_d) \in \beta_x\}| = 1 \end{aligned} \tag{6.16}$$

This second step only needs to be done once, irrespective of the number of replicas. For a transfer, the destination resource must be the voter on which the subnetwork of v_d is mapped. In case of IO, the destination resource must be the IO element on which process v_d is mapped. If these requirements are all met, the internal and external communication is completely defined within message dispatch δ.

An example of the dispatch for the MJPEG application is shown in Figure 6.12. This illustration shows the complete synthesized design of the mapped application from Figure 6.10 where the processes of subnetwork G_{DMR} (DCT, DMUX and RGB2YUV) are replicated twice and mapped onto PROC-1 and PROC-2. Internal communication links (like DMUX, DCT) are duplicated for each replica and, therefore, are present twice in the dispatch: both for PROC-1 and for PROC-2. As the source and destination elements are the same, an internal buffer within the processor can handle the communication. For this communication link no other messages are required.

This is different for external communication. As shown in Equation 6.13, it consists of verified communication from the replicated process to the voter and an additional

communication step to get the message to the target process. An example of verified communication is the write request between the DCT process on processor PROC-1 and the voter on the NMR component. A verified communication of a read request, on the other hand, is connected to the splitter. An example would be the communication between the DMUX process on processor PROC-2 and the splitter on the NMR component. After the verification of the data, the splitter must forward the request to its destination. Depending on the type of destination node, different types of communication are involved: IO and Transfer.

A transfer is between two normal process nodes that are on different fault-tolerant subnetworks. In Figure 6.12, this is the case for the communication between processes DCT and Q. Upon the write request of DCT the verified data will be transferred from the voter component on the NMR element to the splitter component of the PROC-FT element. Before the DCT will read the data, it will send a read request. This read request is passed on to the splitter. If the data has already arrived earlier with a write request, the splitter will immediately provide the data. Otherwise, it will send the read request to the voter of the NMR and wait until the data is available.

Another possibility is an IO request that is addressed to an OWP process. As an OWP process is passive, its only activity is to wait on requests and to process them as soon as the data can be read or written. In Figure 6.12 the communication link (VLE, OWP1) involves IO. After the voter of the NMR component verifies the data, it will be sent to the DISK element that is now able to reliably store the sequence of frames encoded by the MJPEG encoder.

A fault-tolerant mapping is the complete combination of the patternization ψ, the computational binding β_x, the IO binding β_{io} and the dispatch δ. To be valid, the conditions in Equations 6.1, 6.2, 6.3, 6.5, 6.7, 6.8, 6.9, 6.10, and 6.11 must be met.

6.4 Simulation Model

A valid fault-tolerant mapping can be used to synthesize a fault-tolerant MPSoC design, as illustrated in Figure 6.12. In order to obtain the quality (like performance, energy, but also reliability) of such a design, SAFE simulates the multi-application workload in an unreliable environment. For this purpose, the SAFE simulation model extends the Sesame simulation framework in several ways. The first subsection discusses the fault injection into the architecture, followed by a subsection that describes the fault detection. After the detection of a fault, it must be handled in some fashion. Therefore, fault correction is introduced in the simulation model by means of restart with or without the usage of checkpoints. These aspects are elaborated on in the third subsection, after which the last subsection will discuss the obtained performance metrics.

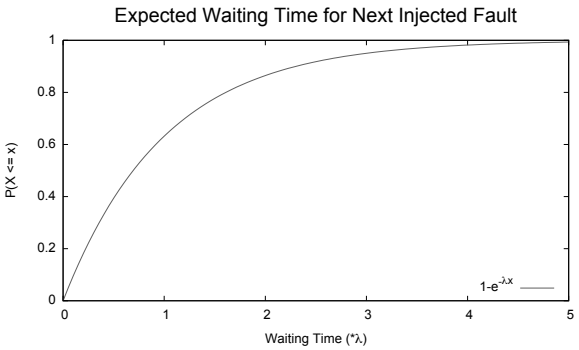


Figure 6.13: *The cumulative distribution function of the exponential distribution that is used to model the waiting times between the faults.*

6.4.1 Fault Injection

SAFE is able to model a fault-tolerant MPSoC design, but in order to test the quality of the design, it must be determined how faults will affect the execution of a multi-application workload on the MPSoC. Given the complexity of the simulation that is done within Sesame, an analytical analysis method would not be applicable. Sesame models the behavior of each individual component, including contention and other metrics that are hard to capture in an analytical method, and, therefore, the effect of the faults on the behavior of each individual component must also be taken into account. Examples are the potential communication bottleneck due to all the additional communication that is required, but also the effects of making checkpoints on a processor and the maximal amount of storage that such checkpoints require can be obtained when simulation is used.

We assume that only the processor elements are susceptible to faults. Other components like the communication network, reliability elements and the IO elements are assumed to be fault tolerant. The processors may have many components that can be affected by faults [5] like the register file and the logical units. Our architecture model, however, uses a high level of abstraction to describe the processor: it is known what function the processor is executing, when the function is running, how long it takes and when data is read or written. Therefore, we use the *SoftWare Initiated Fault Injection (SWIFI)* method [61]. Although both transient and permanent faults can be modeled this way, currently we only focus on transient faults.

For the occurrence of faults an exponential random distribution is used. As Figure 6.13 illustrates, the waiting time between the different faults is described using an exponential distribution. This resembles a Poisson process where the value λ^{-1} is equal to the *Mean Time To Failure (MTTF)*. This distribution is very suitable for modeling the fault injection times [18], as the events in a Poisson process occur continuously and independently at a constant average rate. This is also the case for

transient errors that are caused by a SUE: they are independent of earlier faults and they happen infrequently [29]. Some other classes of transient errors do show correlation between different errors. An example is the common mode failure [29]. These correlated faults are out of scope of this thesis and should be avoided using external techniques [47].

During the simulation, faults are injected one by one by iteratively picking a random number that is exponentially distributed. When a processor is hit by a transient fault, it does not automatically mean that all the applications are affected. The fault will only have consequences for the process that is running at the moment that the fault occurs. In case there is an active process, all future output of the process for the current frame will be considered erroneous. In reality, a fault may not affect the output, but in order to analyze this we require more details than can be provided by our high level model. Therefore, we take the most pessimistic assumption.

6.4.2 Fault Detection

Erroneous data will be propagated through the network, until it reaches a point where the fault-tolerant subnetwork verifies the data. For active redundancy, this is only the case when the data leaves the subnetwork. At this point in time, the majority voter will compare the results of the different replicas. For other techniques, like the fault-tolerant processor, the fault detection may almost be instantaneous.

Upon the detection of a fault, it depends on the fault tolerance pattern what is done to handle it. Generally, there are two possible actions that can be taken after the detection of a fault: fault masking and fault correction. *Fault masking* is applied when the correct data is known. When TMR is used, for example, a single fault among the three replicates can be masked as the correct value still has a majority. Without any time overhead, the fault can be masked by forwarding the correct data. As we currently only take transient faults into account, the corrupt replica is allowed to continue its execution. Since there is no explicit state between different frames in the KPN based applications (as discussed in Section 3.3), the next frame will not be affected by this transient fault anymore.

When the correct data is not known, the fault must be corrected. This is, for example, the case with DMR where a single fault can be detected, but the majority vote cannot identify which replica is incorrect. Depending on the available slack time, the current frame can be skipped or the fault is corrected. In the next section we will show how restarting is one of the potential ways of correcting a fault.

6.4.3 Fault Correction

In contrast to fault masking, fault correction will involve some resource overhead (like cycle time, energy or storage space). Therefore, it may seem to be the case that fault correction is not usable for embedded systems as the resources are quite

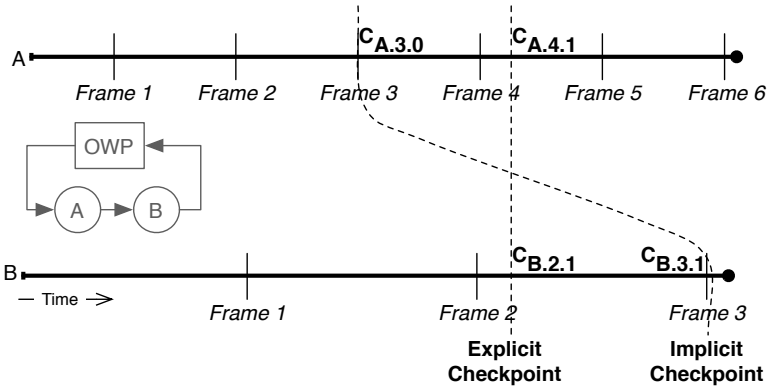


Figure 6.14: *The timeline of a simple application to illustrate an implicit versus an explicit checkpoint.*

stringent. However, with the increasing number of faults in time, the QoS of the embedded system may be severely affected. To be able to provide insight into the trade-off between QoS and resource overhead, SAFE provides the capability to model and simulate checkpoints and restarting the system from one of these checkpoints. Therefore, the fault correction techniques of SAFE can be extended upon the introduction of new fault tolerance patterns.

Checkpoint Budget

As illustrated in Figure 6.14, there are two types of checkpoints: implicit and explicit. *Implicit checkpoints* are located at the frame barriers. In the example of Figure 6.14 checkpoints $C_{A.3.0}$ and $C_{B.3.0}$ form the implicit checkpoint at frame 3. Notice that an implicit checkpoint is not taken at once. Process A reaches the barrier of frame 3 much earlier than process B. The complete implicit checkpoint is available once every process in the subnetwork has reached the specific frame barrier. A complete implicit checkpoint does not require storage (as there is no implicit state between frame barriers), but it allows us to perform message cleanup as will be discussed later on.

Similarly, the restart from an implicit checkpoint is trivial because no state needs to be restored for the processes. Still, restarting from an implicit checkpoint has some disadvantages. Not only is there a need to recalculate the complete frame, but it can also be the case that the application is not restarted at its full capacity. Take the simple application in Figure 6.14. On a restart from an implicit checkpoint at the end of frame 2, both processes start at the barrier of frame 2. In this case, process B needs to wait for output of process A before it can do any work.

To resolve this, explicit checkpoints can be taken during the lifetime of an ap-

plication. *Explicit checkpoints* are initiated by the voter and store the state of all the processes in the active redundancy network and their internal communication channels at a specific point in time. In contrast to implicit checkpoints, the complete subnetwork is halted to capture the state of the current processes and the internal communication channels. To obtain a consistent explicit checkpoint, the voter will ensure that all the replicas stop at the same point in the application. One of the processes in the subnetwork will be responsible for collecting and sending the checkpoint. This results in a checkpoint for each replica, which will be compared by the voter. In Figure 6.14, $C_{A.4.1}$ and $C_{B.2.1}$ are illustrations of an explicit checkpoint.

Implicit and explicit checkpoints can also enhance each other. Take, for example, the checkpoints in Figure 6.14. If there is a restart just after the checkpoint of $C_{B.3.1}$, the restart can take place from a combination of the explicit checkpoint $C_{*.*.1}$ and the implicit checkpoint $C_{*.3.*}$. This means that process A starts from $C_{A.4.1}$ and process B starts from $C_{B.3.1}$. Without the presence of explicit checkpoints, process A would have processed frame 3 again. Similarly, with only explicit checkpoints, process B would have been required to process frame 2 again.

For each fault-tolerant subnetwork, a *checkpoint budget* is defined. The checkpoint budget determines the checkpoint granularity by describing the number of explicit checkpoints per frame (possibly zero). The size, and thus the overhead, of the checkpoint is dependent on the application processes and the amount of data in the internal communication channels. This also means that the voting time of a checkpoint is variable. That is, the explicit checkpoints of the different replicas in a subnetwork must be verified against each other to ensure that on a restart the process state is valid. After verification by the voter, the explicit checkpoint is stored locally at the processor.

Explicit checkpoints are not only useful to minimize the amount of work that has to be redone, but they also allow to implement fault tolerance patterns like RFCS [58] or assertion-based techniques [28] where a fault is corrected by reprocessing the frame. This is, however, beyond the scope of the current paper.

Restart Budget

Using the checkpoints, a subnetwork is able to restart the frame upon the detection of a non-maskable fault, but this is not sufficient. As restarting requires overhead, it must be prevented that a subnetwork keeps restarting the same frame or that the restart of a frame leads to the deadline miss of the next frame. Therefore, a restart budget is specified that limits the number of times that a subnetwork can restart per completed frame. Similarly to the checkpoint budget, this is a nonnegative number that also can be zero to disable restarting. Additionally, a maximal processing time per frame can be defined after which the frame cannot be restarted anymore. In case the restart budget is depleted and a non-maskable fault is observed, the current frame is skipped.

In order to facilitate restarting, some data must be stored. These buffers are

shown in Figure 6.11. At first, the explicit checkpoints can optionally be stored locally at each of the processors that are used to run the replicas. In this case restarting becomes somewhat more efficient at the cost of the additional storage. In case no local copy of the checkpoint is available, the checkpoint will be stored and distributed by the voter. Additionally, a message buffer must be available at the splitter in order to ensure that the input data is still available when the subnetwork is restarted. Input data can be volatile due to two reasons: 1) input data is only temporarily available as input IO or 2) other subnetworks generate the input data. In the case of volatile data, the buffer prevents a cascading effect of other subnetworks that need to be restarted to provide the input data of the subnetwork. Both of the buffers can be protected from errors in several ways, but as SAFE is currently only targeted towards transient faults an ECC protection of the buffer should be sufficient.

The buffers that are modeled in SAFE have a limited capacity. Therefore, a policy is required to periodically clean up the unused messages. For this cleanup, two moments are chosen: during implicit checkpoints and during explicit checkpoints. After a successful explicit checkpoint, the complete contents of the message buffer can be discarded. As the explicit checkpoint contains the state of the process at that moment, it is guaranteed that the earlier messages are not required anymore. In case of an implicit checkpoint, the messages of the earlier frames can be discarded. In case the message buffer is out of capacity, all the new read request are blocked until the next explicit checkpoint (as discussed before, the explicit checkpoint will empty the message buffer and, therefore, will restore the full buffer capacity).

6.4.4 Performance Metrics

All the decisions with respect to the fault detection and handling will affect the quality of the system. Whenever an explicit checkpoint is taken the complete subnetwork needs to be halted. On top of that, local checkpoints need to be sent to the voter to be compared for correctness. A restart requires the restoring of the process states and the internal communication channels (only for an external checkpoint, not for an implicit checkpoint) and the splitter must resend all the cached incoming messages. All these aspects will affect the cycle time, the amount of communication and the required buffer space.

SAFE completely simulates the system including all the aspects at a high abstraction level such that an early fault-tolerant design space exploration can be performed. Metrics that can be obtained are, besides performance and power, frame rate and amount of skipped frames. It should be noted, however, that SAFE is not aimed to give any guarantees for real time behavior. SAFE will prune the design space such that only a limited number of designs need to be studied in detail during later design phases.

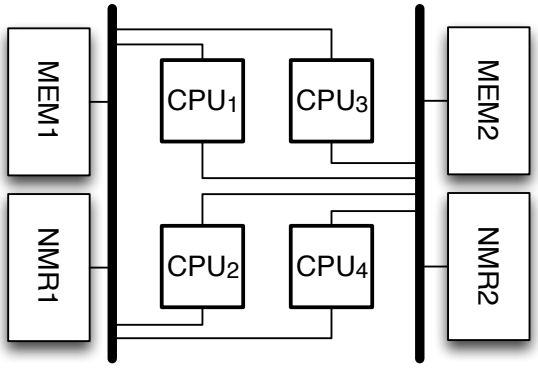


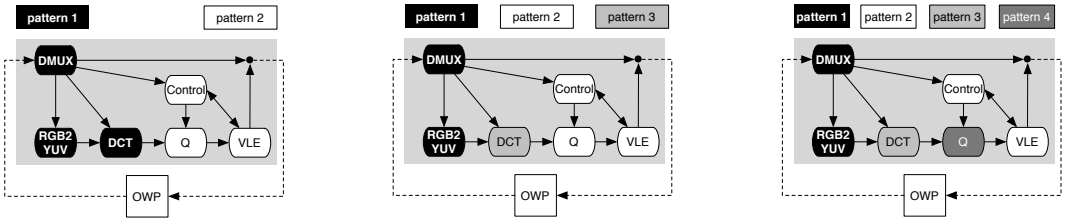
Figure 6.15: *The used architecture for the experiments in this chapter.*

6.5 Experiments

The design space pruning is shown using a set of experiments that explore the fault-tolerant design space. For this purpose, three different applications have been selected: a MJPEG encoder, a Sobel edge detector and an MP3 decoder. All of these applications work on a stream of frames, where each frame is manipulated. The Sobel edge detector, for example, detects edges for each image in a video stream. Pedestrian detection is one of the techniques that utilize the detected edges in a video frame. Earlier in this chapter, we already saw an example of the MJPEG application. In the experiments, we use a slightly modified version that has more communication links to provide dynamic quality control of the image encoding.

Figure 6.15 shows the architecture on which the applications need to be mapped. It has four general-purpose processors that are connected to two communication buses. Both of these buses have a shared memory and a NMR component to support active redundancy. In this thesis, we have limited the fault tolerance patterns to the different flavors of DMR (two replicas) and TMR (three replicas). For both the DMR and TMR the checkpoint and restart budget can be varied from zero to six. If the restart budget is zero, the checkpoint budget is by definition also zero. As no restart is done, the use of a checkpoint is very limited. This results into 74 different fault tolerance patterns. There is one exception for the MP3 application, where the checkpoint budget is always zero. This is due to the fact that our model only models explicit checkpoints at communication events. As the MP3 application is modeled quite coarse grained, there are only two communication events per frame (read and write).

The fault tolerance patterns are meant to make the application fault tolerant. In our target architecture the four processors are the unreliable parts. Each of the processors has the same mean time to failure: 10^6 () FIT. Although one fault per 1000000 time units is relatively large, the use of such a high fault rate allows us



(a) *2 subnetworks.* **(b)** *3 subnetworks.* **(c)** *4 subnetworks.*

Figure 6.16: *Optimal subnetworks for a specific number of fault-tolerant subnetworks.*

to magnify the differences between the different fault tolerant mappings that are explored. As a result, the trends become more clear[1]. All the other components, like the communication buses and the NMR components are already fault tolerant. Important to note is that for each experiment the same seed is used for the exponentially distributed soft errors that occur during the system simulation. As a result, the same sequence of transient faults is used to compare the fault-tolerant mappings. After a complete generation, the seed is changed and another sequence of transient faults is used to evaluate the fault-tolerant mappings.

The experiments will use the fault tolerance patterns to transform the three applications into a fault-tolerant application and to map them onto the unreliable architecture. In the first experiment, the patternization of the MJPEG application is fully analyzed by showing some of the optimal patternizations and to observe some general trends. The next three sections will use SAFE to do exploration of the fault-tolerant design space. Only a search of a small part of the design space will be done for each application in isolation, as in this chapter still an exhaustive search is performed (the next chapter will introduce an efficient technique to explore the complete design space).

6.5.1 MJPEG Patternization

When making a fault-tolerant mapping, three steps are performed: patternization, binding and dispatching. To study the process of patternization, we performed an exhaustive patternization for the MJPEG application and an MPSoC where only TMR patterns are used. There are multiple instances of the TMR in the set of fault tolerance patterns, involving different choices for the restart budget and the checkpoint budget. By limiting the set of available fault tolerance pattern to the

[1]The fault rate that we use is currently only applicable to the extreme environments such as space applications, but with the decreasing technology scales the fault rate of the embedded systems at ground level are quickly increasing to these high fault rates [66]. For current systems, SAFE can perfectly be used after profiling the fault rate of the processors and to adapt the parameter in the architecture model.

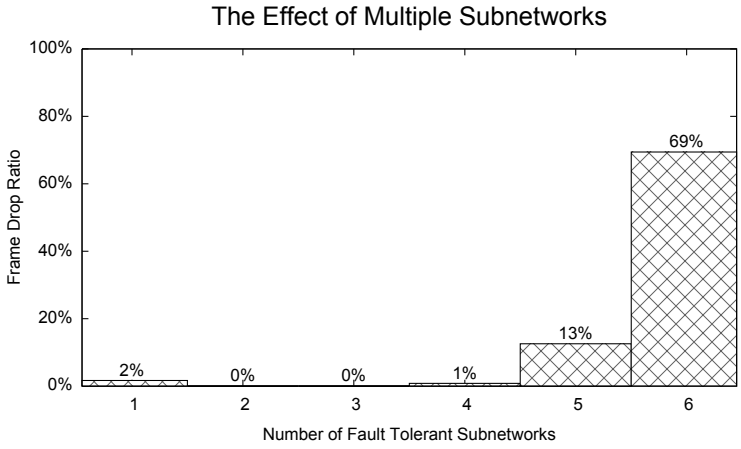


Figure 6.17: *The effect of different numbers of subnetworks on the frame drop ratio of the MJPEG application.*

TMR, the exhaustive search remains feasible.

Figure 6.16 shows the optimal patternization for a different number of fault-tolerant subnetworks. In this case, we have taken frame drop ratio as a primary objective and power consumption as a secondary objective. A first observation is that in this design space the patternization is incremental. By adding an additional subnetwork, one of the processes is moved into the new subnetwork. In the case of two subnetworks, the application is split into two equally sized subnetworks. Not only are these subnetworks equally sized, but also the number of external communication channels is kept minimal. Apart from the I/O communication channels (which are external by definition), only the channels (DCT, Q) and (DMUX, CONTROL) are external. Due to this minimum of external channels, the amount of majority voting (only done on external communication) is minimized.

When increasing the number of subnetworks to three, the DCT process is put into a separate subnetwork. The rationale is not the minimization of external communication, but the guarding of the compute intensive tasks. As the DCT is the most computationally expensive operation, it is beneficial to ensure that verified data is used in the computation. If it would have been unverified, it can be the case that unnecessary computation will be done. The same is true for the optimal patternization with four subnetworks. In this case, the quantization (Q) process is separated, being the second most compute intensive operation in MJPEG.

Having more fault-tolerant subnetworks may increase the quality of the application (with respect to frame drop ratio). However, it also increases overhead. This can be seen in Figure 6.17. Up to four subnetworks, the frame drop ratio reduces to 0 percent. With five or more subnetworks, the frame drop ratio quickly climbs up to

69 percent when each process is placed in a separate subnetwork.

In general, the patternization of the MJPEG application on the given architecture benefits from: 1) equally sized subnetworks, 2) a minimization of external communication channels and 3) the guarding of compute intensive tasks. These conclusions, however, only hold for this specific architecture. An architecture where, for example, the fault rate of the processors is much higher, may benefit from a larger fraction of external communication channels. Therefore, for such an architecture the fraction of external communication channels would be maximized instead of minimized in order to intercept faulty data as soon as possible.

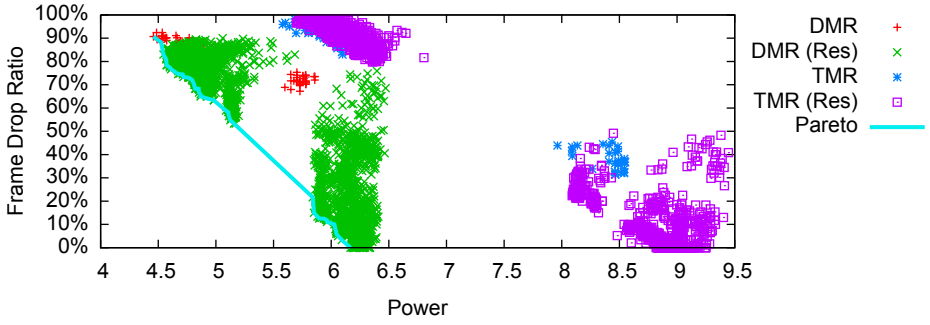
6.5.2 Power versus Frame Drop Ratio

Figure 6.18 shows the frame drop ratio and power consumption for all the evaluated design instances for the MJPEG, Sobel and MP3 applications. To keep the experiment feasible, only a part of the design space is explored. The fault-tolerant mapping can freely segregate the application into different subnetworks, but only one and the same fault tolerance pattern can be used for all the subnetworks. Given this patternization, all possible bindings are explored. Therefore, the design instances in Figure 6.18 are colored based on the fault tolerance pattern that is selected. Two decisions are highlighted: 1) the usage of DMR or TMR and 2) restarting enabled or not.

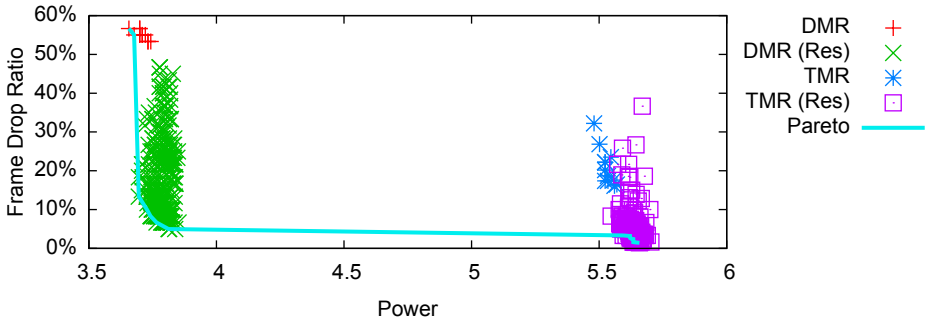
An obvious conclusion is that TMR requires more power than DMR. Both DMR with and without restarting take less power than the design instances where TMR is chosen as the fault tolerance pattern of a design instance. The exception to this case can be seen in Figure 6.18c where a DMR based design instance uses both more power and has a higher frame drop ratio than the TMR based design instances. This is an example of a restart policy that is poorly suited to the specific application because often a frame is restarted just before the deadline of the frame. Although the restart policy may manage to correctly fix the frame, the frame will arrive too late and will be dropped. In this way a lot of power is invested to restart frames without any potential gain (the frame drop rate is above 80%).

Since power is not the only objective, it depends on the application if it makes sense to use TMR as a fault tolerance pattern or not. In case of the MJPEG application (Figure 6.18a) the design instances that make use of DMR dominate all the design instances that use the TMR pattern. None of the points in the Pareto front (the blue line) of the MJPEG application use the TMR pattern. For the other two applications Sobel (Figure 6.18b) and MP3 (Figure 6.18c) the Pareto front contains TMR based design instances that, at the cost of a higher power usage, obtain a lower frame drop ratio. It is up to the designer if the investment in power is worth the improved frame drop ratio.

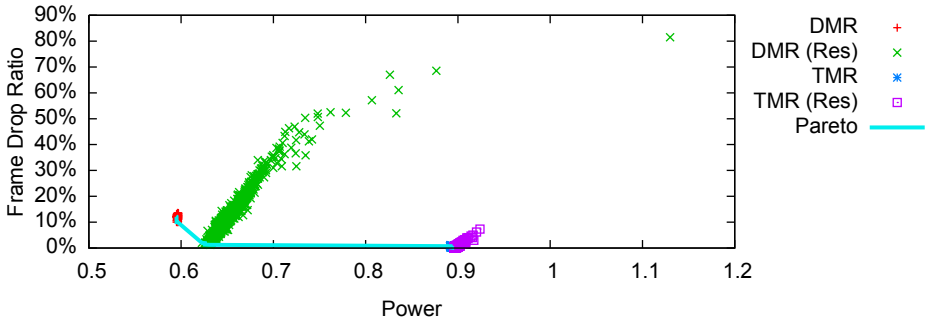
Figure 6.19 summarizes the gain of the different fault tolerance patterns by showing the optimal frame drop ratio given a type of fault tolerance pattern. From the different types of fault tolerance patterns, plain DMR (without restart possibilities)



(a) *MJPEG*



(b) *Sobel*



(c) *MP3*

Figure 6.18: *The evaluated design space of the three test applications.*

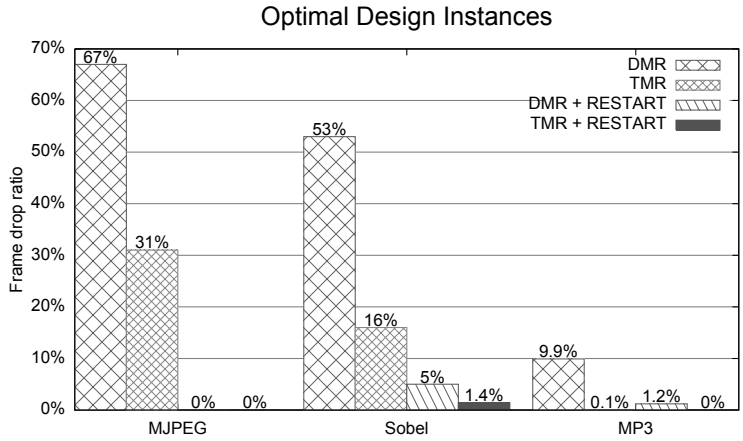


Figure 6.19: *The optimal frame drop rates differentiated over the type of fault tolerance pattern that is used.*

has the highest frame drop ratio. This is to be expected, as plain DMR can only detect the faults and drop the frames when one of the replicas has suffered a fault (DMR is not able to identify which one is correct). As a result, it has the lowest power usage. If, for example, halfway a frame a fault is detected, the rest of the frame does not need to be processed. As the plain DMR only detects faults, it also shows the vulnerability of the application to faults in the architecture. The MJPEG application is the most vulnerable with 67% of the frames being dropped, whereas the MP3 application only has 9.9% of the frames that are affected by a fault. These frame drop ratios, however, give a worst-case scenario. Due to our pessimistic assumption that each fault leads to an unusable frame (which is not always the case for applications like MJPEG), some of the unaffected frames may not be dropped in the real execution of the fault-tolerant design. As SAFE provides early design space exploration of the fault-tolerant designs, the relative performance of the fault-tolerant designs is the most important. After the early design space exploration, the remaining fault-tolerant designs can be investigated in more detail by lower level analysis tools.

In case an additional replica is introduced, a single fault can be masked. The plain TMR technique already greatly reduces the frame drop ratio for all the tested applications. For both the MJPEG application and the Sobel application the frame drop ratio is reduced by approximately 36% to 31% and 16% respectively. MP3 was already less vulnerable to faults in the architecture and with plain TMR almost no frames are dropped (only 0.1% frame drop ratio remains). So, if no restart is taken into account, TMR definitely pays off.

For the MJPEG application it means that in at most 31% of the frames at least

two of the replicas are affected by a fault while processing the same data packet. To reduce the number of faults that cannot be masked one could adapt the MJPEG application to divide the frame into smaller subblocks. In this case, less time is spent for processing individual data packets. Consequently, this leads to a smaller probability on faults at two different replicas. Another technique is to use the restart possibility to retry the processing of the frame. For the MJPEG application this completely prevents the dropping of frames for both the DMR and the TMR technique. As the TMR requires more power, without restarting the TMR is worse than the DMR based design instances. Just as restarting allows us to reduce the frame drop ratio of the MJPEG application from 31% to 0%, the Sobel application also benefits from the restart mechanism. With DMR the frame drop ratio drops to 5% and with TMR only 1.4% of the frames are dropped. This is considerably lower than the 16% of frames that are dropped when no restarting is available. For the MP3 application, however, restarting is not necessarily beneficial. The overhead of the restarting leads to a frame drop ratio of 1.2% when restarting is used in combination with the DMR technique. This is slightly larger than the frame drop ratio of plain TMR that was able to reduce the frame drop ratio to almost zero. As TMR masks almost all the faults, adding the restarting possibility does not significantly affect the design instances. This is true for both frame drop ratio as well as energy (In Figure 6.18c the plain TMR design instances are hardly visible as they almost completely overlap with the design instances that use TMR with a restarting possibility).

We already mentioned that restarting requires overhead with respect to cycle time. This can clearly be seen in the case of the MJPEG application (Figure 6.18a). The design space of the evaluated MJPEG design instances shows two clusters for each of the fault tolerance types: on with a high and one with a low frame drop ratio. These clusters are not present in the design space of the Sobel and MP3 applications. Further analysis on the MJPEG application revealed that when the power consumption of the fault-tolerant design becomes below 5.8 the frame drop ratio quickly grows from 50% to values above 70%. In these cases, the fault handling capability of the MJPEG encoder is too low to process the frames correctly in the given timing requirements. As a result, the encoder cannot keep up pace with the incoming frames. Hence, many frames are skipped as the deadline of the frame is already passed before starting the processing of the frame. This higher number of unprocessed frames has a positive side effect: even less power is consumed.

To conclude, both the MJPEG and the Sobel applications can greatly benefit from the restart mechanism. Using the restart mechanism, the MJPEG application does not even need TMR to prevent the dropping of frames. The MP3 application is only negatively affected by the restart mechanism, as can be seen by the shape of the cluster of design instances that use DMR with the restart mechanism (Figure 6.18c). In this case, the power and cycle time that is spent to make the application more fault-tolerant only leads to a higher frame drop ratio. The way the fault tolerance overhead influences the frame drop ratio is discussed in the next experiment.

6.5.3 Breakdown of Frame Drop Ratio

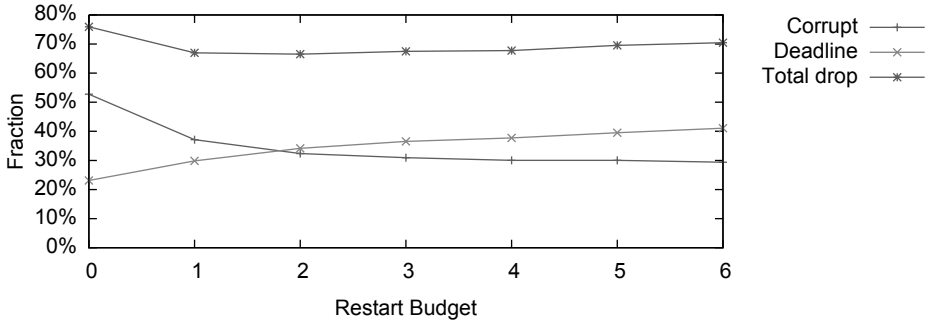
Dropping a frame can have several causes, which is illustrated in Figure 6.20. First, there are corrupted frames. In these frames, a fault is detected (due to a transient error), but the restart budget was too small to correct these faults. Second, there are deadline misses. In these cases, the application is too late to retrieve or deliver a frame from / to the OWP process. For the experiment in Figure 6.20, we have taken all the design points of the previous experiments where the explicit checkpoint budget was zero. These design points are differentiated by restart budget and for each category the average frame drop ratio is given. Not only the total drop ratio is given, but also the fractions that are due to corrupt frames and deadline misses.

A larger restart budget can both improve the frame drop ratio (less corrupted frames) and degrade the frame drop ratio (more deadline misses). The more effort is put in fault correction, the lower the number of corrupted frames. However, the effort has a negative effect on the number of deadline misses. The optimal restart budget is thus application dependent. For all our applications, the gain in the reduction of corrupted frames is overshadowed by the increase of deadline misses at about one or two restarts per frame. However, the MP3 application is not influenced anymore once the restart budget is above three. In this case, the MP3 application is already able to circumvent corrupted frames and the additional restarts will not be used.

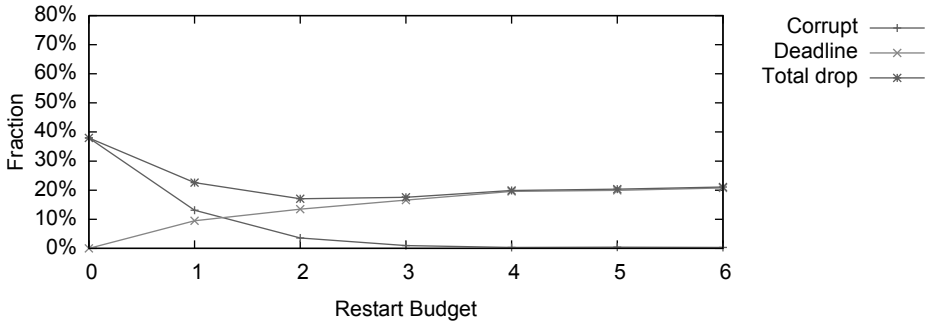
Clearly, the MJPEG application is more challenging with respect to fault-tolerant mapping than the Sobel and MP3 application. The high average number of deadline misses show that the requirements of this application are quite stringent. On average, the fault-tolerant mappings do not seem to have sufficient slack time for utilizing the complete restart budget to correct the faults within the corrupt frames.

6.5.4 Buffer Requirements

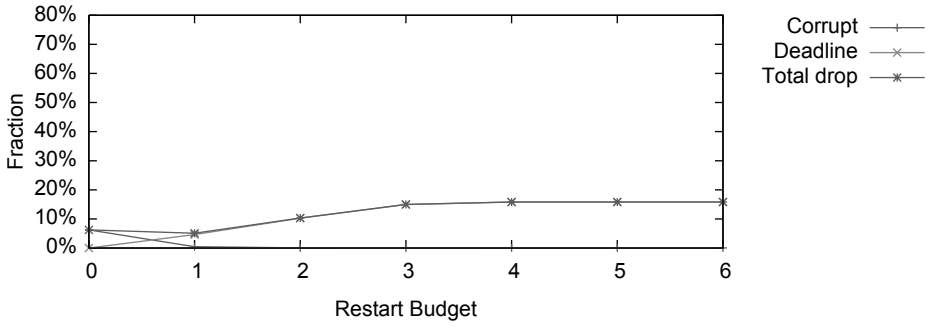
Earlier experiments clearly showed the trade-off between the advantages and disadvantages of the fault tolerance patterns on the frame drop ratio of an application. Potentially, fault tolerance patterns consume more power, but, depending on the application and architecture, this may improve the reliability of the design. One of the costs of fault tolerance patterns, however, has not yet been studied. Although restarting and the explicit checkpoints improve the frame drop ratio, it also involves an additional cost with respect to buffer requirements. To show these requirements, the best design instances for the MJPEG and Sobel applications have been selected for a checkpoint budget between zero and six. For all of these design instances restarting was enabled, as without restarting checkpoint buffers are not required. The outcome of this experiment is shown in Figure 6.21, with a horizontal axis that shows the checkpoint budget and a vertical axis showing the normalized buffer size (Figure 6.21a) or the frame drop ratio (Figure 6.21b). Normalization of the buffer sizes is done per application (i.e., the buffer sizes of the different applications are incomparable) such that the maximal buffer size is 1.0.



(a) *MJPEG*

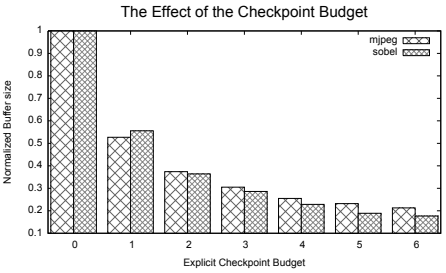


(b) *Sobel*

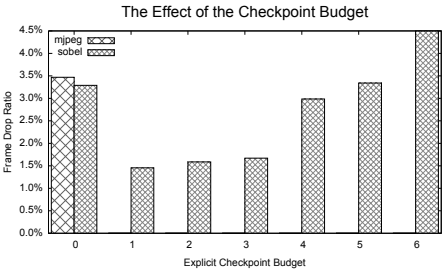


(c) *MP3*

Figure 6.20: *The frame drop ratio versus the size of the restart budget.*



(a) *Buffer requirements*



(b) *Frame drop ratio*

Figure 6.21: *Varying the checkpoint budget with restarting enabled.*

Figure 6.21a shows the relation between the explicit checkpoint budget size and the required buffer size. When explicit checkpointing is disabled, the system still takes implicit checkpoints. For these implicit checkpoints, the message cache (see Section 6.4.3) must contain all the incoming data messages since the last complete implicit checkpoint. As a result, the size of the message cache quickly grows. To reduce the size of the message cache, explicit checkpoints can be taken that will periodically flush the message cache. The more often an explicit checkpoint is made, the lower the maximal size of the message cache. Still, explicit checkpoints also need to be stored. The required size of the checkpoint buffer, however, is significantly smaller than the message cache and less influenced by the frequency of taking checkpoints (not shown in the graph). Combining these two trends, the required buffer size only reduces whenever the explicit checkpoint buffer size increases.

Although a larger explicit checkpoint budget may lead to a reduced buffer size, it will have a negative effect on other system objectives. Figure 6.21b shows how the explicit checkpoint budget affects the frame drop ratio. The MJPEG application can take up to six checkpoints per frame without any misses, but the Sobel application suffers from dropped frames. Initially, from an explicit checkpoint budget from zero to one the frame drop ratio is reduced since less work needs to be redone upon a restart. With two or more explicit checkpoints per frame, however, the overhead of taking the explicit checkpoints leads to a larger amount of dropped frames due to a higher amount of deadline misses.

6.6 Related Work

Due to the technology scaling, reliability becomes a major design objective [66, 46] for the embedded system world. A lot of research is dedicated to fault-tolerant embedded processors that, for example, increase reliability by replicating internal hardware [20, 27] or by making use of idle components [7, 49]. Introducing reliability, however, potentially has a major effect on other objectives. This effect must be quantized to allow the designers to reason about the amount of reliability they are willing to

put into the system and how this reliability will affect the different objectives of the systems. It is therefore important that reliability is explored as a separate design objective [24] and not hierarchically.

Therefore, DSE environments must be reliability aware. Most of the research in reliability aware DSE environments [3, 11, 28, 29, 32] is based on static task scheduling. Generally, these environments make use of task replication and re-execution to detect and handle faults for different fault scenarios. A fault scenario is a certain sequence of faults that occurs at specific processing elements. The approach in [32] uses a Tabu search to find a static fault-tolerant schedule that is able to make the application deadlines. It will automatically replicate and re-execute tasks to cope with the worst case fault scenario. Just as COFTA [11], this approach does not give reliability as an outcome of the DSE. The approaches of [28, 29] and [3] calculate the reliability of a static schedule of an application by determining the set of fault scenarios for which the schedule will meet the deadlines of the application (for [28, 29] this set of scenarios is encoded into a system fault tree and [3] calls this set of scenarios a working set). Next, the probability for each fault scenarios is combined to obtain the reliability of the static schedule. All of these approaches only consider sets of independent tasks, although COFTA [11] provides a trade-off between fault tolerance overhead and reliability by clustering tasks in different groups.

The method of Bolchini [6] provides an analysis framework that is capable of performing reliability analysis for an embedded system where dynamic scheduling is used. This analysis tool provides, given a fault scenario and an application schedule, a classification of how the faults affect the system. Examples of a classification scheme are no-effect, safe or dangerous.

One of the main novelties of SAFE is that it provides exploration of the fault tolerance patterns in a transparent fashion. Due to the separation of concerns, the fault tolerance patterns are completely separated from the application model and the architectural model such that the set of fault tolerance patterns can be extended. On top of that, SAFE is capable of fully simulating the complete embedded system with dynamic scheduling on a high abstraction level. As a result, the framework is capable of providing the reliability of the system *next to the other potential objectives of the system*. As a part of this system a unique fine-grained and parametrized checkpoint model is included. Most of the other fault-aware DSE frameworks only take implicit checkpoints (the strict separation between frames) into account [41]. This provides a unique view on all the effects of the provided fault tolerance patterns. Not only the way that the fault influences the system is classified, but also the effect of faults and fault tolerance patterns on all the objectives can be observed. This can be the buffer requirements, but also the number of dropped frames and the reason why the frames are dropped.

Additionally, SAFE operates on application networks instead of a set of independent tasks that is used by the aforementioned frameworks. In contrast to the clustering technique of COFTA, the clustering of processes that are part of a process

network takes the dependencies of tasks into account to leverage the overhead of the fault tolerance patterns. As a result, a real fault-tolerant DSE can be performed.

6.7 Conclusion

In this chapter SAFE was introduced that provides a framework to facilitate the fault-tolerant design of embedded systems. Fault-tolerant design takes into account the dynamism in the architecture with respect to transient faults that can occur. These faults lead to unreliability due to erroneous application outcomes. To make the output of the application more reliable, fault tolerance patterns are used. An example of a fault tolerance pattern is active redundancy where multiple replicas are used to run the program. By voting on the outcome, the detected faults can be masked or handled in other ways like skipping the frame or restarting it. Strictly, a fault tolerance pattern can be seen as a transformation on an application to introduce techniques to detect and handle faults. The policies (like the frequency of taking checkpoints) used during the fault detection and handling are also part of the fault tolerance pattern.

A fault-tolerant mapping automatically segregates the application into subnetworks, applies fault tolerance patterns on the subnetwork and binds the fault-tolerant application onto the architecture. By exploring the fault-tolerant mappings, the trade-offs of different fault-tolerant designs can be analyzed. Fault tolerance patterns may, for example, reduce the number of frames that need to be skipped or restarted due to faults, but they also introduce overhead. This overhead can be in buffer space for checkpoints, but also processor cycles that may lead to dropped frames due to deadline misses. Our experiments showed that the optimal fault-tolerant mapping is completely application dependent: for our Sobel and MJPEG application, for example, a lower frame drop ratio was achieved when the restarting mechanism was applied, whereas the frame drop ratio of the MP3 application was hardly affected. As a result, in case of the MP3 application the overhead in power consumption due to the restarting mechanism did not improve the fault-tolerant mapping.

This means that an automated way of exploring all fault-tolerant mappings is crucial to gain insight in the way fault tolerance patterns can improve applications that are mapped onto unreliable architectures. In the next chapter, the DSE framework will be presented that allows for performing a real automated exploration. Additionally, we take a closer look at the dynamism in the architecture and how this can play a role during the automated DSE.

7 Fault-Tolerant DSE

The *Sesame Automated Fault-tolerant Explorer (SAFE)* is a framework that takes fault tolerance into account while mapping the application onto the architecture. Fault tolerance patterns are automatically applied to the application, which makes an automatic exploration of the fault-tolerant design space possible. Such a fault-tolerant design space, however, involves a large degree of freedom with respect to the number of objectives that can be explored. Not only performance, power and cost are part of these objectives, but also metrics like frame drop ratio and additional buffer space that is required to store application checkpoints. For this frame drop ratio, it can even be analyzed which part is due to transient faults and which part is due to deadline misses. Such detailed statistics are extremely usable to extensively analyze a small set of fault-tolerant mappings by hand.

During early DSE, however, there are simply too many mapping possibilities to evaluate them by hand. This is already true for the traditional mapping problem that is handled by Sesame, but it becomes even worse when fault tolerance is taken into account. For a fault-tolerant mapping, not only binding needs to be done (like is the case with the traditional mapping problem), but also the patternization that divides the application into subnetworks and selects fault tolerance patterns. There are many potential ways of performing patternization and, due to the larger fault-tolerant application, there are even more possibilities to bind the application onto the architecture.

In Figure 7.1 the Ψ-chart is shown that was introduced in the previous chapter (Figure 6.8). This chart shows the general technique that can be applied to explore the fault-tolerant design space. Given the applications, the fault tolerance patterns and the architecture, a fault-tolerant mapping can be made that is evaluated using SAFE. Based on the performance numbers of the design instance (like performance, power or frame drop ratio of the given fault-tolerant mapping), the input of SAFE can be adapted. One can adapt the applications, the fault tolerance patterns or the architecture, but in this thesis we will only focus on changing the fault-tolerant mapping. The manipulation of the fault-tolerant mapping can be automated, whereas a change in the architecture (adding for example a processor, because all the processors are over-utilized) cannot easily be automated.

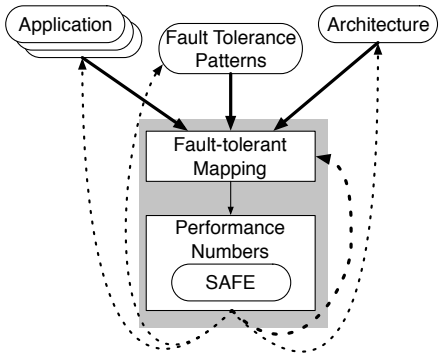


Figure 7.1: *The Ψ-chart that shows an exploration method the fault-tolerant design space can be explored by using SAFE.*

Apart from the automatic exploration of the fault-tolerant mapping, it is also important to focus on the dynamism of the architecture. The previous chapter only took the fault-tolerant mapping into account. For the SAFE simulations, the faults were injected randomly into the architecture. This sequence of injected faults was only generated once per generation and used for all the fault-tolerant mappings that needed to be compared. However, these injected faults are also dynamic in nature. As we already discussed in the related work of SAFE (Section 6.6), some approaches capture the dynamics of the transient faults into fault scenarios that are used to obtain the reliability of a system. On itself, SAFE cannot yet take reliability into account, apart from the frame drop ratio of a single specific fault scenario. Therefore, this chapter will utilize so-called architecture scenarios (i.e., fault scenarios) to describe how reliable the obtained objectives are of a given fault-tolerant mapping. This chapter starts by giving an overview of the fault-tolerant DSE framework. Next, the two main components of the fault-tolerant DSE framework are discussed in Sections 7.2 and 7.3. Finally, the chapter will be closed by two case studies and a conclusion.

7.1 Overview

In Section 5.1, we already introduced a framework to statically explore the design space to take the dynamism of application scenarios into account. In this chapter we do something similar, only in this case we are not dealing with application scenarios, but with architecture scenarios. An architecture scenario defines the sequence of independent transient faults of an architecture that will occur with the time and place of each individual fault. Not only influence architecture scenarios the quality of the design instances (as was the case with the application scenarios in Chapter 5), but a fault-tolerant mapping must also be able to cope with faults that are listed

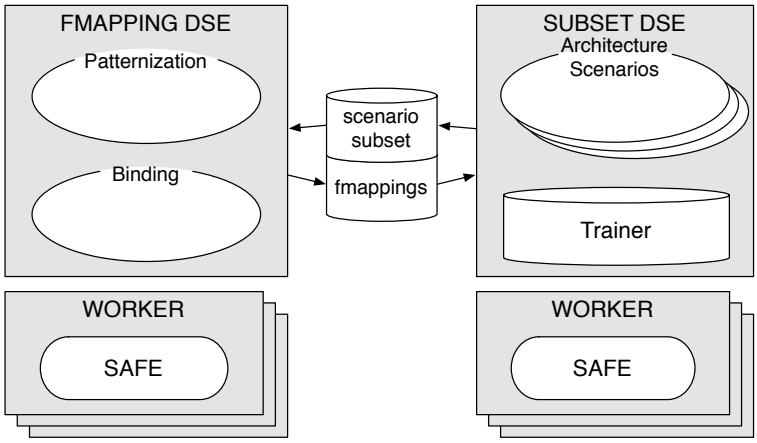


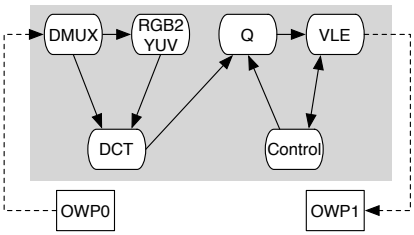
Figure 7.2: *The overview of the fault-tolerant DSE exploration framework.*

in the architecture scenario. More detail on architecture scenarios will be given in Section 7.3.

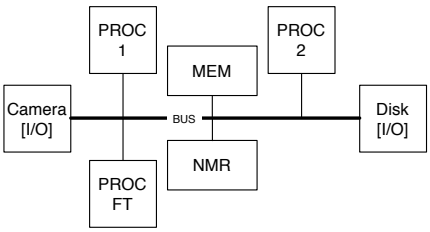
Figure 7.2 shows the overview of the fault-tolerant DSE framework. Conceptually, this framework shows similarities to the framework of Chapter 5 that deals with application scenarios. There are two main components: the fault-tolerant mapping DSE (or shortly FMapping DSE) and the subset DSE. Similarly to Chapter 5, the FMapping DSE explores the different design instances given the subset of scenarios. This subset of scenarios is selected by the subset DSE that tries to select a representative subset of scenarios based on the current set of design instances in the FMapping DSE.

The FMapping DSE explores the complete range of possibilities of a fault-tolerant mapping. In contrast to a normal mapping, this not only includes the binding of the application processes onto the architecture, but also the patternization of the application that transforms the application into a fault-tolerant application (for further detail see Section 6.3). Corresponding to the two hierarchical steps during a fault-tolerant mapping, the FMapping DSE uses two populations: a patternization population and a binding population. The patternization population defines the fault-tolerant application, whereas the binding population maps these fault-tolerant applications onto the architecture.

In the subset DSE, a representative subset of architecture scenarios is selected. Just as Chapter 5, this raises the question "What is representative?". To answer this question, one must consider what a designer wants to know when it designs a fault-tolerant embedded system. In our perspective, this is how resilient the embedded system is with respect to transient faults in the architecture (i.e., the architecture scenarios). Resilience in this case means that the rate of change of the objectives of a system when the number of transient faults increases. Ideally, one wants a



(a) *MJPEG Application*



(b) *Destination Architecture*

Figure 7.3: *The MJPEG application and the architecture that is used to illustrate the chromosome design of the fault-tolerant DSE.*

system that is affected by these transient faults as least as possible. Therefore, the fault-tolerant DSE will explore the design space given a subset of architecture scenarios with different rates of transient faults. The subset selector will identify these architecture scenarios such that the worst-case fitness is found for each transient fault rate (as will be discussed later in Section 7.2.3).

The complete framework is implemented using a combination of C++ and the *Message Passing Interface (MPI)*. To evaluate the fault-tolerant mappings both the FMapping DSE and the subset DSE make use of a dedicated set of SAFE workers. These workers are implemented in a separate MPI process. Due to the coarse grained nature of SAFE jobs (processing a single job will normally take a couple of seconds); the combination of C++ and MPI is efficient enough to quickly explore the fault-tolerant design space.

As heuristic search method, both the FMapping DSE and the subset DSE use a modified GA. This modification is elaborated in the next two sections where the two main components are described in more detail. To clarify these explanations both of the sections will use the MJPEG application in Figure 7.3 as an example of an application that must mapped onto an architecture in a fault-tolerant fashion to optimize the different objectives of the embedded system. This example, which was used earlier in the previous chapter, searches for a fault-tolerant design of a MJPEG encoder that is mapped onto an architecture with two unreliable processors (PROC-1 and PROC-2). To achieve this, two fault tolerance patterns are available: a fault-tolerant processor (PROC-FT) or active redundancy that replicates the computation on the unreliable processors in time and / or space after which it checks the output of the different replicas with a majority voter (the NMR component).

7.2 Fault-Tolerant Mapping DSE

A fault-tolerant mapping describes the complete transformation of an application to a mapped fault-tolerant application. This involves three steps (Section 6.3): 1) patternization, 2) binding and 3) dispatch. The patternization segregates the applica-

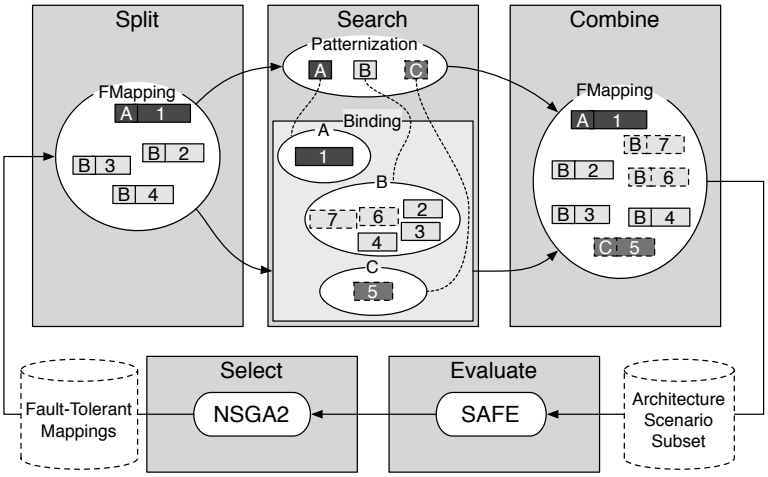


Figure 7.4: *The dimension oriented approach to co-explore the patternization and the binding of the fault-tolerant mappings. Gray scale is used to differentiate between the different patternizations of the mappings, whereas a dashed line type is used to emphasize the offspring mappings.*

tion into distinct subnetworks and applies fault tolerance patterns on the individual subnetworks. During the binding, these subnetworks are mapped onto the architecture together with the OWP processes of the application. Finally, the message dispatch determines the routing of the data that is exchanged between the different processes. As discussed earlier, the message dispatch remains fixed and, therefore, the FMapping DSE only needs to explore the patternization and the binding.

Basically, the patternization groups the processes of the application into sets to optimize the fault-tolerant mapping. To address these kind of grouping problems, traditional GAs do not suffice. Therefore, the exploration of patternizations uses a special representation for the chromosomes where the group part and the object part are separated [17]. The object part assigns the processes to the subnetworks, whereas the group part selects the fault tolerance patterns. For most of the grouping problems, the fitness of the partitioned objects can directly be obtained. One example is the bin-packing problem that can directly determine if the total size of the objects in a bin does not violate its constraints. Combined with a cost function that promotes the minimization of the number of bins, the search becomes straightforward. For our patternization problem, however, the fitness can not directly be obtained. Given a patternization, the application is already fault tolerant, but without a binding the quality (like energy, frame drop ratio, etc.) of the patternization cannot be evaluated. Therefore, the fault-tolerant DSE uses a dimension-oriented approach [33] to simultaneously co-explore both the patternization and the associated binding. This approach is illustrated in Figure 7.4 and it consists of five steps:

1) Search: The search step is responsible for creating offspring for the next generation of the genetic algorithm. It maintains a separate population for both the patternization and the binding. As a fault-tolerant mapping is a combination of both, each of the binding chromosomes is linked to a patternization chromosome. This patternization chromosome defines the transformed fault-tolerant application and, thus, the processes that need to be bound to the architecture. As it is hard to completely judge about a patternization using only a single binding, there is a one-to-many relation between the patternization and bindings. Therefore, a single patternization is quite likely to have several bindings in the binding population. This is also illustrated in Figure 7.4. Each of the patternizations has a different gray scale. Additionally, the binding population consists of multiple subpopulations: one for each patternization. Patternization chromosome A has only one associated binding chromosome (number 1) in its subpopulation, whereas the subpopulation of patternization chromosome B has five linked binding chromosomes (numbers 2, 3, 4, 6 and 7).

Creating offspring for the GA is done separately for both the patternization population and the binding populations that belongs to each of the patternizations. For this purpose, specialized genetic operators are used (as will be discussed later in Section 7.2.2) to create slightly manipulated individual chromosomes that are based on the selected chromosomes. For the binding chromosomes these genetic operators are sufficient to create the offspring, but to make the offspring of the patternization population complete a subpopulation of binding chromosomes must be generated. Without a subpopulation of binding chromosomes the patternization chromosome cannot be evaluated since only complete fault-tolerant mappings can be evaluated. One option would be to randomly generate matching binding chromosomes, but, quite likely, this will result into fault-tolerant mappings of a low quality. To improve the probability that an offspring patternization chromosome survives it to the next generation, the generated binding chromosome is based on an existing binding chromosome. This binding chromosome is selected using the binding chromosome that belong to the original patternization chromosome (the chromosome on which the patternization offspring is based). To make a valid binding for the new patternization, the changes in the patternization are reflected in the binding chromosome. If, for example, the patternization adds an additional replica to a fault-tolerant subnetwork, the binding is extended with an extra gene that maps this additional replica. Similarly, when a replica of a fault-tolerant subnetwork is removed, the gene binding of this replica will also be removed from the binding chromosome.

Figure 7.4 shows the offspring chromosomes using a dashed border. Chromosomes 6 and 7 are offspring chromosome from the binding population that are based on the parent chromosomes 2 and 4. There is only one offspring chromosome in the patternization population: chromosome C with parent chromosome A. The patternization chromosome also has an associated binding chromosome to complete the new fault-tolerant mapping. Adjusting the binding chromosome 1 of the parent chromosome

A creates the new binding chromosome 5.

2) Combine: The next step is to combine all the patternization and binding chromosomes to create fault-tolerant mappings that can be evaluated using SAFE. Since each of the binding chromosomes is linked to exactly one patternization chromosome, each of the binding chromosomes is transformed into a fault-tolerant mapping by combining it with the associated patternization chromosome.

3) Evaluate: Combined fault-tolerant mappings are evaluated with SAFE using a representative subset of architecture scenarios. Based on the architecture scenarios that are obtained from the subset DSE (see Figure 7.2), the fitness of each of the fault-tolerant mappings can be obtained.

4) Select: Using the fitness of the fault-tolerant mappings, the parent for the offspring of the next generation is selected. For this purpose, the NSGA-II [12] is used.

5) Split: As there are two separate populations, the selected parents cannot directly be used to select the offspring for the search step. Therefore, the evaluated fault-tolerant mapping population is split into two populations again. Each of the chromosomes is split in its patternization and binding part. The binding chromosomes can directly be used to produce the binding population. For the patternization chromosomes, however, the redundant copies of the patternizations are removed first. In Figure 7.4, for example, patternization chromosome B is only added once to the population.

Next, the selected parents for offspring are split into two parts. A small set of them will be used to create offspring in the patternization population and the rest will be used for the offspring in the binding population. In the offspring example given in the search step, fault-tolerant mapping chromosomes $A1$, $B2$ and $B4$ were selected as parents. Chromosome $A1$ was assigned to the patternization offspring (which applied genetic operators on patternization chromosome A and adjusted binding chromosome 1 to match the changes in A). The other two chromosomes $B2$ and $B4$ were used for the binding offspring. This means that patternization chromosome B is kept intact and the genetic operators are only applied on binding chromosomes 2 and 4.

7.2.1 Chromosome Representation

One of the aspects that affect the search efficiency of the GA is the chromosome design. In the introduction of this section, it was already emphasized that the grouping of the processes must carefully be encoded into the patternization chromosome. These patternization chromosomes, however, only describe the grouping of application processes into fault-tolerant subnetworks. A binding chromosome is required to map these fault-tolerant subnetworks onto the architecture. In this subsection, we will describe both of the patternization and binding chromosomes using a fault-tolerant mapping of the MJPEG application in Figure 7.3. More specifically, we

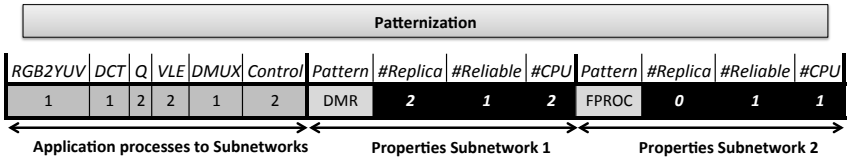


Figure 7.5: *An illustration of a pattern chromosome for the MJPEG application.*

describe the chromosomes that result into the fault-tolerant mapping that is shown in Figure 6.12 on page 142.

At first, the application is segregated into a set of fault-tolerant subnetworks. For each of these subnetworks, a fault tolerance pattern is selected. With respect to the design of the GA, this resembles a global grouping problem like bin packing or graph coloring [17]. Therefore, the chromosome consists of two parts: the object part and the group part. In the object part the application processes are assigned to subnetworks. To describe the properties of these subnetworks, the group part is used. An example of a patternization chromosome for the MJPEG application is given in Figure 7.5. The first six genes are the object part where, for each of the application processes (RGB2YUV, DCT, Q, VLE, DMUX and CONTROL), the number of the used subnetwork is encoded. This grouping must comply with the conditions of a fault-tolerant mapping that was given in Section 6.3. One of the most important conditions that must be enforced is that all subnetworks are weakly connected (Equation 6.7).

Properties of the subnetworks are encoded in the group part of the chromosome. Strictly seen, a single gene per group that encodes the fault tolerance pattern of the subnetwork (the pattern field in Figure 7.5) is sufficient. For efficiency purposes, however, some derived information of the fault tolerance pattern is also encoded in the patternization chromosome. This derived information includes: 1) the number of replicas of the fault tolerance patterns (#Replica), 2) the number of potential reliable architecture elements that can be used for mapping the voter process (#Reliable) and 3) the number of architectural processor elements that can be used to map the replica processes (#CPU). For every fault tolerance pattern $f \in F$, these numbers are derived as follows:

$$\#\text{Replica}(f) = n_{\text{proc}}(f) \tag{7.1}$$

$$\#\text{Reliable}(f) = |\{r | r \in R_R \wedge \text{feasible}(f, r)\}| \tag{7.2}$$

$$\#\text{CPU}(f) = |\{r | r \in R_P \wedge \text{feasible}(f, r)\}| \tag{7.3}$$

As an example, the patternization of the MJPEG application that is encoded in the chromosome in Figure 7.5 is the same as the patternization shown in Figure 6.9. In the object part the grouping into two subnetworks is encoded: subnetwork 1 (RGB2YUV, DCT and DMUX) and subnetwork 2 (Q, VLE and CONTROL). Subnetwork 1 is guarded using a DMR pattern. DMR has two replicas and for the

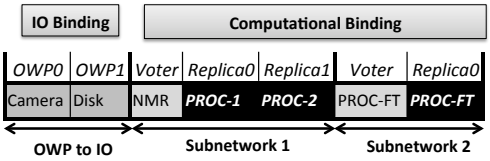


Figure 7.6: *An example of a binding chromosome based on the patternization shown in Figure 7.5*

architecture in Figure 7.3b it has one feasible reliable architecture element for the voter and two feasible processor elements (for the definition of the feasible function see Table 6.1). Similarly, the second subnetwork uses a fault-tolerant processor that has one replica, one feasible reliable architecture element and one feasible processor architecture element.

One of the main purposes of including the derived properties of the subnetwork inside the patternization chromosome is that the binding chromosome can easily be manipulated. Only the linked patternization chromosome is required during manipulation. Just as the patternization chromosome, the binding chromosome consists of two parts. The first part encodes the IO binding, whereas the second part describes the computational binding. At first, the IO binding maps each OWP process onto the architecture. Therefore, it is completely independent of the patternization chromosome. The computational binding, on the other hand, is completely based on the group part of the patternization chromosome. For each subnetwork, a set of genes maps the voter process and the process replicas. From these genes, the first gene encodes the reliable component onto which the voter process is mapped using an integer that addresses one of the reliable components (Equation 7.2). All the remaining genes address one of the processor elements (Equation 7.3) that are used for binding the specific replica. Consequently, the total number of genes per subnetwork is one more than the number of replicas (Equation 7.1).

An example of the binding for the MJPEG application is given in Figure 7.6. Starting from the MJPEG patternization in Figure 7.5, a corresponding binding is created that is equal to the binding shown in Figure 6.10. For the IO binding there is only one feasible mapping: for OWP0 the Camera component is used and for OWP1 the Disk component is used. Since the first subnetwork has two replicas, the next three genes encode the binding of the first subnetwork. The first gene selects the NMR component as the reliable element, whereas the next two genes select processors PROC-1 and PROC-2 for the replica processes. In the second subnetwork, a fault-tolerant processor is used. The binding of such a pattern is quite straightforward. As the processor is already reliable, the pattern only requires a single element: the PROC-FT processor that is used both as reliable and as processor element.

7.2.2 Genetic Operators

After defining the chromosome representations, the genetic operators need to be defined. For the binding chromosome, the traditional implementations can be used (see Section 2.2) as the chromosome uses a list of integers that can freely be manipulated. Within this chromosome, the alleles are defined by the feasibility function (for the IO binding) or by the linked patternization chromosome (for the computational binding). We should note that Figure 7.6 does not show integers, but the names of the architectural elements. This is purely done for visualization purposes.

For the patternization chromosomes specialized genetic operators are used. Important of these genetic operators is that they operate on the group part and not on the object part. Operating on the group part does not mean that the object part remains the same. A change in a group can result in the change of some of the references in the object part. The main reason for using a group-based genetic operator is that it improves the efficiency of the genetic algorithm [17]. Firstly, the group-based mutation operator can perform one of the following actions:

- Use a different fault tolerance pattern for a subnetwork.
- Move a subset of the nodes of a subnetwork to another subnetwork (existing or new).

Important is that after applying the specialized mutation operator the patternization chromosome may need to be repaired. At first, the derived group properties are updated. Secondly, there is verified if all the subnetworks are weakly connected. In case the nodes in a subnetwork are not weakly connected, the subnetwork is split into multiple subnetworks such that each of the individual subnetworks becomes weakly connected.

Our specialized uniform crossover operator is currently only defined to apply crossover for a multi-application workload. To be certain that the chromosome remains valid, only the complete set of subnetworks per application is exchanged during a crossover operation. Due to the weakly connected subnetworks, it is impossible that within a subnetwork processes of different applications are present. Therefore, on exchanging the complete set of subnetworks of an application the patternization chromosome will always remain valid.

7.2.3 Fitness Function

The fault-tolerant DSE provides a design space exploration for different rates of transient faults. To achieve this, the fitness function of a fault-tolerant mapping evaluates the fitness of different *reliability classes*. A reliability class characterizes the maximal probability of the architecture scenarios that are considered while determining the quality of the fault-tolerant mapping. For each of the different reliability classes, the worst-case objective is minimized. Consequently, the complete optimization problem can be posed as follows:

Architecture Scenario	Probability	Power	Dropping (ratio)
s_1	0.3	0.5	0.0
s_2	0.4	0.4	0.1
s_3	0.4	0.7	0.1
s_4	0.7	0.7	0.15
s_5	0.7	0.6	0.2
s_6	0.8	0.6	0.25
$c =$	$[c_1, c_2] = [0.5, 0.9]$		
$F(m) =$	$[F^{c_1}_{\text{Power}}, F^{c_1}_{\text{Dropping}}, F^{c_2}_{\text{Power}}, F^{c_2}_{\text{Dropping}}]$		
$=$	$[0.7, 0.1, 0.7, 0.25]$		

Table 7.1: *An example of the fitness calculation of a fault-tolerant mapping for two reliability classes 0.5 and 0.9.*

$$\underset{m}{\text{minimize}}\ F(m) = [F_1^{c_1}(m), ..., F_k^{c_1}(m), ..., F_1^{c_n}(m), ..., F_k^{c_n}(m)] \tag{7.4}$$

$$F_i^c(m) := \max(\{\text{SAFE}_i(m, s) | s \in S \wedge \text{Prob}(s) \leq c\}) \tag{7.5}$$

When a fault-tolerant mapping m is optimized for k different objectives (like energy or frame drop ratio), the worst-case objectives are obtained for each of the n reliability classes independently (Equation 7.4). In order to obtain the worst-case objective i for reliability class c, the fault-tolerant mapping m is evaluated with SAFE to find the maximal value of the i-th objective for all the architecture scenarios that belong to the reliability class c. As Equation 7.5 shows, this is done by simulating the fault-tolerant mapping m with SAFE for each of the architecture scenarios that have a probability lower or equal to the probability of reliability class c. After the SAFE simulation, the maximal value of the objective is determined.

An example of a fitness calculation of a fault-tolerant mapping is given in Table 7.1. This mapping is evaluated for 6 architecture scenarios for the two objectives power consumption and frame drop ratio. Before starting the fault-tolerant DSE, it is decided that two reliability classes will be investigated: one for architecture scenarios with a probability of 50 percent or lower (c_1) and one for architecture scenarios with a probability of 90 percent or lower (c_2). By definition, reliability classes with a lower probability are contained in reliability classes with higher classes ($c_1 \leq c_2 \iff S^{c_1} \subseteq S^{c_2}$). As set of the architecture scenarios belonging to c_2 is a superset of the architecture scenarios of c_1, the minimal worst-case objectives for the reliability class c_2 is also always larger or equal to the objectives of reliability class c_1. For our example architecture scenarios s_1, s_2 and s_3 belong to reliability class c_1. The worst-case power for these scenarios is 0.7. Similarly, the worst-case frame drop ratio of reliability class c_1 is 0.1. Reliability class c_2 contains all the listed

architecture scenarios. As we will see in the next section, a higher probability of an architecture scenario corresponds to a higher fault rate. This means that, generally, the objective values are worse. In our example, the worst-case power is not affected and remains 0.7. The frame drop ratio, however, has grown to a ratio of 0.25.

7.3 Subset DSE

The fitness calculation of the fault-tolerant DSE shows a similar problem as the scenario-based DSE of Chapter 4 and Chapter 5 where the mapping fitness was calculated for the different application scenarios in the multi-application workload. During the early DSE, it is not feasible to evaluate all the possible scenarios. For the architecture scenarios it is not even possible to exhaustively evaluate a single fault-tolerant mapping for all architecture scenarios. There are simply too many scenarios in which transient faults can influence the architecture. That is why the fault-tolerant DSE also requires a simultaneous exploration of the design space of the architecture scenarios.

Before the exploration of the architecture scenarios can be described, a detailed definition of an architecture scenario must be given. An architecture scenario is a sequence of independent transient faults with a certain time and place. Currently, we do not consider dependent transient faults and permanent faults. For each of the architecture components that are susceptible to faults, a sequence of independent transient faults is defined. Within this fault sequence, the fault rate depends on the probability of the architecture scenario, whereas the placement of faults is completely random. Recall from the previous chapter (Section 6.4.1) that a Poisson distribution is used to model independent transient faults. This distribution can only provide the probability of a given number of events within an interval. Hence, the Poisson distribution can be used to obtain the expected waiting time on an event (as was done in the previous chapter) because waiting time corresponds to an interval in which no events occur. A Poisson distribution, however, cannot be used to model the way in which multiple faults are spread in an interval. Consequently, the probability of an architecture scenario is only dependent on the number of potential faults and not their placement:

$$\text{Prob}(S) = \max_{r \in R_P} \left(\text{Prob}_r(x \leq |\text{faults}(S, r)|)\right) \tag{7.6}$$

$$\text{Prob}_r(x \leq X) = e^{-\lambda_r} \sum_{i=0}^{X} \frac{{\lambda_r}^i}{i!} \tag{7.7}$$

To be precise, the probability of an architecture scenario is the maximum of all the probabilities of the transient faults in the individual architecture components (Equation 7.6). For each of the components $r \in R_P$, the architecture scenario contains a number of potential transient faults ($|\text{faults}(S, r)|$). Using the Poisson distribution,

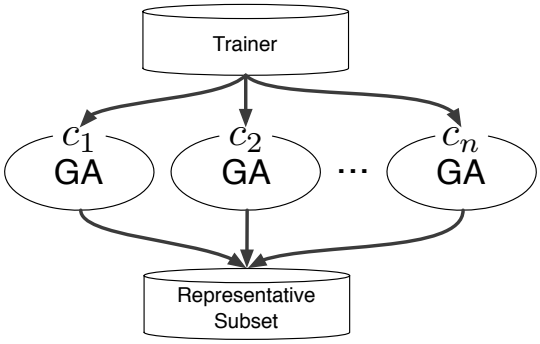


Figure 7.7: *The overview of the subset DSE that performs a separate GA to make a representative subset of architecture scenario that covers all of the reliability classes c_1 to c_n.*

the probability of at most X potential transient faults can be obtained by using the cumulative distribution function that is shown in Equation 7.7. Within this formula, λ_r is the average number of transient faults of the architecture component r during the simulation time. This average rate is obtained using the following formula:

$$\lambda_r = \frac{\text{time}}{\text{MTTF}_r} \tag{7.8}$$

For `time`, the deadline of the last frame in the multi-application workload is used. Together with the *Mean Time To Failure (MTTF)* of the architectural resource, the expected number of transient faults is obtained.

The FMapping DSE and the subset DSE are competing with each other. In the FMapping DSE the objective values are minimized, whereas the subset DSE tries to maximize the objective values. For the representative subset, this means that it contains architecture scenarios that challenge the fault-tolerant mapping as much as possible (i.e., make the worst-case objective values as large as possible), but still have a probability that is small enough to be contained in the associated reliability class.

In Figure 7.7, an overview is given of the exploration of the representative subset of architecture scenarios. For each of the reliability classes, a separate GA is used to identify the representative architecture scenarios. Generally speaking, the populations are training a fitness predictor for the worst-case fitness. This closely resembles the approach that is used in [63]. The GA will optimize the fitness predictors (i.e., architecture scenarios) such that the predicted fitness is as close to the real worst-case fitness as possible.

From each of the reliability class populations, the best architecture scenarios are selected as the representative subset of scenarios. This periodically updated representative subset of scenarios is used to predict the fitness of the fault-tolerant

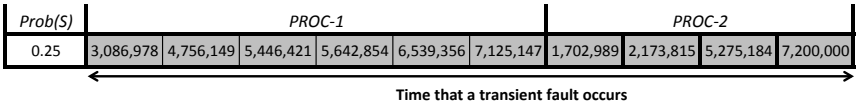


Figure 7.8: *One of the potential architecture scenario chromosome for the architecture of Figure 7.3b*

mappings in the FMapping DSE. From the FMapping DSE, on the other hand, fault-tolerant mappings are obtained that are used for the trainer. This trainer contains a set of fault-tolerant mappings (including the current Pareto front) that is used to evaluate the quality of each individual architecture scenario.

7.3.1 Chromosome Representation

To represent an architecture scenario, a chromosome is used that encodes the time for each of the transient fault that occurs. This is done separately per architecture element that is susceptible to faults (i.e., the fault rate is larger than zero).

Figure 7.8 shows an example of an architecture scenario for the architecture of Figure 7.3b. In the first gene, the probability of the complete scenario is shown. This gene is static and it depends on the reliability class in which the architecture scenario is located. The shown chromosome is located in reliability class 0.25 (i.e., its probability of occurrence is at most 25 percent). After the first gene that encodes the reliability class, the transient faults of the different components are encoded. The first group of genes is for processor PROC-1 and the second group of genes is for processor PROC-2. As we are looking for the worst-case architecture scenario, the potential number of transient faults should be as large as possible. Therefore, the number of potential faults X_r for architectural element r in reliability class c is the largest integer satisfying:

$$Prob_r(x \leq X_r) \leq c \tag{7.9}$$

In our example, this results into six potential transient faults for PROC-1 and four potential transient faults for processor PROC-2. The genes per group are chosen in such a way that it is a sample from all the integers in the range from zero to `time` (where `time` is the deadline of the last frame in the multi-application workload). Hence, multiple transient faults in the same component at the same time are not allowed. Whenever multiple things go wrong at the same time, it is still a single fault. There may, however, be a transient fault at multiple elements simultaneously.

7.3.2 Genetic Operators

The genetic operators for the architecture scenarios are rather straightforward. Upon mutation, the time of a transient fault is changed into a value that is not yet present

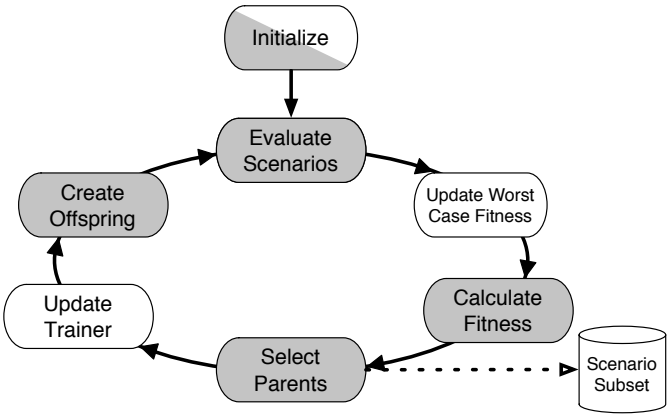


Figure 7.9: *The GA procedure of the architecture scenarios combined with the trainer update procedure. The steps that are shaded in gray need to be done for each reliability class population, whereas the other steps maintain the trainer.*

within the current group of transient faults of the specific architectural element. For the crossover, a one-point crossover is used (see Figure 2.8 at page 25): the first part of the chromosome is exchanged between the two scenario chromosomes of the same reliability class. In case the crossover results in a chromosome where a component has two transient faults at the same time, one of the transient faults is mutated.

7.3.3 Fitness Function

After manipulating an architecture scenario its fitness must be determined in order to know if the change to the architecture scenario was beneficial or not. Since the FMapping DSE tries to optimize the worst-case objectives, the predicted fitness of the architecture scenario must be as close to the worst-case objective as possible. Therefore, the fitness function F_{scen} becomes as follows:

$$F_{\text{scen}}(s) = \frac{1}{|T|} \sum_{t \in T} \overline{d}(\text{SAFE}(t, s), \text{WC}(t)) \tag{7.10}$$

To maximize the objective values of the training mappings, the fitness function tries to minimize the average normalized Euclidean distance $\overline{d}$ (see Definition 5 on page 28) between the fitness that is obtained by the SAFE simulator and the worst-case (WC) objective values for the fault-tolerant mappings t.

In order to evaluate this fitness, two issues must be addressed: 1) how to determine the trainer T and 2) how to obtain the real worst case objectives of each training mapping. To address these issues, we propose a modified GA procedure for the subset DSE that is shown in Figure 7.9. The gray steps in this picture show the

steps that are taken for the GA on all the reliability populations (as shown in Figure 7.7). These steps correspond to the steps of a generic GA: population initialization, fitness calculation of all the architecture scenarios, selection of the parents for the next generation and creating offspring from these parents. As there is one population per reliability class, the GA steps need to be done for each of the populations.

Three steps of our procedure are dedicated to the trainer. In the initialization, which is both gray and white as initialization is done both for the GA and the trainer, an initial trainer is created. At this point, the worst case fitness is not known. As it is not feasible to exhaustively evaluate training mappings with all possible architecture scenarios, the worst-case fitness will be estimated simultaneously with the search for the representative architecture scenarios. Before the fitness of an architecture scenario can be obtained, each of the training mappings must be evaluated with this architecture scenario (see equation 7.10). Possibly, the evaluation of the architecture scenario will update the worst-case fitness of one of the training mappings. Therefore, the longer a mapping is in the trainer, the more reliable its worst-case fitness is.

After the parent chromosomes of the architecture scenarios are selected for the next generation, the representative subset of scenarios is updated. At this point in time, the population only consists of scenarios with an evaluated fitness. Moreover, the NSGA-II procedure has removed the individuals with a low fitness. The update of the representative subset of scenarios also allows us to update the trainer with fault-tolerant mappings from the FMapping DSE. These mappings are already evaluated and can, therefore, be added directly. From the imported fault-tolerant mappings, the mappings are removed that are already in the trainer. After that, the new fault-tolerant mappings are added.

The final step is to control the size of the trainer. As a SAFE simulation must be done for each training mapping, the trainer should be kept as small as possible. For this purpose, non-dominated sorting is used [12]. First, the training mappings are sorted on their Pareto dominance rank. If the ranks of two mappings are the same, the crowding distance is used. The crowding distance corresponds to the average distance of neighboring mappings (with respect to fitness value). A higher average distance means a better trainer as there is a higher diversity of mappings. After sorting the training mappings, the tail of the list of training mappings is removed to truncate the trainer.

This trainer is not only beneficial to evaluate the quality of the architecture scenarios to predict the worst-case fitness. It is also used as an elite population of the best fault-tolerant mappings that are found during the fault-tolerant DSE. As this elite population is kept in the subset DSE and not in the FMapping DSE the high quality fault-tolerant mappings will be evaluated for a more diverse population of architecture scenarios than it would be the case in the FMapping DSE. Therefore, the FMapping DSE quickly selects promising fault-tolerant mappings using the representative subset of scenarios. This set of promising fault-tolerant mappings is evaluated in more detail by the subset DSE. As a result, the fault-tolerant DSE will

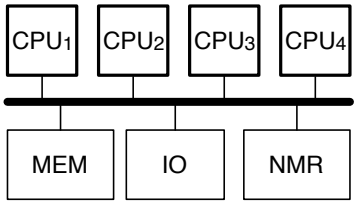


Figure 7.10: *The used architecture for the experiments in this chapter.*

end up with a Pareto front that is as precise as possible.

7.4 Case Studies

Due to the limited amount of time that was left before the thesis deadline, this chapter only shows two case studies of the fault-tolerant DSE. Future work should incorporate more extensive experiments to fully analyze the fault-tolerant DSE framework. In the first case study of this chapter, a single fault-tolerant DSE of a multi-application workload is performed. This multi-application workload consists of two applications, namely a MJPEG encoder application and a Sobel edge detector. For a static multi-application workload, the resulting Pareto fronts of the different reliability classes are obtained and analyzed. We will also investigate the effect of the reliability classes on a single fault-tolerant mapping. Basically, this shows the resilience of the different fault-tolerant mappings to the potential faults in the architecture. For the second case study, the effect of the frame period on the drop ratio of both a Sobel edge detector and an MP3 decoder is investigated.

The three different applications (a MJPEG encoder, a Sobel edge detector and an MP3 decoder) are taken from the experiments in the previous chapter (Section 6.5). A fixed workload of 63 frames is used for the Sobel edge detector and the MP3 decoder. Our used workload for the MJPEG encoder consists of smaller images sizes than the Sobel edge detector and, therefore, 267 frames are modeled for the MJPEG encoder. Each of the frames starts eight periods before the final deadline of the specific frame. In the second case study, the frame period is a parameter of the experiment. Instead, the first case study fixates the frame period: the MJPEG encoder has a frame period of $40K$ cycles (i.e., with one million cycles per second this would be 25 frames per second). On the other hand, the Sobel edge detector uses a frame period of $160K$ cycles. As a result, the deadline of the final frame of the MJPEG encoder and the Sobel edge detector is equal.

All of these applications are mapped onto the architecture that is shown in Figure 7.10. This architecture has four general-purpose processors with a mean time to failure of 10^5. As was discussed in the previous chapter, such a fault rate is relatively large and will result very pessimistic frame drop ratios. However, the reliability classes that were introduced in this chapter make it possible to show the behavior

Table 7.2: *Experimental settings*

Population size	30
Offspring size	24
$P_{crossover}$	0.8
P_{mutate}	0.02

(a) *Pattern population*

Population size	120
Offspring size	300
$P_{crossover}$	0.8
P_{mutate}	0.02

(b) *Binding population*

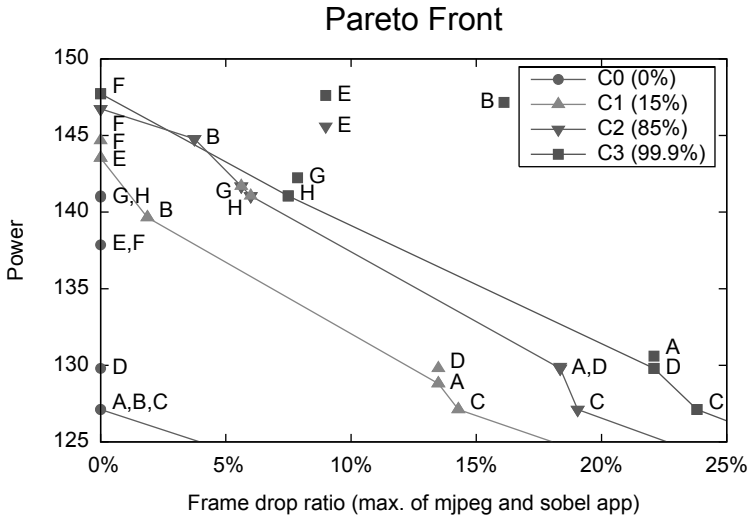
of a single fault tolerant mapping for a wider range of fault ratios at the same time. In our experiments four reliability classes are used. At first, reliability class $C_0(0\%)$ corresponds to the situation without any transient faults. Next, reliability class $C_1(15\%)$ corresponds to a situation with a relative low number of transient faults. The other two reliability classes $C_2(85\%)$ and $C_3(99.9\%)$ correspond to situations where the application is stressed with a large number of transient faults.

Additionally, our architecture also contains a NMR component to implement active redundancy. As the architecture only has a NMR component, the set of fault tolerance patterns is limited to different flavors of DMR (active redundancy with two replicas) and TMR (active redundancy with three replicas). In the previous chapter, two parameters were used for the active redundancy pattern: the restart budget and the checkpoint budget. The restart budget specified the maximal number of times that a single frame was restarted. If a voter was out of the budget, the frame was dropped. For this chapter, we adapted our model: a frame will only be restarted if there is sufficient time left. In all other cases, it is dropped. As a result, for both the DMR and TMR we have seven different versions: 1) one pattern without restart capabilities, 2) one pattern with restart capabilities, but without explicit checkpoints and 3) five patterns with restarting capabilities and an explicit checkpoint budget (10, 20, 30, 40 and 50 explicit checkpoints per million cycles).

Table 7.2 shows some of the GA parameters of the FMapping DSE component of the fault-tolerant DSE. In the subset DSE component a representative subset of architecture scenarios will be searched with at most three scenarios for each of the four reliability classes. This complete fault-tolerant DSE is run on the DAS-4 cluster [2] using five dedicated nodes with two 2.4GHz quad core Intel E5620 processors. On each node 16 MPI processes are placed to perform latency hiding (each node has eight hardware threads).

7.4.1 Pareto Front for a Fault-tolerant Multi-Application Workload

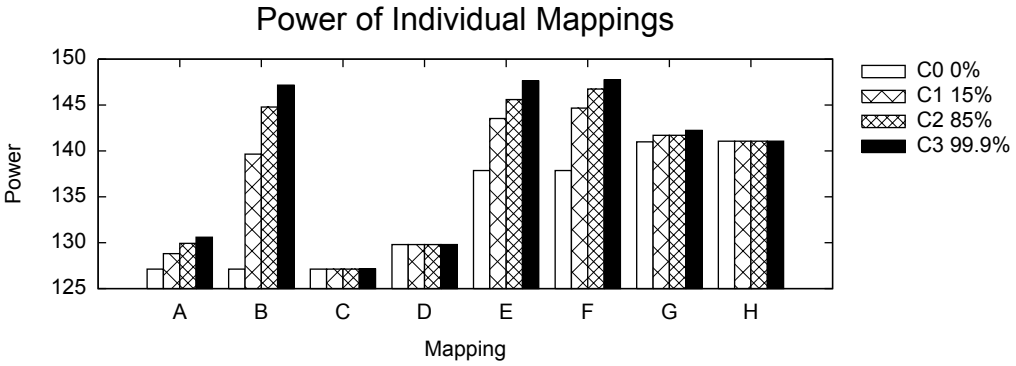
In the first case study, a fault-tolerant DSE of a multi-application workload with a MJPEG encoder and a Sobel edge detector is performed. During this fault-tolerant DSE four reliability classes were used: 0%, 15%, 85% and 99.9%. As explained



(a) *Pareto front.*

	MJPEG	Sobel
A	DMR	DMR-R
B	DMR-R	DMR-R
C	DMR	DMR
D	DMR	TMR
E	TMR-R	DMR-R
F	TMR-R	DMR-R
G	TMR	TMR-R
H	TMR	TMR

(b) *Mapping list.*



(c) *Power of individual mappings.*

Figure 7.11: *An excerpt of the Pareto front that is obtained from a DSE for a fault-tolerant implementation of a MJPEG application.*

earlier, a higher probability in a reliability class corresponds to a higher number of potential faults. This means that the reliability class of 0% has no potential faults at all, whereas the reliability class with 99.9% has the most potential faults.

Figure 7.11a shows the trade-off between the frame drop ratio (X-axis) and the average power (Y-axis) of the Pareto optimal fault-tolerant designs. As there are two applications in our multi-application workload, the objectives contain two individual frame drop ratios. Together with the average power, this would result in a three-dimensional Pareto front in which it is quite hard to visually compare the different mappings. Instead, we have chosen to show the maximum frame drop ratio of the individual MJPEG and Sobel applications. As a result, a two-dimensional Pareto

front is obtained. To further improve the clarity of the Pareto front, there is also chosen to only show the Pareto front for designs with a frame drop ratio of 25 percent or lower. Low power designs with a higher frame drop ratio are left out. Apart from the fact that designs with a high frame drop ratio are not interesting as a final design, most of these designs achieve a low power by dropping frames early in the process. Hence, a lot of energy is spared, but this is done by completely ruining the QoS of the system.

A letter between A and H indicate all of the eight different non-dominated fault-tolerant mappings in the Pareto front. In Figure 7.11b a description is given of each of these mappings. At first, all of the non-dominated fault-tolerant mappings have a single subnetwork per application. Given the small architecture and the single shared communication bus, the fault tolerance overhead (especially, the communication part) is too large to support the communication overhead of more than two fault-tolerant subnetworks. Therefore, the "best" fault-tolerant mapping with more than one subnetwork per application has a frame drop ratio of more than 96 percent. Since each application has a single fault-tolerant subnetwork, Figure 7.11b can show the patternization of each fault-tolerant mapping by showing per application which fault tolerance pattern is used in the fault-tolerant mapping. As an example, mapping A has a DMR pattern for both the MJPEG and the Sobel application. For the Sobel application, the DMR in mapping A is extended with restart capabilities (as is shown by the `-R` postfix after DMR). None of the non-dominated fault-tolerant mappings used explicit checkpointing. This was due to two reasons. At first, the computational overhead was too large; our target architecture only contains four processors that are used to support two fault-tolerant applications. Secondly, fault-tolerant mappings where the frame drop ratio was improved by using explicit checkpoints resulted in a significant increase of power. Given our multi-application workload and target architecture, a DMR with a nonzero explicit checkpoint budget had both a larger frame drop ratio and power usage than a TMR alternative where no explicit checkpoints were used.

Pareto Fronts of the Different Reliability Classes

Within the Pareto front of Figure 7.11a, all of the fault-tolerant mappings are shown for each individual reliability classes C_0 to C_3. Additionally, the Pareto front of each individual reliability class is obtained by comparing the worst-case objective values for the specific class. For reliability class C_0 only the mappings A, B and C are optimal. All of these mappings use a DMR for both the MJPEG and Sobel application. The only difference is that mapping A has restart capabilities for the Sobel application and mapping B has restart capabilities for the MJPEG applications. As reliability class C_0 has no potential faults, restart capabilities do not affect the power of a fault-tolerant design. A restart will only be done after a non-maskable fault, which will never occur at reliability class C_0. Therefore, the frame drop ratio and average power of mapping A, B and mapping C are the same for reliability class

C_0. Reliability class C_0 purely measures the overhead of the fault tolerance: do all frames meet the deadline (which is the case for mapping A to H) and how much power is used by the fault-tolerant design. In the given mappings, the only parameter that affects the power usage is the total number of replicas that is used for each application.

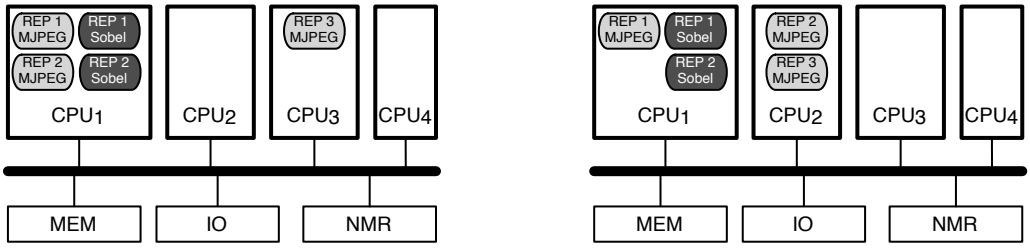
As a higher reliability class also includes the architecture scenarios of the lower reliability classes, the frame drop ratio and power of reliability classes C_1 to C_3 are always larger or equal to their preceding classes. Except for mapping E and F, all mappings have a nonzero frame drop ratio when the architecture is affected by faults. The more potential faults there are (i.e., the higher the reliability class), the higher the frame drop ratio. One of the most obvious examples is mapping B that is Pareto optimal for reliability classes C_0 up to C_2. Due to the large growth in frame drop ratio, however, the mapping is not Pareto optimal any more for reliability class C_3.

Fault Sensitivity

For some cases the fault tolerance of a mapping is sufficient to keep the frame drop ratio constant on an increasing number of potential faults. This is the case for mapping G and mapping H where both applications are using a TMR and, therefore, the fault-tolerant mapping is able to tolerate the increasing number of potential faults at the reliability classes of 15 (C_1) and 85 (C_2) percent. Similarly, fault-tolerant mapping E does not need to drop frames when the reliability class is lower or equal to 15 percent. For mapping F, the frame drop ratio even remains zero for the highest reliability class C_3 (99.9 percent).

This fault sensitivity of the power of a fault-tolerant design depends on the fault tolerance patterns that are used. This is highlighted in Figure 7.11c where for each mapping the average power of the individual reliability classes are shown. Mapping C, D and H are insensitive to the faults with respect to power. As none of the used patterns have enabled restarting, the fault-tolerant design does not use fault correction. Without any fault correction, faults only lead to an increased ratio of dropped frames, but not to an increase of power usage. The more fault correction that is applied within the system, the more sensitive the power usage of the system is to faults. For mapping B, E and F, where both applications have a fault tolerance pattern with an enabled restarting mechanism, the power usage is affected more heavily by the reliability classes than within mapping A and mapping G that only have a single application that uses fault correction.

Due to the fault sensitivity, the Pareto dominance differs over the reliability classes. This can be seen for the Pareto dominance relation of mapping A and D. The power of mapping D is fault insensitive, whereas the power of mapping A is fault sensitive (due to the restart mechanism that is used in the Sobel application). As discussed earlier, the frame drop ratio is equal to the maximum of both applications. For these mappings, the frame drop ratio is equal to the frame drop ratio of the MJPEG application. As both mappings use the same fault tolerance pattern for the



(a) *Mapping E.* **(b)** *Mapping F.*

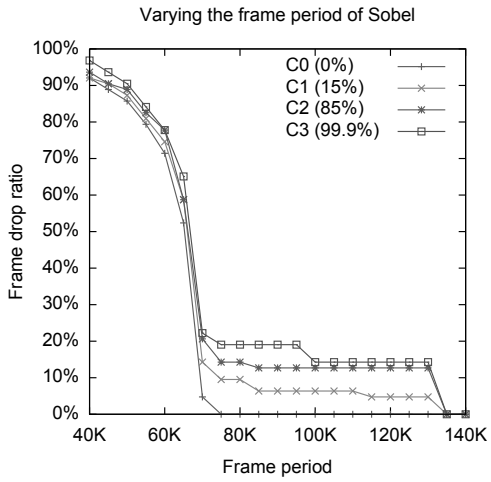
Figure 7.12: *The different computational bindings of the replicas (abbreviated by REP) in mapping E and F.*

MJPEG application, the frame drop ratio of both mappings is equal. Initially at reliability class C_0, the power of mapping A is lower than the power of mapping D. With a higher reliability class, however, the power of mapping A increases. As a result, mapping A is dominated by mapping D in reliability class C_3.

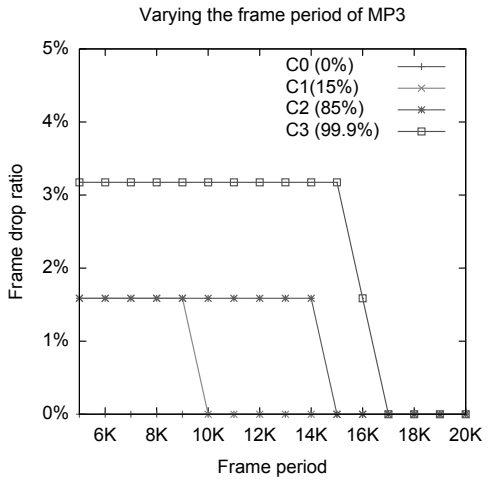
Pareto Dominance over Different Reliability Classes

For mapping G and H the Pareto dominance relation also differs per reliability class. In this case, however, it is caused by the advantages and disadvantages of the restart mechanism. Mapping G and H both use TMR for the two applications in the multi-application workload, but in mapping G the fault tolerance pattern in the Sobel application has an additional restart mechanism. As a result, mapping G takes more power for reliability classes C_1 to reliability class C_3. For reliability class C_1 and C_2, the restart mechanism still leads to a lower frame drop ratio by correcting non-maskable faults. In case of reliability class C_3, however, the computational overhead of the restart leads to more deadline misses (which results in a larger frame drop ratio).

Finally, not only the chosen pattern matters, but also the binding influences the effect of the reliability classes on the quality of the mapping. Mapping E and F, for example, have exactly the same patternization, but a different frame drop ratio. As a result, mapping E is still Pareto optimal in reliability class C_1, but for reliability class C_2 and C_3 mapping F dominates mapping E with a 9 percent lower frame drop ratio. The computational binding of the fault-tolerant mapping is the cause of this difference in frame drop ratio. As illustrated in Figure 7.12, mapping E binds two of the replicas of MJPEG to processor CPU1 and one replica to processor CPU3. In mapping F one of the replicas of the MJPEG application is bound to processor CPU1 and the other two replicas are bound to processor CPU2. For both mapping E and F, the replicas of the Sobel application are bound to processor CPU1. The processors in our architecture are homogeneous, so there is no difference between processor CPU2 and CPU3. Only the number of replicas on processor CPU1 differs.



(a) *Sobel edge detector*



(b) *MP3 decoder*

Figure 7.13: *The optimal frame drop ratio for the different reliability classes given several frame periods.*

In mapping E, there are four replicas that share the risk of being affected by a fault on processor CPU1. Mapping F, on the other hand has a more even distribution of faults. Theoretically, the worst-case frame drop ratio of both mappings should be the same (as faults are independent with respect to place and time). In practice, however, the representative subset of architecture scenarios should contain worst-case scenarios that both stress the MJPEG and Sobel application as much as possible. Since the potential faults in mapping E need to be spread over four replicas, a single architecture scenario can only stress one application at the time. For mapping F, however, a single architecture scenario can stress both applications. Consequently, the representative subset of architecture scenarios is harder to find for mapping E than it is for mapping F.

Mapping E and F emphasize the need of a representative subset of scenarios. The more representative a subset of scenarios is the better the fitness prediction of the fault-tolerant mappings become. In our current case study, the representative subset of scenarios only contained 12 architecture scenarios to facilitate a quick exploration of the design space of potential fault-tolerant mappings. As a result, poor fault-tolerant mappings are easily identified, but it becomes harder to correctly differentiate between the good mappings.

7.4.2 Frame period versus frame drop ratio

In the second case study, we shifted our focus to the frame period. Up to now, a fixed frame period was picked and used as a basis for a fault-tolerant DSE. With

the introduction of the unreliable architecture, however, it becomes even harder to pick a frame period of an application. Therefore, the second case study shows the relation between the frame drop ratio and frame period for the different reliability classes that are explored in the fault-tolerant DSE. For this purposes, we performed several fault-tolerant DSEs for different frame periods for both the MP3 and Sobel application. Each of the applications is run in isolation and the only objective that is considered is the frame drop ratio.

Figure 7.13 shows the result of the experiment. The horizontal axis shows the frame drop ratio and the vertical axis shows the lowest frame drop ratio that was encountered during the fault-tolerant DSE. This optimal frame drop ratio is shown for four different reliability classes: C_0 (0%), C_1 (15%), C_2 (85%) and C_3 (99.9%). Each DSE is shown with a single point, the lines are added for visualization purposes.

As reliability class C_0 only contains architecture scenarios with a probability of 0% or lower, the architecture scenario contains no potential faults. Therefore, it shows if there is a possible fault-tolerant mapping that is able to meet the deadlines of the application given the overhead of the fault tolerance patterns. Take for example the Sobel edge detector (Figure 7.13a) where a frame period of at least $75K$ is required to meet the deadlines of all the frames. If the frame period is lower than $75K$, the frame drop ratio quickly grows: at $65K$ the frame drop ratio is 4.7 percent and at a frame period of $60K$ it is already 52.3 percent. The MP3 application can meet its deadlines with a much lower frame period. For all the shown frame periods in Figure 7.13b, the MP3 application is able to meet its deadlines. Below a frame period of $5K$, the frame drop ratio quickly grows to 66.6 percent for a frame period of $4K$ (for visualization purposes, the graph is cut of at $5K$).

The number of corrupt frames is relatively low for the MP3 application. In case of the highest reliability class, 3.1 percent of the frames are dropped (i.e., 2 out of the 63 frames). Up to a frame period of $15K$, the best fault-tolerant mapping for the highest reliability class C_3 uses TMR without a restart mechanism. With a frame period of $16K$ and higher, the best fault-tolerant mapping also uses TMR, but then a restart mechanism is available. To be able to successfully restart a frame, the frame period should be at least $17K$. In the same fashion, the fault-tolerant DSEs are able to find a fault-tolerant mapping for reliability classes C_1 and C_2 with a zero frame drop ratio when the frame period is large enough to facilitate the restart of a frame.

Our Sobel edge detector suffers from a larger percentage of frame drops than our MP3 application. However, as Figure 7.13a shows, for all of the reliability classes the frame drops are resolved at a frame period of $135K$. Similarly to the MP3 application, at this frame period it becomes possible to extend the used TMR pattern in the fault tolerance pattern with a restart mechanism and a nonzero explicit checkpoint budget. With this increased fault tolerance, all of the faults can be corrected.

7.5 Conclusion

In this chapter, a fault-tolerant DSE framework is presented that uses architecture scenarios to explore the design space of potential fault-tolerant mappings. These fault-tolerant mappings optimize the embedded system given the potential transient faults in the architecture. One of the challenges during such a DSE is the dynamic nature of the transient faults: faults are independent and the time and place of transient faults is unpredictable. The only thing that can be given is the probability that a system is hit by a given number of transient faults within a specific interval. To describe this dynamic nature of the occurrence of transient faults within the architecture, an architecture scenario is used. Basically, an architecture scenario is a sequence of transient faults with the time and the architectural resource that is hit by the fault. For the time and placement of these faults, there are a huge number of possibilities. It is absolutely infeasible to evaluate a fault-tolerant mapping for all possible architecture scenarios and, therefore, a scenario-based framework is used. This scenario-based framework is composed of two components: an FMapping DSE and a subset DSE. In the subset DSE, a representative subset of architecture scenarios is selected to quickly predict the fitness of a fault-tolerant mapping. The FMapping DSE, on the other hand, searches for optimal fault-tolerant mappings by using two different two steps: finding optimal patternizations to transform the application into a fault-tolerant application and, given a patternization, finding optimal bindings of a fault-tolerant application onto the target architecture. To both search for the optimal patternizations and bindings, the FMapping DSE contains two separate populations: a patternization population and a binding population. For each patternization in the patternization population, a subpopulation of bindings is present in the binding population. In contrast to a single population with fault-tolerant mappings, this approach is able to easily identify successful patternizations and to exploit this knowledge during the design space exploration.

One of the main features of the fault-tolerant DSE framework is the use of reliability classes that give the reliability of the fitness of a fault-tolerant mapping. More specifically, a reliability class defines the maximal probability of the architecture scenarios that are used to determine the worst-case fitness of the fault-tolerant mapping. The higher the probability used in a reliability class, the more potential faults are injected in the unreliable target architecture when determining the fitness of a fault-tolerant mapping. Another use of reliability classes is to observe the resilience of the fault-tolerant mappings to an increasing number of transient faults. These reliability classes are integrated in the fault-tolerant DSE by simultaneously exploring the design space for all the different reliability classes. During the exploration, the fitness of a fault-tolerant mapping is the combination of the fitness values of the individual reliability classes. Hence, the representative subset contains architecture scenarios for all the different reliability classes.

At the end of this chapter case studies showed how the fault-tolerant DSE framework can be used to explore the design space of fault-tolerant embedded systems.

First, a Pareto front of fault-tolerant designs for a multi-application workload was shown. Within this Pareto front, the fault sensitivity of the mappings differs. For fault-tolerant designs that do not apply any fault correction the power is fault insensitive. The frame drop ratio of these mappings, however, quickly grows with an increasing number of potential faults. Therefore, a restart mechanism is required to achieve a low frame drop ratio when the number of potential faults increases. This restart mechanism was also crucial to achieve a zero frame drop ratio in the second case study. In this case study the relation between the frame period and the frame drop ratio was investigated for two applications. The frame drop ratio only becomes zero, when the frame period is large enough to facilitate the restart of a frame.

Part IV

Conclusion and Future Work

8 Conclusion

One of the challenges during embedded system design is the application driven design. Due to the application driven design, the objectives that are steering the design of an embedded system are mainly based on the needs of the application(s). Examples embedded system objectives are performance, power, but also battery lifetime and security. Such, potentially conflicting, objectives severely complicate embedded system design. The system level design methodology reduces the complexity of embedded system design by providing a structured approach to reduce the implementation efforts.

Part of the system level design is the early design space exploration. At a high level of abstraction, the design space of potential designs is partially explored to make early design decisions that reduce the effort that is spent in later design phases. Chapter 2 introduced the design space exploration as it used in this thesis. In our case, a design is defined by a mapping of a multi-application workload onto an MPSoC architecture. This mapping is evaluated using our high-level simulation framework Sesame (Section 2.1) that is capable to determine the quality of a single mapping in the order of seconds. There are many different ways of mapping a multi-application workload onto the architecture. As a result, there is a huge design space of potential mappings that cannot be exhaustively explored. One of the techniques to efficiently search such a large design space is a multi-objective evolutionary algorithm (MOEA). A MOEA uses a genetic algorithm (Section 2.2) that mimics the natural evolution process by keeping a population of mappings and to evolve them over a number of generations. As high quality mappings have a higher probability to reproduce, the fittest mappings will survive.

Although early design space exploration is already complex, it is becomes even more complex due to the growing dynamism in the embedded systems. In this thesis, two sources of dynamism are investigated: at the application side (Part II) and at the architecture side (Part III). At the application side, the reason for the increasing dynamism is twofold: at first the number of applications on a single embedded system increase and, secondly, the behavior of applications themselves become more dynamic. Next, at the architecture side the decreasing technology scale leads to less reliable computation. The smaller the technology, the more susceptible the ar-

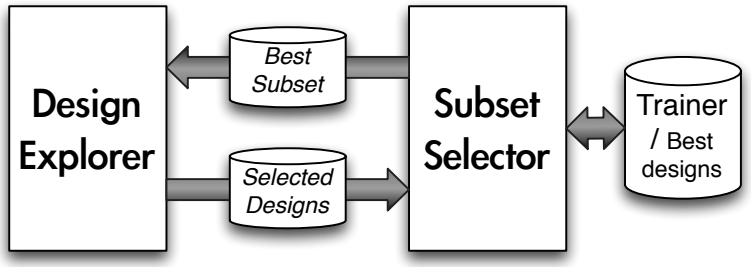


Figure 8.1: *A generalized technique to perform scenario-based DSE.*

chitecture is to faults. These faults may corrupt the output of an application that is running on the architecture. Irrespective of the dynamism inside the embedded system, it must meet the objectives of the system. For that purpose, this thesis captures the dynamism of the embedded system in scenarios and uses them to statically search for a design that is capable of meeting the objective of the embedded system in all cases.

Our problem definition (Section 1.4) concluded with the following research question: *"How can scenarios be used to enhance the design space exploration of dynamic embedded systems"*. This question was answered in Chapters 4, 5 and 7 where a scenario-based design space exploration was described. Figure 8.1 shows the general technique of the scenario-based DSE that aims at finding static mappings that perform well for all potential scenarios. This set of potential scenarios, however, is large and, therefore, it is infeasible to exhaustively evaluate all the explored mappings with the full set of potential scenarios. Our solution is to use a representative subset of scenarios that predicts the quality of a mapping by a representative subset of scenarios. For this purpose, our scenario-based DSE has two components: a design explorer and a subset selector. In the design explorer there is searched for optimal mappings. To perform the search as efficient as possible, a representative subset of scenarios is used to evaluate the mappings. For this purpose, the subset selector is responsible for selecting a representative subset of scenarios. A part of this selection problem is defining the metric that is used to determine if a subset is representative. One of the challenging issues is that the representativeness of a scenario subset is dependent on the current set of mappings in the design explorer (Chapter 4). Therefore, the trainer will contain a set of mappings that is used to train the scenario subset. While exploring the design space, the design explorer will send the selected mappings to the subset selector. These selected mappings will be used to keep the trainer up-to-date with the design explorer. On top of that, mappings in the trainer will be evaluated more thoroughly than in the design explorer. Therefore, the trainer is not only used to contain the mappings that are used to train a representative subset of scenarios, but also to contain the set of currently best mappings. Hence, at

the end of the scenario-based DSE the trainer will contain the final population of (sub-)optimal mappings.

8.1 Application Scenarios

At first, we described how application scenarios were used in scenario-based DSE to describe the dynamism in a multi-application workload. An application scenario considers dynamism of two sources: an inter-application scenario that describes which applications can be active simultaneously and an intra-application scenario that describes the dynamism within applications. Identification of application scenarios is done partly automatically and partly manually. Intra-application scenarios are detected automatically using a profiling-based method. This method is optimized to store the application scenarios as efficient as possible using a so-called scenario database. Next, inter-application scenarios need to be defined manually.

Chapter 3 showed that objectives like execution time and power differ per application scenarios. For early design space exploration, the absolute difference between the obtained non-functional requirements (i.e., mapping fitness) and the value of the non-functional requirement in the final embedded system is not relevant as long as the ordering of mapping is correct. If early design space exploration predicts that a mapping has a higher performance than another mapping, this should really be the case. The difference between the real and predicted fitness, however, does not really matter. In the experiments of Chapter 3, it turned out that the relative ordering of mappings could also vary along the different application scenarios. Therefore, it is of uttermost importance that subset of application scenarios is used that is representative for the complete set of application scenarios to compare the different mappings.

Therefore, Chapter 4 introduced a scenario-based DSE to search for a mapping that, on average, optimizes the non-functional requirements (i.e., metrics like execution time and power of the mapping when the application workload is executed on the system) given all the application scenarios in the scenario database. More specifically, a co-exploration is performed where both the design space of potential mappings and scenario subsets are explored simultaneously. This means that, in comparison to a traditional DSE a part of the computing power is not spend on the search to the optimal mapping, but the search to the best representative scenario subset.

Our experiments showed that for our multi-application workloads this investment in training a representative subset of scenarios pays off. Using stochastic applications, both multi-application workloads with a low and high dynamism (i.e., the difference between the values of the non-functional requirements when the different workloads are executed on the system) were used to compare the scenario-based DSE with a traditional DSE. By using a stochastic workload a wide range of application workloads could be tested. In the results, scenario-based DSE outperformed both a

traditional exhaustive DSE where each mapping is evaluated for all the application scenarios and a traditional DSE with a statically selected scenario subset to predict the fitness of the individual mappings. As could be expected, scenario-based DSE was much faster than an exhaustive DSE. On top of that, the quality of the resulting Pareto front of the scenario-based DSE was better or equal to the quality of the Pareto fronts of the DSE with a statically selected scenario subset. With a low diversity in the multi-application workload, the result of scenario-based DSE was comparable, but over different runs the results of scenario-based DSE where much more stable than the statically selected scenario subset. Still, with a low diversity in the multi-application workload a random scenario subset may, by coincidence, be of a relatively high quality. Hence, the outcome of the traditional DSE may also be of relatively good, but this highly depends on the subset that initially selected. The scenario database is more stable, as the subset selector continuously tries to improve the subset. With an higher diversity in the multi-application workload, however, it becomes harder to select a representative subset of scenarios and, as a result, the scenario-based DSE starts to deliver better Pareto fronts than the traditional DSE.

Nevertheless, we should comment about the selection of the representative subset of scenarios. As was illustrated by Figure 8.1, scenario-based DSE needs to divide its computational resources between the design explorer and the subset selector. The gain of a better representative subset of scenarios is that a design explorer is able to provide better fitness predictions and, thus, a potentially lower search time for the design explorer. Important, however, is that the investment in computational resources for the subset selector does not exceed the gain in computational resources at the design explorer. The opposite is the case for the size of the representative subset of scenarios. Generally, a larger representative subset of scenarios is easier to find. Hence, the computation time required at the subset selector decreases. A large representative subset of scenarios, however, leads to a larger evaluation time at the design explorer.

As the representative subset of scenarios is of significant importance to the scenario-based DSE, Chapter 5 takes a more detailed look on the subset selector. As a first step, the representativeness of a scenario subset must be defined. For this purpose, two metrics are used: misclassification ratio and the number of misclassified relations with respect to the mappings in the trainer. The misclassification ratio determines the capability of a scenario subset to correctly predict the Pareto rank. In case two subsets have the same misclassification rate, the number of misclassified Pareto dominance relations is used. These metrics, however, require that the mappings in the trainer are exhaustively evaluated. Although it is infeasible to evaluate all the mappings in the design explorer, it is feasible to exhaustively evaluate the small amount of mappings that are present in the trainer.

Furthermore, Chapter 5 also introduces three different selection techniques for representative subset of scenarios: 1) a genetic algorithm, 2) a feature selection algorithm and 3) a hybrid method combining the two aforementioned approaches.

Experiments show that the hybrid method is capable of exploiting the benefits of the two other approaches. In the genetic algorithm the design space of potential scenario subsets is quickly explored, whereas in the features selection algorithm a more systematical exploration of the local neighborhood of a scenario subset is performed. Especially for larger scenario subsets, the hybrid method can first quickly prune the design space using the genetic algorithm. This pruning delivers the first decent scenario subsets must faster than the feature selection algorithm that has a relatively slow convergence. A genetic algorithm, however, is much less effective to explore the local neighborhoods of the scenario subsets. In this case the feature selection is applied.

Finally, the sensitivity of the scenario-based DSE to the size of the scenario subset is investigated. It turns out that there is an accuracy / overhead trade-off. A larger representative subset of scenarios results in a longer evaluation time for the design explorer. Still, a large subset of scenarios is also potentially more accurate. Our experiments showed that there is an accuracy threshold with respect to the subset size. If a smaller scenario subset size is chosen, the outcome of the scenario-based DSE will be affected by the inaccurate fitness predictions. When a scenario subset is larger than the accuracy threshold, the outcome of the scenario-based DSE will not be affected. The only aspect that is affected is the convergence time of the scenario-based DSE. Therefore, the scenario-based DSE is not highly dependent on the subset size. As long as it exceeds the accuracy threshold, the outcome of the scenario-based DSE will eventually converge to an optimal set of mappings for a dynamic multi-application workload.

8.2 Architecture Scenarios

Another source of dynamism in embedded systems that is discussed in this thesis are the transient faults that may occur in unreliable MPSoC architectures. One of the sources of these transient faults is a single upset event that is caused by cosmic rays. It used to be the case that transient faults were only an issue in embedded application that were used in space, but with the decreasing technology scale it also becomes an issue at ground level. Due to these transient faults, the outcome of the computation from the processors can be corrupted. This may lead to incorrect outcomes of applications that are running on these processors. To model these unreliable architectures, architecture scenarios are used. Basically, architecture scenarios provide a sequence of transient faults that occur on the architecture with a given time and place. As transient faults are independent and infrequent, a Poisson distribution is used to model the probability of an architecture scenario.

As a first step towards a fault-tolerant DSE, Chapter 6 presents the Sesame Automated Fault-tolerant Explorer (SAFE). SAFE is an extension of Sesame that facilitates the fault-tolerant design of embedded systems. Completely in line with the separation of concerns within Sesame, SAFE has an additional pattern layer

that describes the automatic transformation of a normal application into a fault-tolerant application. To this purpose, the pattern layer consists of multiple fault tolerance patterns that define how an application is transformed into a fault-tolerant application and which policies are used for fault detection and handling. One of the examples of a type of fault tolerance pattern is active redundancy where multiple replicas (two for DMR and three for TMR) are used to run the application. All the outgoing data of the application is verified by comparing the outputs of the different replicas using a voter. In this way, corrupt data is detected as long as the majority of the replicas have the correct data. Depending on the policy, the current frame of the application must be dropped or restarted in the absence of a majority.

Based on the available fault tolerance patterns, a fault-tolerant mapping can be defined. Fault-tolerant mappings perform three different steps: 1) patternization, 2) binding and 3) dispatch. Firstly, the patternization segregates the applications into different subnetworks. Each of the subnetworks is made fault tolerant by selecting a fault tolerance pattern. By applying this fault tolerance pattern, all the applications can be transformed from a normal application into a fault-tolerant application. Secondly, the fault-tolerant application is bound onto the architecture. Per subnetwork, the binding selects the architectural resources that are required to all the replicas of the subnetworks and the additional processes to verify the computation (like the majority voter for the active redundancy). Finally, the dispatch generates the routing for the additional communication that is required to implement the fault tolerance patterns.

Using the architecture scenarios, SAFE can simulate a fault-tolerant mapping and obtain the non-functional properties of a fault-tolerant embedded system. During the simulation, the transient faults that are encoded in the architecture scenario are injected using software initiated fault injection (SWIFI). The detection and handling of these faults is completely modeled up to the restart of frames and the procedure to make explicit checkpoints. As a result, the obtained mapping fitness completely takes the fault tolerance patterns into account. On top of that, additional metrics like frame drop ratio can be introduced to make reliability a first class citizen of the DSE. This exploration of the fault-tolerant mapping is important as our experiments in Section 6.5 show that optimal type of fault tolerance pattern is completely application dependent. On top of that, the fault tolerance patterns have a non-trivial effect on system objectives like power, performance and frame drop ratio. Therefore, it is necessary to already incorporate the reliability during the early design space exploration.

Chapter 7 shows a first implementation of a fault-tolerant DSE framework. It provides an efficient early DSE of optimal fault-tolerant mappings taking into account the wide range of potential architecture scenarios. Accordingly, the scenario-based DSE framework that is shown in Figure 8.1 is perfectly applicable to the fault-tolerant DSE: the set of architecture scenarios is too large to straightforwardly evaluate all the mappings exhaustively. To discriminate between the scenario-based DSE for a

dynamic multi-application workload and the fault-tolerant DSE, the design explorer of the fault-tolerant DSE is called FMapping DSE and the subset selector is called subset DSE. As expected, the subset DSE searches for a representative subset of architecture scenarios and the FMapping DSE uses this subset to predict the fitness of the evaluated fault-tolerant mappings.

In the fault-tolerant DSE reliability classes are introduced to be able to analyze how the optimal fault-tolerant mappings behave under a different number of transient faults in the architecture. The benefit of combining the fitness of different reliability classes is shown using two case studies. Among other things, the fault sensitivity of the fault-tolerant mappings can be easily observed.

8.3 Future Work

There are many potential directions for future research based on the contributions in this thesis. At first, the contributions of this thesis purely exploit scenarios in a static fashion. Given the application or architecture scenarios in the system, the best mapping was identified that was optimized for the complete set of potential scenarios. A static mapping is used during the complete lifetime of the embedded system. Scenarios, however, can be exploited to improve the performance of the system by switching mappings during runtime based on the current scenario. In case of application scenarios, for example, starting an application may result in the activation of an additional processor or the increase of the frequency on a specific processor. Similarly, an architecture scenario can be used to optimize the performance by migrating processes from a faulty processor to another processor. The first steps for this future work are already taken in [59, 60].

To go from static mappings to dynamic mappings that can be changed during runtime, the embedded system must be able to detect scenarios. Currently, the scenarios as described in this thesis cannot easily be detected except for the inter-application scenarios that describe which applications are active simultaneously. For architecture scenarios, the closest architecture scenario can easily be defined by a distance metric and, potentially, this can also be done for intra-application scenarios. In this way, the next scenario can be predicted based on the current scenario. Next, the scenario-based DSE must be extended such that the DSE not comes up with a single optimal static mapping, but a set of mappings that is as a basis for the dynamic switch of mappings based on the scenario. This could be analogue to the reliability class of Chapter 7. Just as a fixed number of reliability classes, a fixed number of scenario clusters can be defined for which a mapping is identified. This means, however, that there must also be a technique to cluster mappings.

Apart from researching the possibilities for dynamic mappings, the work on the fault-tolerant DSE needs to be extended. Currently, the work in Chapter 7 is quite preliminary. Several steps need to be taken to deepen the research on the fault-tolerant DSE. At first, the fault-tolerant DSE should be extended to incorporate

permanent faults. This should be done alongside the dynamic mapping, as a permanent fault on an architectural resource should result in a remapping that is able to efficiently map the multi-application workload without using the defective processor. More importantly, the subset DSE of the fault-tolerant DSE should be fully analyzed. Similarly to the approach where the selection of the representative subset of application scenarios was analyzed in Chapter 5, the selection of architecture scenarios must be looked into. In the case study of Section 7.4.1 there was already emphasized that it is important that the subset of architecture scenarios is representative. However, in contrast to the subset of application scenarios, it is infeasible to exhaustively evaluate a mapping for all architecture scenarios. This complicates the subset DSE as the real fitness of a training mapping can only be approximated. Hence, other subset selection techniques may be work more efficient than the genetic algorithm that is currently used. Additionally, more extensive experiments must be done in order to study the selection of subset of architecture scenarios. Important to know, for example, is the optimal size of the representative subset. Another aspect is if there is a minimal size of the subset of architecture scenarios that is required for accurate results of the fault-tolerant DSE.

Bibliography

[1] M. Agostinelli et al. 'Random charge effects for PMOS NBTI in ultra-small gate area devices'. In: *IEEE Int. Symp. on Reliability Physics*. 2005, pp. 529–532.

[2] *ASCI DAS-4 cluster*. `http://www.cs.vu.nl/das4/home.shtml`.

[3] P. Axer, M. Sebastian and R. Ernst. 'Reliability analysis for MPSoCs with mixed-critical, hard real-time constraints'. In: *Proc. of the 9th Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Oct. 2011, pp. 149–158.

[4] L. Benini, D. Bertozzi and M. Milano. 'Resource Management Policy Handling Multiple Use-Cases in MPSoC Platforms using Constraint Programming'. In: *Logic Programming*. Vol. 5366. Lecture Notes in Computer Science. Dec. 2008, pp. 470–484.

[5] C. Bolchini et al. 'A model of soft error effects in generic IP processors'. In: *Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005. 20th IEEE Int. Symposium on*. Oct. 2005, pp. 334–342. DOI: `10.1109/DFTVS.2005.10`.

[6] C. Bolchini and A. Miele. 'An Application-Level Dependability Analysis Framework for Embedded Systems'. In: *IEEE Int. Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. Oct. 2011, pp. 171–178.

[7] G. Bournoutian and A. Orailoglu. 'Dynamic transient fault detection and recovery for embedded processor datapaths'. In: *Proceedings of the 8th IEEE / ACM / IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '12)*. Tampere, Finland, 2012, pp. 43–52.

[8] J. Branke and J. Rosenbusch. *New Approaches to Coevolutionary Worst-Case Optimization*. Vol. 5199. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008. Chap. New Approaches to Coevolutionary Worst-Case Optimization, pp. 144–153.

[9] J. M. Caroll. *Scenario-based design: envisioning work and technology in system development*. New York, NY, USA: John Wiley & Sons, Inc., 1995. ISBN: 0-471-07659-7.

[10] C. A. C. Coello, G. B. Lamont and D. A. Veldhuizen. 'Evolutionary Algorithms for Solving Multi-Objective Problems Second Edition'. In: *Evolutionary Algorithms for Solving Multi-Objective Problems*. 2nd ed. Genetic and Evolutionary Computation. Springer US, 2007. Chap. Alternative Metaheuristics. ISBN: 978-0-387-33254-3. URL: http://www.springerlink.com/content/978-0-387-33254-3.

[11] B.P. Dave and N.K. Jha. 'COFTA: hardware-software co-synthesis of heterogeneous distributed embedded systems for low overhead fault tolerance'. In: *IEEE Trans. on Computers* 48.4 (Apr. 1999), pp. 417–441.

[12] K. Deb et al. 'A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II'. In: *IEEE Transactions on Evolutionary Computation* 6.2 (Apr. 2002), pp. 182–197.

[13] O. Derin, E. Diken and L. Fiorin. 'A Middleware Approach to Achieving Fault Tolerance of Kahn Process Networks on Networks on Chips'. In: *International Journal of Reconfigurable Computing* 2011 (2011), p. 15.

[14] E. N. Elnozahy. 'Address trace compression through loop detection and reduction'. In: *SIGMETRICS Perform. Eval. Rev.* 27.1 (May 1999), pp. 214–215.

[15] C. Erbas, S. Cerav-Erbas and A. D. Pimentel. 'Multiobjective Optimization and Evolutionary Algorithms for the Application Mapping Problem in Multiprocessor System-on-Chip Design'. In: *IEEE Transactions on Evolutionary Computation* 10.3 (June 2006), pp. 358–374.

[16] *EU FP7 MultiCube Project*. http://www.multicube.eu/.

[17] E. Falkenauer. 'A new representation and operators for genetic algorithms applied to grouping problems'. In: *Evolutionary Computation* 2.2 (June 1994), pp. 123–144.

[18] F. Faure et al. 'Single-event-upset-like fault injection: a comprehensive framework'. In: *IEEE Transactions on Nuclear Science*. Vol. 52. Dec. 2005, pp. 2205–2209.

[19] M. Fleischer. 'The Measure of Pareto Optima Applications to Multi-objective Metaheuristics'. In: *Evolutionary Multi-Criterion Optimization* (2003), pp. 74–89. URL: http://dx.doi.org/10.1007/3-540-36970-8_37.

[20] J. Gaisler. 'A portable and fault-tolerant microprocessor based on the SPARC v8 architecture'. In: *International Conference on Dependable Systems and Networks (DSN 2002)*. June 2002, pp. 409–415.

[21] S. V. Gheorghita et al. 'System-scenario-based design of dynamic embedded systems'. In: *ACM Transactions on Design Automation of Electronic Systems* 14.1 (2009), pp. 1–45. ISSN: 1084-4309. DOI: http://doi.acm.org/10.1145/1455229.1455232.

[22] S. V. Gheorghita, T. Basten and H. Corporaal. 'Profiling Driven Scenario Detection and Prediction for Multimedia Applications'. In: *Embedded Computer Systems: Architectures, Modeling and Simulation, 2006. IC-SAMOS 2006. International Conference on*. 2006, pp. 63–70.

[23] S. V. Gheorghita et al. 'Automatic scenario detection for improved WCET estimation'. In: *Proceedings of the 42nd annual Design Automation Conference*. 2005, pp. 101–104.

[24] M. Glass et al. 'Reliability-Aware System Synthesis'. In: *Design, Automation Test in Europe Conference Exhibition 2007 (DATE '07)*. Apr. 2007.

[25] M. Gries. 'Methods for evaluating and covering the design space during early design development'. In: *Integration, the VLSI Journal* 38.2 (2004), pp. 131–183.

[26] S. Ha et al. 'Hardware-Software Codesign of Multimedia Embedded Systems: the PeaCE Approach'. In: *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Aug. 2006, pp. 207–214.

[27] C. Hafer et al. 'LEON 3FT Processor Radiation Effects Data'. In: *IEEE Radiation Effects Data Workshop*. July 2009, pp. 148–151.

[28] J. Huang et al. 'Reliability-Aware Design Optimization for Multiprocessor Embedded Systems'. In: *14th Euromicro Conf. on Digital System Design (DSD)*. Sept. 2011, pp. 239–246.

[29] J. Huang et al. 'A framework for reliability-aware design exploration on MPSoC based systems'. In: *Design Automation for Embedded Systems* (Apr. 2013), pp. 1–32.

[30] J. Hur et al. 'Systematic Customization of On-Chip Crossbar Interconnects'. In: *Reconfigurable Computing: Architectures, Tools and Applications*. Vol. 4419. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 61–72.

[31] *International Technology Roadmap for Semiconductors (ITRS) 2011 Edition*. www.itrs.net.

[32] V. Izosimov et al. 'Design Optimization of Time-and Cost-Constrained Fault-Tolerant Distributed Embedded Systems'. In: *Proc. of the conf. on Design, Automation and Test in Europe (DATE)*. 2005, pp. 864–869.

[33] Z.J. Jia et al. 'NASA: A generic infrastructure for system-level MP-SoC design space exploration'. In: *8th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*. Oct. 2010, pp. 41–50.

[34] Y. Jin and J. Branke. 'Evolutionary Optimization in Uncertain Environments-a Survey'. In: *IEEE Transactions on evolutionary computation* 9.3 (2005), pp. 303–317.

[35] E. E. Johnson, J. Ha and M. B. Zaidi. 'Lossless Trace Compression'. In: *IEEE Transactions on Computers* 50.2 (2001), pp. 158–173.

[36] G. Kahn. 'The Semantics of Simple Language for Parallel Programming'. In: *IFIP Congress*. 1974, pp. 471–475.

[37] V. Khare, X. Yao and K. Deb. 'Performance Scaling of Multi-objective Evolutionary Algorithms'. In: *Evolutionary Multi-Criterion Optimization*. Vol. 2632. Lecture Notes in Computer Science. Springer Berlin Heidelber, 2003, pp. 376–390.

[38] B. Kienhuis et al. 'A methodology to design programmble embedded systems: the Y-chart approach'. In: *Lecture Notes in Computer Science - Embedded Processor Design Challenges* 2268 (2002), pp. 18–37.

[39] M. Kim et al. 'Energy-Aware Cosynthesis of Real-Time Multimedia Applications on MPSoCs using Heterogeneous Scheduling Policies'. In: *ACM Transactions on Embedded Computing Systems* 7.2 (Feb. 2008), pp. 1–19.

[40] A. Kumar et al. 'Multiprocessor Systems Synthesis for Multiple Use-Cases of Multiple Applications on FPGA'. In: *ACM Transactions on Design Automation of Electronic Systems* 13.3 (July 2008), pp. 1–27.

[41] K. Lee et al. 'Mitigating the impact of hardware defects on multimedia applications: a cross-layer approach'. In: *Proc. of the 16th ACM int. conf. on Multimedia*. 2008, pp. 319–328.

[42] *LISA cluster*. `http://www.sara.nl/systems/lisa`. 2011.

[43] M.L. Maher and J Poon. 'Modeling Design Exploration as Co-Evolution'. In: *Microcomputers in Civil Engineering* (1996).

[44] A. Milenković and M. Milenković. 'An efficient single-pass trace compression technique utilizing instruction streams'. In: *ACM Transactions on Modeling and Computer Simulation* 17.1 (Jan. 2007).

[45] M. Mitchell. *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262631857.

[46] S. Mitra et al. 'Robust system design with built-in soft-error resilience'. In: *IEEE Computer* 38.2 (Feb. 2005), pp. 43–52.

[47] S. Mitra, N.R. Saxena and E.J. McCluskey. 'Common-mode failures in redundant VLSI systems: a survey'. In: *IEEE Transactions on Reliability* 49.3 (2000), pp. 285–295.

[48] S. Murali et al. 'A Methodology for Mapping Multiple Use-Cases onto Networks on Chips'. In: *DATE '06: Proceedings of the conference on Design, Automation and Test in Europe*. Mar. 2006, pp. 118–123.

[49] F. Oboril et al. 'Reducing NBTI-induced processor wearout by exploiting the timing slack of instructions'. In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '12)*. Tampere, Finland, 2012, pp. 443–452.

[50] H. Oh and S. Ha. 'Hardware-Software Cosynthesis of Multi-Mode Multi-Task Embedded Systems with Real-Time Constraints'. In: *Proceedings of the 10th international symposium on Hardware/Software Codesign (CODES)*. May 2002, pp. 133–138.

[51] H. Orsila. *kpn-generator*. `http://zakalwe.fi/~shd/foss/kpn-generator/`. Feb. 2009.

[52] G. Palermo, C. Silvano and V. Zaccaria. 'Robust Optimization of SoC Architectures: A Multi-Scenario Approach'. In: *Proceedings of ESTIMedia 2008 - IEEE Workshop on Embedded Systems for Real-Time Multimedia. Atlanta, Georgia, USA*. October 2008.

[53] J. Paredis. 'Coevolutionary Computation'. In: *Artificial Life* 2.4 (1995), pp. 355–375.

[54] D. Park et al. 'Exploring Fault-Tolerant Network-on-Chip Architectures'. In: *International Conference on Dependable Systems and Networks (DSN 2006)*. June 2006, pp. 93–104.

[55] S. Pasricha, N. Dutt and F. J. Kurdahi. 'Dynamically Reconfigurable On-Chip Communication for Multiple Use-Case Chip Multiprocessor Applications'. In: *ASP-DAC '09: Proceedings of the 2009 Conference on Asia and South Pacific Design Automation*. Jan. 2009, pp. 25–30.

[56] J. M. Paul, D. E. Thomas and A. Bobrek. 'Scenario-oriented design for single-chip heterogeneous multiprocessors'. In: *IEEE Trans. Very Large Scale Integr. Syst.* 14.8 (Aug. 2006), pp. 868–880. DOI: `http://dx.doi.org/10.1109/TVLSI.2006.878474`.

[57] A. D. Pimentel, C. Erbas and S. Polstra. 'A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels'. In: *IEEE Transactions on Computers* 55.2 (2006), pp. 99–112. ISSN: 0018-9340. DOI: `http://doi.ieeecomputersociety.org/10.1109/TC.2006.16`.

[58] D.K. Pradhan and N.H. Vaidya. 'Roll-forward checkpointing scheme: a novel fault-tolerant architecture'. In: *IEEE Trans. on Computers* 43.10 (Oct. 1994), pp. 1163–1174.

[59] W. Quan and A. D. Pimentel. 'A Scenario-based Run-time Task Mapping Algorithm for MPSoCs'. In: *Proc. of the 50th ACM/IEEE Int. Design Automation Conference (DAC '13)*. Austin, USA, June 2013.

[60] W. Quan and A. D. Pimentel. 'An Iterative Multi-Application Mapping Algorithm for Heterogeneous MPSoCs'. In: *to appear in 11th IEEE Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*. Oct. 2013.

[61] A. Rohani and H.G. Kerkhoff. 'A Technique for Accelerating Injection of Transient Faults in Complex SoCs'. In: *14th Euromicro Conf. on Digital System Design (DSD)*. Sept. 2011, pp. 213–220.

[62] D. J. Schaffer et al. 'A study of control parameters affecting online performance of genetic algorithms for function optimization'. In: *Proceedings of the third international conference on Genetic algorithms*. George Mason University, United States: Morgan Kaufmann Publishers Inc., 1989, pp. 51–60. ISBN: 1-55860-006-3.

[63] M. D. Schmidt and H. Lipson. 'Coevolution of Fitness Predictors'. In: *IEEE Transactions on Evolutionary Computation* 12.6 (2008), pp. 736–749.

[64] L. Schor et al. 'Scenario-based design flow for mapping streaming applications onto on-chip many-core systems'. In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. Tampere, Finland, Oct. 2012, pp. 71–80.

[65] N. Seifert et al. 'Soft Error Susceptibilities of 22 nm Tri-Gate Devices'. In: *IEEE Transactions on Nuclear Science* 59.6 (2012), pp. 2666–2673.

[66] P. Shivakumar et al. 'Modeling the effect of technology trends on the soft error rate of combinational logic'. In: *Proc. of the Int. Conference on Dependable Systems and Networks (DSN'02)*. June 2002, pp. 389–398. DOI: 10.1109/DSN.2002.1028924.

[67] A. K. Singh et al. 'Mapping on Multi/Many-Core Systems: Survey of Current and Emerging Trends'. In: *Proc. of the 50th ACM/IEEE Int. Design Automation Conference (DAC '13)*. Austin, USA, June 2013.

[68] P. Somol et al. 'Dynamic Oscillating Search algorithm for feature selection'. In: *International Conference on Pattern Recognition 2008 (ICPR 2008)*. Dec. 2008, pp. 1–4. DOI: http://doi.ieeecomputersociety.org/10.1109/ICPR.2008.4761773.

[69] C. Spearman. 'The Proof and Measurement of Association between Two Things'. In: *The American Journal of Psychology* 15.1 (Jan. 1904), pp. 72–101.

[70] J. Stoye and D. Gusfield. 'Simple and Flexible Detection of Contiguous Repeats Using a Suffix Tree'. In: *Theoretical Computer Science* 270.1-2 (Jan. 2002), pp. 843–850.

[71] S. Stuijk, M. Geilen and T. Basten. 'A Predictable Multiprocessor Design Flow for Streaming Applications with Dynamic Behaviour'. In: *13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*. Lille, France, Sept. 2010, pp. 548–555.

[72] S. Stuijk et al. 'Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications'. In: *International Conference on Embedded Computer Systems (SAMOS)*. June 2011, pp. 404–411.

[73] B. Theelen et al. 'A Scenario-Aware Data Flow Model for Combined Long-Run Average and Worst-Case Performance Analysis'. In: *Proc. of the Int. Conference on Formal Methods and Models for Codesign*. 2006, pp. 185–194.

[74] M. Thompson. 'Tools and techniques for efficient system-level design space exploration'. PhD thesis. Universiteit van Amsterdam, Jan. 2012.

[75] M. Thompson et al. 'A Framework for Rapid System-Level Exploration, Synthesis and Programming of Multimedia MP-SOCs'. In: *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM International Conference on Hardware / Software Codesign and System Synthesis*. Salzburg, Austria, Sept. 2007, pp. 9–14.

[76] D. A. van Veldhuizen. 'Multiobjective Evolutionary Algorithms: Classifications, Analyses, and New Innovations'. PhD thesis. Air Force Institute of Technology, 1999.

[77] D. A. van Veldhuizen and G. B. Lamont. 'Multiobjective Evolutionary Algorithms: Analyzing the State-of-the-Art'. In: *Evolutionary Computation* 8.2 (2000), pp. 125–147.

[78] W. Werum and H. Windauer. *Introduction to PEARL: process and experiment automation realtime language*. 2nd ed. Philadelphia, PA, USA: Heyden & Sons, Inc., 1983.

[79] W. Wolf, A.A. Jerraya and G. Martin. 'Multiprocessor System-on-Chip (MPSoC) Technology'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.10 (Oct. 2008), pp. 1701–1713.

[80] E. Zitzler, D. Brockhoff and L. Thiele. 'The Hypervolume Indicator Revisited: On the Design of Pareto-compliant Indicators Via Weighted Integration'. In: *Evolutionary Multi-Criterion Optimization*. Vol. 4403. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 862–876.

[81] E. Zitzler, M. Laumanns and L. Thiele. 'SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization'. In: *Evolutionary Methods for Design, Optimisation, and Control*. CIMNE, Barcelona, Spain, 2002, pp. 95–100.

[82] E. Zitzler et al. 'Performance assessment of multiobjective optimizers: An analysis and review'. In: *IEEE Transactions on Evolutionary Computation* 7.2 (2003), pp. 117–132.

List of Author's Publications

[83] P. van Stralen. 'Scenariogebaseerde exploratie van de MPSoC-ontwerpruimte'. In: *Tijdschrift Informatie* (Feb. 2011).

[84] P. van Stralen. *Using Chip Multithreading to Speed Up Scenario-Based Design Space Exploration*. Tech. rep. arXiv:1308.6469 [cs.PF]. Universiteit van Amsterdam, Aug. 2013. URL: `http://arxiv.org/abs/1308.6469`.

[85] P. van Stralen and A. D. Pimentel. 'Signature-based Microprocessor Power Modeling for Rapid System-level Design Space Exploration'. In: *IEEE / ACM / IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*. Oct. 2007, pp. 33–38.

[86] P. van Stralen and A. D. Pimentel. 'A High-level Microprocessor Power Modeling Technique Based on Event Signatures'. In: *Journal of Signal Processing Systems* 60.2 (Aug. 2010), pp. 239–250.

[87] P. van Stralen and A. D. Pimentel. 'A Trace-based Scenario Database for High-level Simulation of Multimedia MP-SoCs'. In: *Proc. of the Int. Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS '10)*. Samos, Greece, July 2010.

[88] P. van Stralen and A. D. Pimentel. 'Scenario-Based Design Space Exploration of MPSoCs'. In: *Proceedings of IEEE International Conference on Computer Design (ICCD '10)*. Oct. 2010.

[89] P. van Stralen and A. D. Pimentel. 'A SAFE approach towards early design space exploration of fault-tolerant multimedia MPSoCs'. In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '12)*. Tampere, Finland, Oct. 2012, pp. 393–402.

[90] P. van Stralen and A. D. Pimentel. 'Fast Scenario-Based Design Space Exploration using Feature Selection'. In: *Proc. of the Int. Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA'12)*. Feb. 2012.

[91] P. van Stralen and A. D. Pimentel. 'Fitness Prediction Techniques for Scenario-based Design Space Exploration'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 32.8 (Aug. 2013), pp. 1240–1253.

[92] P. van Stralen and A.D. Pimentel. 'Using Chip Multithreading to Speed Up Scenario-Based Design Space Exploration: A Case Study'. In: *To be presented at the workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*. Jan. 2014.

Samenvatting

In dit proefschrift ligt de nadruk op de vroegtijdige ontwerpexploratie voor embedded systemen. De vraag naar deze systemen is sterk gegroeid. Mede door de veelzijdigheid en de multifunctionaliteit van de applicaties die in embedded systemen wordt gebruikt is het ontwerpen van een embedded systeem een complexe procedure (Hoofdstuk 1). Dit wordt nog eens versterkt door de vele niet-functionele eisen (zoals uitvoeringstijd en energieverbruik) die aan het systeem worden gesteld, die ook nog tegenstrijdig kunnen zijn. Gedurende het ontwerpproces wordt er een afbeelding van de applicatie naar de gebruikte architectuur gemaakt. Met behulp van een abstracte beschrijving van de applicatie(s) en de architectuur worden de niet-functionele eigenschappen van het beschreven systeem bepaald. Hierdoor kunnen er al vroeg in het ontwerpproces ontwerpkeuzes gemaakt worden en kan er zodoende veel tijd worden bespaard gedurende de rest van het ontwerptraject.

Centraal in dit proefschrift staat de vraag "Hoe kunnen scenario's worden toegepast om de dynamiek te modelleren van embedded systemen". Twee scenario types worden beschreven: applicatie scenario's en architectuur scenario's. Applicatie scenario's beschrijven de dynamiek van de applicaties die draaien op het embedded systeem. Dit kan zowel het gedrag binnen een individuele applicatie, als het gedrag tussen meerdere applicaties beschrijven. Architectuur scenario's daarentegen, beschrijven het dynamische gedrag van de architectuur zelf met betrekking tot tijdelijke fouten in het systeem. Als gevolg van deze fouten kan de uitkomst van een berekening incorrect zijn. Dit heeft uiteraard direct gevolg op de output van de applicatie(s) binnen het embedded systeem.

In hoofdstuk 2 wordt er beschreven hoe een genetisch zoekalgoritme gebruikt kan worden om een set van optimale afbeeldingen te vinden van een applicatie op een architectuur op basis van een of meerdere niet-functionele eisen. Dit hoofdstuk introduceert ook de *Sesame* simulatie omgeving. Vervolgens wordt in hoofdstuk 3 beschreven hoe applicatie scenario's gedetecteerd kunnen worden met behulp van Sesame. Na de gedeeltelijk automatische detectie van de scenario's wordt de efficiënte opslag in een scenario database beschreven.

Met behulp van de scenario database wordt er in hoofdstuk 4 en 5 een scenario gebaseerde ontwerpexploratie gedefiniëerd voor embedded systemen met een of meerdere (dynamische) applicaties. Kern van deze exploratie is het onderscheid van twee componenten: de ontwerpexploratie en de scenario selectie. Het doel van de

ontwerpexploratie is de zoektocht naar optimale afbeeldingen van de applicaties op de architectuur. Deze afbeeldingen moeten goed functioneren voor alle mogelijke applicatie scenario's. Echter, het aantal potentiele applicatie scenario's is te groot om tijdens de exploratie alle onderzochte afbeeldingen grondig te evalueren voor alle potentiele applicatie scenario's. Vandaar dat een kleine groep scenario's wordt geselecteerd die representatief is voor alle mogelijke applicatie scenario's. Waar in hoofdstuk 4 de nadruk wordt gelegd op de gehele scenario gebaseerde omgeving, wordt in hoofdstuk 5 met name gericht op de verschillende technieken om een representatieve groep van scenario's te selecteren.

De scenario gebaseerde exploratie omgeving kan ook worden gebruikt om met behulp van architectuur scenario's een embedded systeem te ontwerpen dat tolerant is met betrekking tot tijdelijke storingen in de architectuur. Dit wordt beschreven in hoofdstuk 6 en 7. In hoofdstuk 6 wordt beschreven hoe Sesame wordt uitgebreid om tijdelijke storingen te modeleren. Ten eerste wordt er een fouttolerante afbeelding geïntroduceerd die een normale applicatie automatisch omzet in een fouttolerante applicatie, waarna de gehele fouttolerante applicatie wordt afgebeeld op een architectuur. Ten tweede wordt de Sesame simulatie omgeving uitgebreid om de aanwezigheid van fouten, foutdetectie en foutcorrectie compleet te modelleren. In hoofdstuk 7 wordt de exploratie van fouttolerante afbeeldingen met behulp van een scenario gebaseerde exploratie methode beschreven, die tegelijkertijd zoekt naar een optimale set van fouttolerante afbeeldingen en een set van representatieve architectuur scenario's om de afbeeldingen te evalueren.

In de conclusie wordt teruggekomen op de centrale vraag van dit proefschrift: Hoe kunnen scenario's worden toegepast om dynamiek te modelleren in embedded systemen. Als oplossing wordt gegeven om een scenario gebaseerde ontwerpexploratie te gebruiken die resulteert in een set van afbeeldingen worden gevonden die zich goed gedragen in alle mogelijke scenario's van het dynamische embedded systeem.

Dankwoord

De basis van dit proefschrift was al lang gelegd voordat ik in 2009 met mijn promotieonderzoek begon. Om precies te zijn, begon het allemaal op mijn verjaardag in 2004. Op de open dag voor de bachelor Informatica van de Universiteit van Amsterdam werd niet alleen het studentenleven opgehemeld, maar werd er ook een proefcollege gegeven. Ik weet niet precies meer wie dat college gaf (volgens mij was het Yde Venema), maar de enthousiasme waar het college mee gegeven werd zorgde ervoor dat ik ter plekke besloot om Informatica te gaan studeren aan de Universiteit van Amsterdam. Dit enthousiasme bleef duidelijk aanwezig tijdens de zeven jaar dat ik aan de Universiteit van Amsterdam heb gestudeerd en ik ben het docententeam daarvoor ook erg dankbaar. Ongetwijfeld vergeet ik wat van de docenten, maar Alfons Hoekstra, Dick van Albada, Inge Bethke, Rein van den Boomgaard, Ben Bruidegom, Esdert Edens, Toto van Inge, Tom Koornwinder, José Lagerberg, Leen Torenvliet, Yde Venema, Andy Pimentel, Chris Jesshope, Alban Ponse en Jaap Kaandorp bedankt voor jullie inzet!

Gedurende mijn Bachelor kwam ik ook in contact met Andy. De eerste dat ik hem tegenkwam was tijdens colleges van het vak Architectuur en Computerorganisatie. Het bleek het begin van een lange samenwerking. Zowel het eindproject van mijn bachelor en master heb ik gedaan onder begeleiding van Andy. Dit heb ik niet alleen met veel plezier gedaan, maar ook was daarbij het resultaat van het onderzoek boven mijn verwachting. Ik hoefde dan ook niet lang na te denken toen hij mij een promotieplek aanbood binnen het EASY project. Hier kon ik het onderzoek wat ik tijdens mijn master had gedaan voortzetten en verder uitdiepen. Samenwerken met Andy was ideaal, je kon altijd bij hem binnenlopen om dingen te bespreken. Daarbij gaf hij heel veel vrijheid binnen je eigen onderzoek. Ik heb het ook gewaardeerd dat Andy me heeft meegenomen naar Salzburg om mijn bachelor project te presenteren op een workshop. Omdat ik het niet helemaal aandurfde om binnen een internationale groep mensen mijn werk te presenteren heeft hij daar mijn werk gepresenteerd. Na meerdere presentaties daar gezien te hebben, zag ik in dat presenteren eigenlijk niet zo lastig is. Dit is iets waar ik vandaag de dag nog van profiteer!

Dat ik met mijn promotie begon, zaten we nog in het Nikhef gebouw. Hier kwam ik op de kamer bij Mark Thompson, Toktam Taghavi en Roberta Piscitelli. Het was een erg fijne kamer en het mooie was dat we ondanks de afwezigheid van klimaatbeheersing altijd nog wat frisse lucht in de kamer konden krijgen. Niet alleen

hadden we de mobiele koeling van Toktam, maar het raam kon ook open. En dat bedoel ik dan ook letterlijk: je tilde hem er helemaal uit. Ook door de andere collega's werd ik warm ontvangen: o.a. Simon Polstra, Mike Lankamp, Raphael Poss, Michiel van Tol, Irfan Uddin, en Michael Hicks. Mike zorgde op een hele andere manier voor verkoeling: ik voel nog steeds de sneeuwbal in mijn nek die eigenlijk voor Michael bedoeld was (en naar de woorden van Mike; Hij moest toch wat met de sneeuwbal)!

Simon is altijd een goede steun geweest bij het gebruik van Sesame. Zijn installatietekstje voor Sesame die hij me stuurde tijdens mijn Bachelor heb ik jaren daarna nog gebruikt (Het is de oudste email die nog in mijn inbox staat)! Daarnaast was het ook fijn om iemand te hebben die, net als ik, ook een sporthart had. Ik heb genoten van de vele fiets en MTB verhalen.

In 2010 verhuisden we naar het nieuwe gebouw aan Science Park 904. Architectonisch gezien heel mooi, praktisch gezien was het wat minder. Hier zat ik samen met Roberta Piscitelli bij het raam (die net als mij wat daglicht kon waarderen) in een kamer die we uiteindelijk deelde met Simon, Wei, en Bert. In de loop van de jaren zijn er ook verscheidene collega's bijgekomen in ons onderzoeksgroep zoals Quan Wei, Roeland Douma, Roy Bakker, Sebastian Altmeyer, en Bert Gijsbers. Samen met Clemens Grelck en Chris Jesshope resulteerde dit in een kleinschalige onderzoeksgroep waar in een ongedwongen sfeer elkaars onderzoek versterkt werd. Ook moet ik het secretariaat niet vergeten: Erik Hepiteuw en Brechtje Schipper bedankt voor alle ondersteuning.

Daarnaast wil ik de leden van de promotiecommissie (mijn promotor Chris Jesshope, Cees de Laat, Jurgen Teich, Gerard Smit, Henk Corporaal) bedanken voor het beoordelen van mijn proefschrift. Daarbij moet ik ook de anonieme reviewers niet vergeten die in de loop van de jaren mijn artikelen hebben beoordeeld die ik naar verscheidene conferenties heb gestuurd. Met de constructieve feedback die ik heb mogen ontvangen, heb ik vele verbeteringen kunnen doorvoeren in mijn onderzoek. Hierbij dank ik ook Simon, Sabien, Carla, Dick, en Wendy die in de laatste dagen nog hebben geholpen met de omslag van het proefschrift en de Nederlandse samenvatting.

Tenslotte wil ik mijn familie nog bedanken: Allereerst mijn ouders voor het feit dat ik tijdens mijn promotie nog thuis heb mogen wonen. Niet alleen heeft me dat veel tijd gescheeld aan huishoudelijk werk, maar elke dag dat ik thuis kwam van het werk was er altijd iemand om de dag weer door te spreken. Mijn zussen (Renate, Wendy, en Carla), mijn zwagers (Rick, Rob, en Dick) en uiteraard mijn neefjes en nichtjes (Demi, Jayden, Joran, Nathan, Kensi, en Pepijn) elke zondag dat jullie langskwamen was het altijd weer gezellig in huis. Hierdoor kon me elke week weer opladen voor de volgende werkweek. Ook mijn medeatleten en trainer van AV Hera hebben er voor gezorgd dat ik de gedachten van alledag weg kon lopen tijdens de trainingen op dinsdag en donderdag.

Het is onvermijdelijk dat ik nog mensen ben vergeten die ik zou moeten bedanken. Mocht je naam er niet bij staan: bedankt, ook al is het alleen maar voor het lezen van mijn proefschrift!