Applications of scenarios in early embedded system design space exploration
van Stralen, P.

Citation for published version (APA):
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.
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 1.1: An example of the mapping of an application onto a heterogeneous MPSoC.
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
CHAPTER 1. INTRODUCTION

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.