Kamana Sigdel

System-Level Design Space Exploration of Reconfigurable Architectures

System-Level Design Space Exploration of Reconfigurable Architectures

---

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op donderdag 17 februari 2011 om 12:30 uur

door

Kamana SIGDEL

ingenieur in computer engineering
geboren te Nepal

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. H.J. Sips

Copromotor:
Dr. K.L.M. Bertels

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof.dr.ir. H.J. Sips	Technische Universiteit Delft, promotor
Dr. K.L.M. Bertels	Technische Universiteit Delft, copromotor
Prof.dr.ir. K. De Bosschere	Universiteit Gent
Prof. dr. A. van Deursen	Technische Universiteit Delft
Prof. dr. rer. nat. R. Leupers	Rheinisch-Westfalische Technische Hochschule Aachen University
Dr. A.D. Pimentel	Universiteit van Amsterdam
Dr. C. Silvano	Politecnico di Milano
Prof.dr. F.M.T Brazier	Technische Universiteit Delft, reservelid

Met samenvatting in het Nederlands.

Dr. Pimentel heeft als copromotor in belangrijke mate aan de totstandkoming van het proefschrift bijgedragen.

ISBN 978-90-72298-12-6

Graphical cover illustration by Jochem Gugelot Beeld & Vormgeving (www.jochemgugelot.nl)

Printed in The Netherlands

To the immense courage that sometimes provides us the strength to get back and fight against all odds.

System-Level Design Space Exploration of Reconfigurable Architectures

Abstract

Reconfigurable architectures are becoming increasingly popular as they bear a promise of combining the flexibility of software with the performance of hardware. Nevertheless, such architectures are subject to numerous constraints, such as performance, memory requirements, chip area, and power consumption. To create an efficient design, performing Design Space Exploration (DSE) at various stages is essential in order to effectively appraise several design alternatives. DSE at early design stages facilitates designers in rapid performance evaluation of different parameters, such as architectural characteristics, application-to-architecture mappings, scheduling policies, and hardware-software partitionings. DSE methodologies help traversing (typically) huge design spaces efficiently, thus performing DSE at a high level of abstraction facilitates design decisions to be made at very early design stages, which can significantly reduce the overall design time of a system.

Towards this goal, in this dissertation, we develop a generic system-level framework, called *rSesame*, in order to perform modeling and simulation of dynamically reconfigurable architectures at early design stages. The framework can be deployed as a standard modeling and simulation framework for performing system-level DSE to explore several design parameters, while designing dynamically reconfigurable architectures. Performing runtime evaluations together with static explorations, enables reconfigurable architectures to be more efficient in terms of several design constraints. As a result, the rSesame framework combines both static and runtime explorations in order to facilitate system-level DSE of reconfigurable architectures with respect to architectural exploration, hardware-software partitioning and task mapping/scheduling.

We deployed the rSesame framework to evaluate the Molen reconfigurable architecture by assessing and evaluating a wide range of application-to-architecture-mappings. These mappings are evaluated based on different

system attributes, such as execution time, number of reconfigurations, time-weighted area usage, percentage of hardware/software execution, percentage of reconfiguration, and hardware reusability efficiency, under different resource conditions. The case study shows that the rSesame framework can be efficiently deployed to facilitate system-level DSE of reconfigurable architectures by effectively appraising several alternatives, both statically and at runtime. The study also shows that the framework can be deployed, not only to evaluate and compare different architecture-to-application-mappings, but also to efficiently evaluate different architectural conditions at runtime.

Table of contents

List of Tables

List of Figures

List of Algorithms

List of Acronyms and Symbols

ADSP	Advanced Digital Signal Processor
AMAP	As Much As Possible heuristic
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-set Processor
CBH	Cumulative Based Heuristic
CCU	Custom Configured Unit
CPLD	Complex Programmable Logic Device
DSE	Design Space Exploration
DSP	Digital Signal Processor
FIFO	First In First Out
FPGA	Field Programmable Gate Arrays
FSM	Finite State Machine
GPP	General Purpose Processor
HW	Hardware
IBH	Interval Bases Heuristic
ILP	Integer Linear Programming
ICAP	Internal Configuration Access Port
IP	Intellectual Property
KPN	Kahn Processor Network
MJPEG	Motion-JPEG
MPSoC	Multiprocessor System-on-Chip
NoC	Network on Chip
OS	Operating System
QoS	Quality of Service
PLB	Processor Local Bus
$\rho\mu$-code	reconfigurable microcode
rDPA	reconfigurable Datapath Array
R	Read event
RBH	Reusability Based Heuristic
RE	Reusability Efficiency
RE_{Task}	Reusability Efficiency of a Task
RE_{App}	Reusability Efficiency of an Application
RER	Reconfiguration-to-Execution Ratio
RM	Resource Manager
RMM	Runtime Mapping Manager
RP	Reconfigurable Processor
RTL	Register Transfer Level

RU	Reconfigurable Unit
SoC	System-on-Chip
SW	Software
TML	Transaction Level Modeling
T_{HW}	Hardware execution Time
T_{Recon}	Reconfiguration Time
T_{SW}	Software execution Time
UML	Unified Modeling Language
VHDL	Very High Scale Integrated Circuits Hardware Description Language
VP	Virtual Processor
W	Write event
X	eXecute event
X_{pragma}	eXecute pragma event

1 Introduction

In today's world, embedded systems are ubiquitous. Ranging from the simple, such as smart phones and printers, to the very sophisticated, such as wafer steppers and medical imaging equipments, all devices are equipped with a certain kind of embedded system. The design goal of such systems comes with rather contradicting requirements. For instance, on one hand, since these systems targets mass producing consumer goods, they should be cost efficient, small size and have low power. On the other hand, they need to execute a wide range of functionalities, and therefore must have high performance and flexibility. To support such a wide spectrum of functional demands, the construction of heterogeneous system, which consists of different types of processing cores, customizable hardware components, and memories, is often necessary.

The increasing complexity of contemporary embedded systems with their heterogeneous architecture, and at the same time their conflicting design requirements, complicates the design of such systems. There is a wide range of design parameters that have to be tweaked and tuned to find the "best" tradeoff in terms of several design requirements. Thus, the system optimization in these systems becomes a challenging task. In order to create an effective design, performing Design Space Exploration (DSE) to efficiently appraise design requirements, at different stages, is crucial [1–3]. The DSE can be defined as a process of investigating several "functionally equivalent" implementation alternatives, in order to identify an "optimal" solution [1]. The state-of-the-art shows many research efforts to tackle challenges while designing embedded systems, by providing a range of DSE approaches to explore and evaluate extremely large set of design choices [1, 4, 5]. Either analytical [6], simulation [2, 7] or based on optimization algorithms [8–10], targeting

various platforms [11], or fitting diverse application domains, the key requirement of such DSE methodologies is to have a systematic approach to efficiently explore a large design space. The DSE at early design stages helps in rapid investigation of several parameters such as, architectural characteristics, application-to-architecture mappings, scheduling policies and hardware/software partitioning [12–14]. Performing DSE at a high level of abstraction helps traversing (typically) huge design spaces efficiently. As a result, it can quickly identify design candidates, which can satisfy design requirements, such as performance, high flexibility and low power. As the design progresses, the design space can be gradually trimmed and pruned of unsuitable design alternatives until the most suitable solution is obtained. Therefore, performing such early explorations enables design decisions to be made quickly, which can significantly reduce the overall design time of a system.

In recent years, reconfigurable architectures [15–17] have received an increasing attention due to their adaptability and short time-to-market. One of the main advantages of reconfigurable architectures is the ability to increase performance with accelerated hardware implementation, while possessing the flexibility of a software solution. These architectures can increase the performance of an application by mapping selected parts (application kernels) onto reconfigurable hardware. However, the design of such architectures is subject to numerous design requirements, such as performance, chip area, power consumption, and memory requirement. Several choices have to be investigated and evaluated before making any design decision. Therefore, having appropriate tools and methodologies to assist designers with the exploration and performance evaluation of such architectures at an early design stage, is of utmost importance. Towards this goal, the objective of the research proposed in this dissertation is to develop a system-level DSE framework for modeling and simulation of dynamically reconfigurable architectures. The framework can be deployed as a standard modeling and simulation tool, which can assist system designers to perform system-level DSE with respect to hardware-software partitioning, application-to-architecture mapping, task allocation and task scheduling of dynamically reconfigurable architectures.

1.1 Problem Overview

Over the last years, increasing consumer demand and processor technology growth have created unique opportunities for embedded systems to use a va-

riety of platform architectures, each designed to obtain certain goals. These architectures can be categorized into two main domains based on their performance and their degree of flexibility: the general-purpose computing domain and the application-specific computing domain. The general-purpose computing domain provides a high degree of flexibility in terms of the application area and rapid development. Despite of the high flexibility, the key drawback of the general-purpose computing domain is its ineffective performance while computing certain functionalities. On the other hand, the application-specific computing domain uses specific processors to perform a certain computation. Such processors are able to satisfy specific requirements by efficiently executing the target application(s). As a result, application-specific domain can provide better performance than general-computing domain while executing certain applications. An ideal computing technology, however, should combine the flexibility and the high performance of the aforementioned domains. Reconfigurable computing is becoming increasingly popular as it bears this promise - combining the flexibility of the general-purpose computing domain with the performance of application-specific computing domain.

The reconfigurable computing domain uses reconfigurable architectures, which are typically composed of a General Purpose Processor (GPP) and reconfigurable hardware, e.g. a Field Programmable Gate Arrays (FPGA)[1]. Computational intensive application fragments can be accelerated by executing them on the FPGA, while control intensive application fragments can be executed to the GPP. In this way, the architecture can adapt to the application, combining hardware performance with software flexibility. However, such architectures are subject to numerous requirements and design constraints, such as cost, resources, power consumption, timing constraints and dependability.

Dynamically reconfigurable systems can evolve under diverse conditions due to changes imposed either by the architecture, or by the applications, or by the environment. In such systems, the design process becomes more sophisticated as all design decisions have to be optimized in terms of runtime behaviors and values. Static exploration is not sufficient to accurately perform exploration of dynamically reconfigurable architectures, due to their changing and often unpredictable runtime conditions. With runtime exploration, a design candidate can be evaluated for (dynamically) changing system requirements. As a result, by performing a runtime exploration together with static exploration, reconfigurable system can be evaluated more efficiently in terms

[1] Xilinx [18] and Altera [19] are currently the key producers of FPGAs in the market.

of aforementioned requirements and constraints.

In this dissertation, we describe a system-level framework, which can assist designers in tackling several challenges while designing reconfigurable systems. The proposed framework combines static and runtime exploration in order to perform early-stage DSE with respect to hardware-software partitioning, task mapping, task allocation, and task scheduling for dynamically reconfigurable architectures. Before listing the major challenges addressed, and the core contributions of this dissertation in the next section, we first elaborate the problem more in detailed with an example.

Heterogeneous reconfigurable systems consist of a range of processing elements with multiple sets of design criteria, such as performance increase, chip area optimization, and power consumption minimization. For such systems, the design space that must be explored to arrive at a suitable design grows enormously due to the large number of design choices to be investigated. Let us assume that the application description and the architecture resources in such a system consists of T computational tasks and P distinct computational processors respectively, as shown in the top part of Figure1.1. Let us also assume that these application tasks have to be mapped onto the given processors. Each task can be potentially mapped onto every available processor. The mapping of a task onto a processor results in a unique mapping choice, which has a specific set of system attributes, such as performance, power consumption, chip area and memory requirement, as shown in the middle part of Figure1.1. The mapping of T tasks onto P processors results into P^T possible mapping choices, each having unique system attributes. As a consequence, an exhaustive evaluation of all potential mappings quickly becomes computationally intractable. The complexity further increases if multiple objectives are subject to the design evaluation, which is the case with most heterogeneous reconfigurable systems. Under such conditions, the design space quickly explodes as depicted in the bottom part of Figure 1.1. Additionally, the complexity increases when this evaluation has to be performed at runtime, where design decisions have to be optimized in terms of runtime behaviors/values and the current system status. Dynamically reconfigurable systems can change at runtime due to changes imposed either by the architecture, by the applications, or by the environment. Under such conditions, fast explorations have to be performed to satisfy changing conditions. As a result, performing DSE under such conditions becomes a challenging task.

In order to create an efficient design, it is essential to perform DSE to eval-

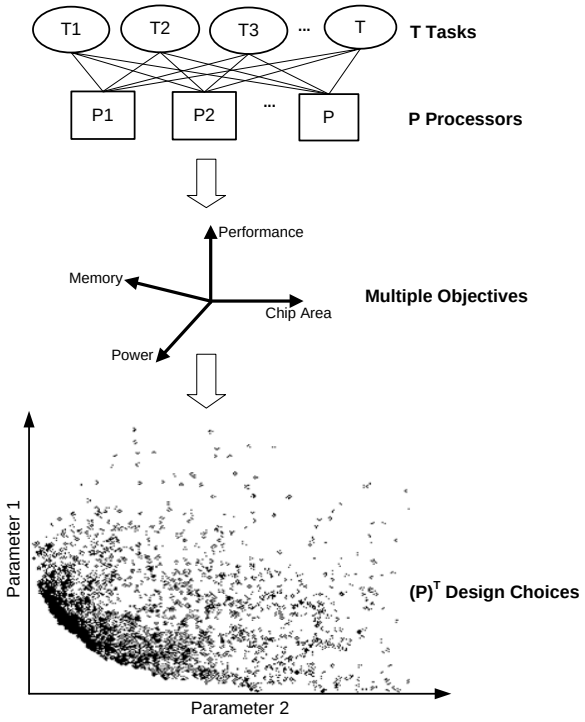


Figure 1.1: Design space explosion while mapping T tasks onto P processors. The mapping is subject to various system attributes such as, memory, performance, power and chip area. The explosion of the design space is highly affected by T and P numbers.

uate design choices before obtaining a suitable solution. In the traditional co-simulation methodology, detailed simulators are used for performing DSE with respect to hardware-software partitioning, application-to-architecture mapping, task allocation and scheduling [20]. Nevertheless, when the functionality demand or the architecture intricacy increases, due to enormous explosion of the design space, it is not feasible to evaluate all possible choices using such detailed simulators even with the most efficient and the fastest ones. To cope with such complexity it is essential to employ a system-level modeling and design methodology. A system-level modeling and design methodology aims to raise the abstraction level of the design process, and it simulates the system using abstract models for the application and the architecture. Such abstracted models are fast to create and easy to use, thus, they can traverse a large design space in an efficient manner. Using such models at a higher level of abstraction facilitates faster design decisions, which in turn can significantly reduce overall

design time. Towards this goal, we are interested in developing a system-level modeling and simulation framework in order to perform DSE with respect to hardware-software partitioning, application-to-architecture mapping, task allocation and task scheduling of dynamically reconfigurable systems.

1.2 Research Challenges

As mentioned before, performing DSE is essential in order to create an efficient design. Nevertheless, due to an enormous explosion of the design space, it is not feasible to evaluate all possible choices in a traditional way using a detailed simulator. As a result, in this dissertation we propose a methodology which can allow fast and efficient exploration of reconfigurable architectures. The main challenges addressed in this dissertation are the following.

Challenge 1. ***How to allow faster exploration of all possible design choices in heterogeneous reconfigurable systems at early design stages, in order to cope with the growing design complexity?***

In heterogeneous reconfigurable systems, the design space that must be investigated to arrive at a suitable solution grows enormously, due to the large number of design choices to be assessed. The traditional co-simulation framework often uses a detailed simulator for performing DSE of reconfigurable systems. One of the main disadvantages of using such detailed simulators for performing DSE at early stage is their high simulation time. Typically, it is inefficient (if not impossible) to explore and evaluate all possible design combinations using these simulators at early design stages, due to the large design space of such systems. When these simulators are applied for exploring such a large design space, they often suffer from low simulation speed, which hinders fast exploration that is essential in early design stages.

An obvious solution to address such issues is to raise the level of abstraction, in order to achieve faster solutions. By abstracting the specification of the system from the detailed design, we gain the ability to perform fast simulation and efficient synthesis of complex heterogeneous and reconfigurable systems. As a result, it is essential to provide a mechanism that can perform faster exploration of dynamically reconfigurable systems by using abstract models.

Challenge 2. ***How to deal with the runtime exploration of the heterogeneous reconfigurable systems, in order to efficiently evaluate them, and how to optimize design decisions in terms of their runtime behaviors and system status?***

Reconfigurable systems can evolve under diverse conditions due to changes imposed either by the architecture or the applications or the environment. A reconfigurable architecture can evolve under different conditions - for instance - processing elements shut-down in order to save power, or extra processing elements can be added in order to meet the execution deadline. The application behavior changes, due to the dynamic nature of the application - application load can change due to the arrival of sporadic tasks. In such systems, the design process becomes more sophisticated as all design decisions have to be optimized in terms of runtime behaviors and values, such as performance, power and memory. A static exploration under such conditions often results in compromised accuracy. As a result, it is essential to provide a mechanism to explore and evaluate reconfigurable systems for a (dynamically) changing system conditions at runtime.

Challenge 3. ***How to provide a generic modeling and simulation framework for DSE which allows designers to model and evaluate any kind of reconfigurable behavior at runtime?***

As mentioned in Challenge 2, reconfigurable systems have to be explored at runtime in order to optimize their designs in terms of runtime behaviors and values. Therefore, having appropriate tools to assist designers with exploration and performance evaluation of such systems at runtime is extremely important. Nevertheless, due to the lack of a standardized modeling and simulation framework for DSE that allows designers to model and evaluate reconfigurable systems' behavior at runtime, many research groups rely on their custom-built proprietary simulators. This results in the following problems:

- the evaluation of various design choices using these custom-built simulators is exceedingly complex and,
- the comparison between different evaluations received from these tools is also extremely difficult if not impossible.

Therefore, a standard modeling and simulation framework becomes es-

sential to evaluate reconfigurable systems, which can be re-used between research groups. Such a framework can provide a standard platform, allowing easy comparison between design evaluations, and hence it can also be used as a reference tool for future research. Such a standardized framework can be used as an excellent tool in the research domain of dynamically reconfigurable systems.

1.3 Research Methodology

This section of the dissertation provides a description of the research methodology used to conduct this research. The research methodology used in this dissertation consists of several phases as shown in Figure 1.2. As it can be inferred from the figure, the methodology starts with the background research and problem identification, followed by a conceptual solution and it implementation, case studies, and finally the result evaluation. Each research phase results with a specific outcome, which is passed as a starting point for the next phase. In the following, we will elaborate each of these phases in more detailed.

- **Background research and problem identification**

  In the first stage, we performed background research in order to identify the existing problem. The state-of-the-art related to the system-level design methodology and the DSE of reconfigurable architectures has been evaluated. The study suggests that performing system-level DSE at early design stages is an essential requirement in case of dynamically reconfigurable architectures, especially in the cases where application requirements and the architecture behavior can change. Furthermore, the research also suggests that there is no system-level framework for performing faster exploration at runtime for reconfigurable architectures. Developing such a framework can model, simulate and explore the reconfigurable architectures at runtime at early design stages such that the design time can be reduced.

- **Conceptual solution development**

  The runtime exploration of a design candidate can provide better accuracy as it takes into account certain feedback from the system. However,

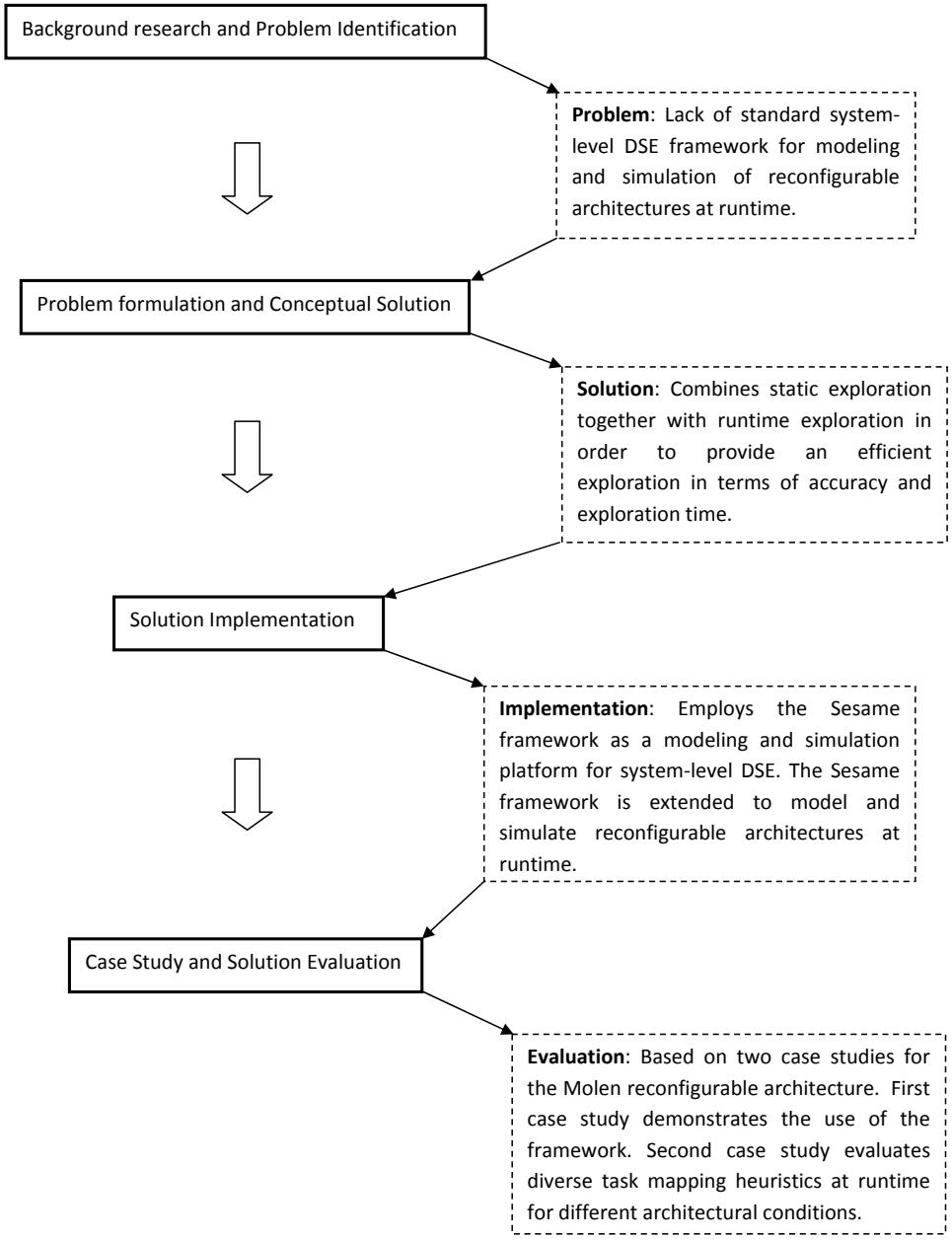


Figure 1.2: Research Methodology carried in this dissertation. Background research and problem identification is carried as the first step, followed by the conceptual solution and the solution implementation. At the end of the case study, solution evaluation is performed. The dashed boxes in the figure list the outcome of the each research phase.

one of the major problems with runtime exploration is its enormous design space generated by the runtime system parameters. As a result, performing runtime exploration of all the design candidates, at the runtime, results with slow exploration. Static exploration performed under offline system can generally be fast. In order to benefit from both static and runtime exploration, we propose a two-level exploration methodology, which combines a static mapping exploration together with a runtime mapping exploration. At first, the static mapping exploration performed under static conditions leads to a set of candidate mappings. After that, the runtime mapping exploration performs a high quality exploration of these candidate mappings to address any runtime change in the application, in the architecture, or in the environment.

- **Solution implementation**

  We developed a framework, called rSesame, which implements the two-level exploration methodology described in the previous bullet. The rSesame framework employs the Sesame framework [2, 21] as a modeling and simulation platform for system-level DSE. To this end, we performed two major extensions on the Sesame framework. Firstly, it is extended to model and simulate the behavior of reconfigurable architectures. In the second extension, we extended the Sesame framework to allow runtime exploration of various architecture-application-mappings of such architectures.

- **Case study and result evaluation**

  We perform two case studies using the rSesame framework. In the first case study, we demonstrate the use of the framework to perform static and runtime mapping exploration. For this we instantiate a model from the rSesame framework for mapping an Motion-JPEG (MJPEG) application onto the Molen [22, 23] reconfigurable architecture. Mapping application only onto the reconfigurable hardware can give better performance, however it consumes more hardware resources. With the runtime mapping, a tradeoff can be obtained in terms of performance and resources. With this case study, we want to show that the rSesame framework can assist to explore such tradeoffs. We show that the model can be efficiently used to perform several design parameters, such as execution time, area usage, number of reconfigurations, and percentage of hardware/software execution. The runtime exploration is evaluated and

compared against static exploration based on the various design parameters obtained from the model.

In second case study, we implement diverse task mapping heuristics for mapping an extended MJPEG application onto the Molen reconfigurable architecture using the rSesame framework. These task mapping heuristics are compared and evaluated based on various architectural exploration parameters obtained from the framework. These explorations are performed under different architectural conditions, where different FPGAs are used to evaluate a number of application-to-architecture mappings under different resource conditions. With this case study, we want to show that the rSesame framework is flexible and can efficiently assess various runtime mapping heuristics in terms of various design parameters under different resource conditions.

1.4 Dissertation Contributions

In this dissertation, we focus on the design of system-level modeling and simulation framework for dynamically reconfigurable architectures to address the aforementioned challenges. The main contributions of this thesis are listed as following.

- ***The proposal of a two-level DSE approach for runtime mapping exploration of dynamically reconfigurable architectures.***

  Static exploration is often performed under *static* system condition, where any changes in the system, such as the application, the architecture or the environment, are not considered during the exploration process. Such exploration can generally be faster, but it may be less efficient, as it does not take into consideration the runtime behavior of the system. Due to changing runtime conditions with respect to e.g. user requirements or having multiple simultaneously executing applications competing for platform resources, static exploration of a system alone is not adequate for any kind of architectural exploration. With runtime exploration, a design candidate is evaluated for (dynamically) varying system constraints, and as a result, any changes in the system are given as a feedback to the evaluation process. Runtime exploration can provide better accuracy compared to static exploration, but is typically hard to

obtain due to the large size of the search space generated by the runtime system parameters.

We propose a two-level approach for performing DSE of reconfigurable architectures, which combines both static and runtime mapping exploration. The static mapping exploration performed, in the first step, leads to a set of candidate mappings, and the runtime mapping exploration, carried in the second step, performs a high quality evaluation of these candidate mappings at runtime, in order to optimize them according to any change in the system conditions. In this way, fast exploration of the static exploration is combined with a detailed evaluation at runtime, to obtain more efficient DSE.

- ***The proposal of a system-level modeling and simulation framework for dynamic reconfigurable architectures, which allows exploration and evaluation of dynamically reconfigurable systems at runtime.***

  The current state-of-the-art on system-level DSE efforts for reconfigurable architectures, typically, focuses on providing system-level modeling and simulation framework that can perform rapid exploration of various reconfigurable design alternatives, only considering static system conditions [24–26]. To the best of our knowledge, there is no existing standardized modeling and simulation infrastructure for DSE, which allows designers to model and evaluate the reconfigurable systems' behavior at runtime. In order to fill this gap, we propose a system-level modeling and simulation framework, called *rSesame*, for performing DSE of dynamically reconfigurable architectures. The proposed framework allows the modeling, simulation and evaluation of reconfigurable architecture, and to carry out aforementioned two-level DSE approach of such architectures.

  The proposed framework employs the Sesame framework [2, 21] as a modeling and simulation platform for system-level DSE. In the context of Daedalus framework [27], Sesame is used to perform static exploration of multimedia Multiprocessor System-on-Chip (MPSoC) architectures. In this context, typically, the Sesame framework is limited to model MPSoC architectures, and to perform exploration of various application-to-architecture mappings at static time. In our context, we performed two major extensions on the Sesame framework. Firstly, Sesame is extended to support the modeling of the reconfigurable archi-

tectures. Secondly, the Sesame is extended to perform application-to-architecture mapping exploration of such architectures at runtime. The detailed description on how the Sesame framework is extended to model reconfigurable architectures at runtime is provided in Chapter 4.

The rSesame framework is a generic system-level framework for performing DSE of dynamically reconfigurable architectures, targeting streaming applications from the multimedia domain (e.g. JPEG, MJPEG and MPEG codecs). The main characteristic of streaming applications is that they are data-flow oriented applications, i.e. large streams of data have to be processed. As a result, for the application modeling, the rSesame framework uses Kahn Process Networks (KPNs) [28] at the granularity of coarse-grain tasks. A KPN consists of concurrent processes with explicit communication over FIFO channels. KPNs are deterministic and can conveniently capture the parallel and dynamic nature of streaming applications in the multimedia domain, which is our target domain. The rSesame framework can perform system-level DSE with respect to hardware-software partitioning, application-to-architecture mapping, task allocation and task scheduling, both statically and at runtime. A number of design attributes, such as execution time, area usage, number of reconfigurations, percentage of hardware/software execution and hardware reusability efficiency can be obtained from the framework. As we will explain in the next chapters, the rSesame framework strives for several characteristics, such as flexibility, ease of use, fast performance and its applicability to a wide range or reconfigurable systems.

- ***The use of the rSesame framework to evaluate dynamically reconfigurable architectures both statically and at runtime, based on important system attributes.***

  We present a case study with the rSesame framework by instantiating a model for the Molen reconfigurable architecture [23]. The instantiated model is employed to show that the rSesame framework can be efficiently used to perform DSE with respect to application-to-architecture mappings for the Molen architecture both statically, where the system behavior is fixed, and at runtime, where the system can change at runtime. These mappings are evaluated based on various design attributes obtained from the model, such as execution time, area usage, number of reconfigurations, percentage of hardware/software execution, percentage of reconfiguration and hardware reusability efficiency. The obtained

results show that mapping all tasks onto a reconfigurable hardware gives better performance, but it consumes more hardware resources. With the runtime mapping, a tradeoff can be obtained in terms of performance and resources. The rSesame framework can be efficiently used to investigate and appraise such tradeoffs.

- ***The evaluation of the characteristics of the rSesame framework by showing that the framework can easily and quickly model, simulate and compare a wide range of task mapping heuristics at runtime.***

  We describe a case study to show the characteristics (flexibility, ease of use, fast performance, and applicability) of the rSesame framework tested on the Molen architecture by evaluating and comparing a wide range of task mapping heuristics at runtime. These heuristics are obtained from diverse domains and they are evaluated under different architectural conditions. The case study shows that the rSesame framework can be efficiently deployed to model, simulate and compare a wide range of mapping heuristics from diverse domains based on various design attributes, under different architectural conditions.

In this research, we do not focus on developing techniques for system-level hardware-software partitioning, task mapping, task allocation, task routing and task scheduling. We also do not provide a high-level or a low-level design for reconfigurable architectures to improve the technology behind these architectures. Finally, the implementation of the designs on reconfigurable architectures to achieve high performance is also not addressed in this research. The focus of this research is to develop a generic system-level framework for modeling, simulation and evaluation of dynamically reconfigurable architectures at early design stages. As a matter of fact, the proposed framework is not limited to a type or a class of reconfigurable architectures. The framework is generic and can be applied to evaluate all kinds of heterogeneous dynamically reconfigurable architectures.

1.5 Dissertation Outline

The contributions of the dissertation are presented over several chapters. The dissertation consists of four core chapters, each focusing on different aspects. The organization of the dissertation is as follows.

Chapter 2 provides the necessary overview of the state-of-the-art in the context of reconfigurable architectures and the system-level design methodology. The chapter discusses reconfigurable architectures and their challenges in the context of the embedded system domain. It also describes the early DSE and the system-level design methodology in the context of reconfigurable architectures. Finally, the chapter surveys and classifies DSE tools and methodologies for such architectures.

Chapter 3 presents an overview of the proposed runtime mapping exploration approach for reconfigurable architectures. Firstly, the chapter provides a detailed discussion of both static and runtime application mapping. Secondly, the chapter provides an outline of the two-level mapping exploration, which is based on the combination of static and runtime application mapping. At the first level, static exploration identifies a set of mappings. At the second level, these mappings are optimized at runtime to address any changes in the system.

Chapter 4 describes the rSesame framework, which can realize the two-level mapping exploration discussed in Chapter 3. The framework is a generic system-level modeling and simulation framework, which can model, simulate and evaluate reconfigurable systems statically and/or at runtime. The chapter discusses the methodology behind the rSesame framework, and it also lists the key features of the framework.

Chapter 5 describes a case study to show the characteristics of the rSesame framework tested on a real reconfigurable architecture. The chapter discusses a model instantiation from the rSesame framework for a dynamically reconfigurable architecture. The instantiated model is employed to perform both static and runtime mapping exploration of the given architecture. The chapter shows that the model can be efficiently used to perform exploration of many design parameters, such as execution time, area usage, number of reconfigurations and percentage of hardware/software execution, percentage of reconfiguration, and hardware reusability efficiency.

Chapter 6 provides an evaluation of the rSesame framework by studying and comparing various mapping heuristics at runtime, under different resource conditions, based on a number of design attributes. The chapter evaluates characteristics of the rSesame framework by showing that the framework can easily and quickly model, simulate and compare a wide range of runtime mapping heuristics from diverse domains.

Finally, **Chapter 7** provides concluding remarks on the work presented. This chapter summarizes the dissertation, outlines the main conclusions, followed by a number of recommendations intended to strengthen the vision of the system-level DSE of reconfigurable architecture in the academic and the industry.

This dissertation work has resulted with several numbers of reviewed publications. The content of this dissertation is based on these publications. At the end of each chapter, we list the paper(s) on which the content of that chapter is based on. Furthermore, at the end of this dissertation, a list of all the publication has also been provided.

2

Design Space Exploration of Reconfigurable Architectures

The ever increasing intricacy of the functionalities, as well as the increasing use of reconfigurable heterogeneous resources significantly complicates the design of modern embedded systems. Application complexity increases, either due to large systems with legacy functions, or due to a mixture of event-driven and data flow tasks. Similarly, architecture complexity intensifies, due to the design of heterogeneous and reconfigurable architectures, which consists of mixture of different technologies, processor types and design styles. These systems are subject to numerous constraints and design objectives such as cost, resource constraints, power consumption, timing constraints, and dependability. As a result, the design of heterogeneous reconfigurable systems imposes several challenges to system designers. These challenges include hardware-software partitioning, Design Space Exploration (DSE), application-to-architecture mapping, and application scheduling, among others.

This chapter presents a background study of the various issues involved during the design of the aforementioned systems, and the common solutions available to tackle these design issues. Section 2.1 gives a brief overview of heterogeneous reconfigurable architectures, and the challenges involved while designing such systems. Section 2.2 starts with the description of the system-level design methodology, and it continues further with an overview of the early stage DSE process. Section 2.3 presents a thorough analysis of tools and methodologies for DSE in the context of reconfigurable architectures. Finally, Section 2.4 presents the summary, and some concluding remarks.

2.1 Reconfigurable Architectures

The general-purpose computing domain is based on the Von Neuman computing paradigm. This domain is designed for general computing and provides a high degree of flexibility in terms of the application domain. The main advantage of the Von Neuman paradigm is its flexibility and programmability [29]. Despite of the high flexibility, the fact that algorithms must be sequentially programmed to run on a Von Neuman machine, such as a General Purpose Processor (GPP), many applications, especially highly parallelizable ones, cannot be executed with such machines to their potential best performance. On the other hand, the application-specific computing domain uses specific processors, e.g. Application-Specific Integrated Circuit (ASIC), Application-Specific Instruction-set Processor (ASIP), and Advanced Digital Signal Processor (ADSP), which are designed specifically to perform a certain task computation, and as a result, are able to satisfy specific requirements. The use of specific processors to execute the target application(s) is usually faster and more efficient compared to a GPP based approach. Nevertheless, this performance and efficiency comes at the price of low flexibility and low programmability, as any modification in the function implies redesign and re-fabrication of the hardware, which increases the production cost.

In an ideal case, a computing domain should provide the performance together with a certain level of flexibility. The reconfigurable computing domain [15–17, 30] attempts to combine these benefits, the performance of the hardware execution and the flexibility of the software execution. As illustrated in Figure 2.1, the reconfigurable computing domain can be placed after the application-specific computing and the general-purpose computing domains, when comparing them in terms of flexibility. Reconfigurable architectures use reconfigurable hardware, such as Field Programmable Gate Arrays (FPGAs) [31, 32] or other programmable hardware (e.g. CPLD - Complex Programmable Logic Device [33], reconfigurable Datapath Array - rDPA [34]) to accelerate algorithm execution by mapping compute-intensive calculations onto them. These hardware resources are frequently coupled with a core processor, typically a GPP, as a host processor. The GPP is responsible for controlling the reconfigurable hardware. Parts of application's operation are executed on the GPP, while the rest are executed on the hardware. In general, the hardware implementation of an application is more efficient in terms of performance than a software implementation. By executing selected application part(s) with hardware, the performance of the whole application can

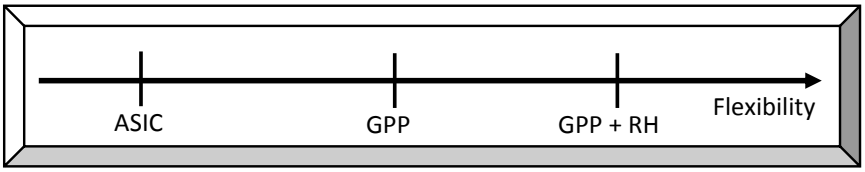


Figure 2.1: Positioning of different computing domain in terms of flexibility. The reconfigurable computing domain combines General Purpose Processor (GPP) and reconfigurable hardware represented by (GPP + RH) in the figure.

be improved. As a result, reconfigurable architectures enhance the whole application through an implementation of selected application kernels onto the reconfigurable hardware, while preserving the flexibility of the software execution with the GPP at the same time.

The reconfigurable hardware is composed of a set of programmable logic blocks, which are customizable at runtime. These logic blocks are connected using a set of routing resources, which are also reconfigurable. An application can be executed on the reconfigurable hardware by programming the logic gates within logical blocks, and using the configurable routing to connect the blocks together to implement the necessary functionality of the program [35]. These logical blocks are easily customizable, and as a result, they can be programmed to execute different functionalities at different instance of time. The main advantages of reconfigurable architectures are:

- changing an existing architecture rather than defining a completely new one. As a result, existing tools can be partially tuned to fit the new purpose
- support for a wider range of applications compared to application-specific processors - reconfigurable architectures have been shown to accelerate a variety of algorithms from different application domains, such as pattern matching, video streaming, signal processing and supercomputing.
- rapid prototyping - reconfigurable architectures can be used for rapid prototyping of different versions of the final product.
- in-system customization - provides capabilities to upgrade in the field by changing the configuration.

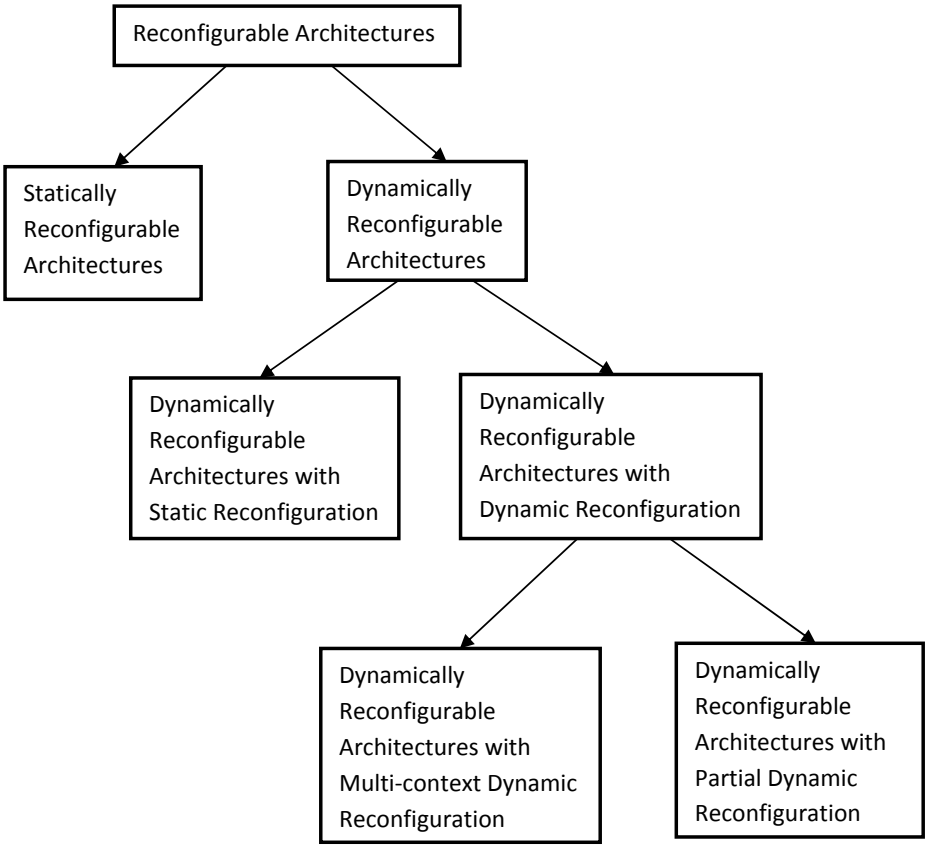


Figure 2.2: General classification of reconfigurable architectures based on their reconfiguration style. These architectures can be configured statically and dynamically.

Reconfigurable device can serve as a runtime re-usable hardware device for performance critical systems, which allows the reduction of the required hardware resources [16]. The reconfiguration nature of such devices provides the ability to change its structure at start-up-time as well as at runtime [30]. Consequently, various application kernels can run on the same chip at different time instances. This offers a great flexibility to accelerate large portions of an application. However, it also introduces a reconfiguration overhead.

The state-of-the-art shows a various trend in reconfigurable architectures. We can generally categorize these architectures into various classes based on their reconfiguration style as shown in Figure 2.2. There are mainly two types of reconfigurable architectures: ***statically reconfigurable*** architectures and ***dy-***

namically reconfigurable architectures. In the following, these architectures are described in more detail.

2.1.1 Statically Reconfigurable Architectures

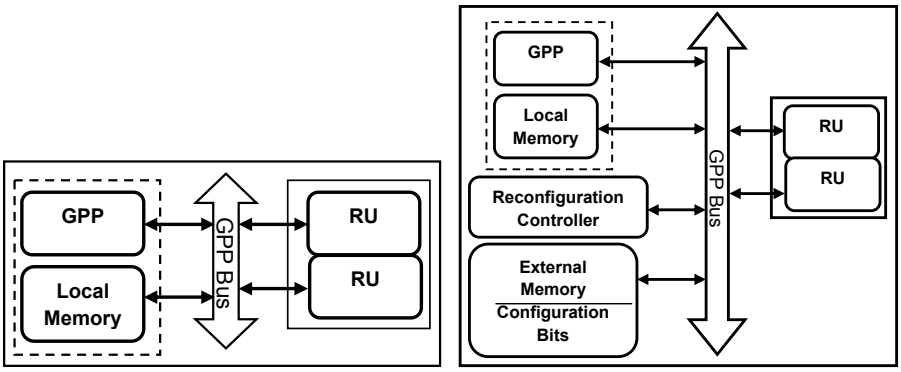
In statically reconfigurable architectures [16], the reconfigurable device is configured only at the beginning of the execution, and it remains unchanged for the duration of the application run. In order to reconfigure a reconfigurable device in such architectures, the system has to be halted while the reconfiguration is in progress, and then restarted with the new configuration. An example of such architecture is depicted in Figure 2.3(a), which consists of a GPP and reconfigurable device, e.g., FPGA, connected via a shared bus. The application to be run on the reconfigurable device is implemented in one or more Reconfigurable Units (RUs) as depicted in the figure.

2.1.2 Dynamically Reconfigurable Architectures

Unlike statically reconfigurable architecture, dynamically reconfigurable architectures [23, 35] allows runtime reconfiguration of the reconfigurable device. As a result, the reconfiguration and execution can proceed at the same time. An example of such architecture is depicted in Figure 2.3(b), where RUs can be configured at runtime. The reconfiguration data is downloaded from the external memory, and it is loaded onto the corresponding RUs at runtime. The reconfiguration controller is responsible for performing this task at runtime.

The runtime reconfiguration in dynamically reconfigurable architectures can be performed in two ways, ***static reconfiguration*** and ***dynamic reconfiguration***, as described in the following.

- In ***static reconfiguration***, the reconfigurable device can be configured when the system is running but the device is not running. In this case, the hardware device is not active during the reconfiguration process. When a partial design is sent to the device, the rest of the device is stopped and resumed only after the configuration is completed. An example of *static reconfiguration* model is depicted in Figure 2.4(a). In order to load any incoming configuration data in a part of a reconfigurable device e.g. FPGA, the whole device has been configured. This mechanism is



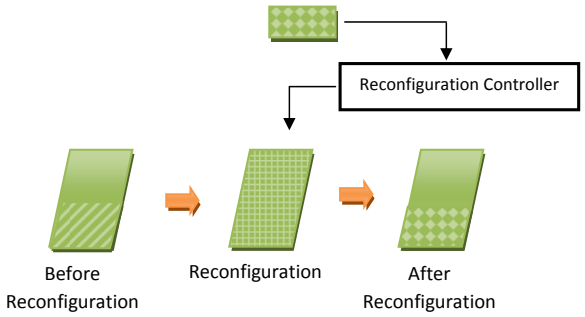
(a) Statically Reconfigurable Architecture (b) Dynamically Reconfigurable Architecture

Figure 2.3: An example of a) statically and b) dynamically reconfigurable architecture, which consists of a General Purpose Processor (GPP) and reconfigurable device, such as an FPGA. The reconfigurable device consists of one or more Reconfigurable Units (RU).

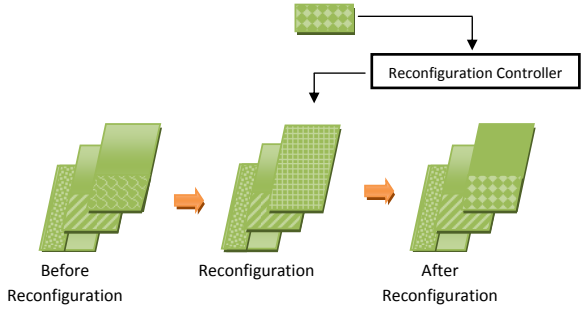
depicted in the figure, by showing the status of the device, before and after the reconfiguration.

- On the other hand, the ***dynamic reconfiguration*** allows the reconfigurable device to be configured when the device is running. It allows overlapping of the computation of one part of an application with the reconfiguration of another part of the same application. As a consequence, it can reduce reconfiguration overhead. The dynamic reconfiguration is further possible in two ways: *multi-context* reconfiguration (as available in Xilinx FPGAs [36]) and *partial* reconfiguration (as available in Xilinx [36] and atmel FPGAs), as discussed in the following.

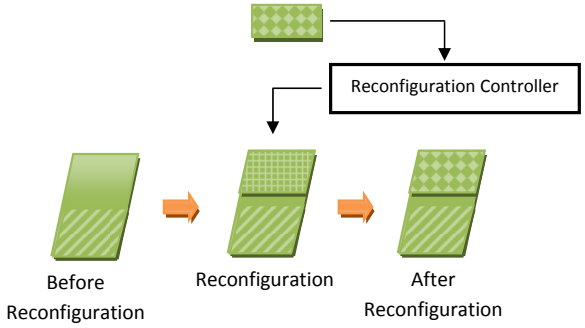
  – *Multi-context* reconfiguration groups configurations into contexts (typically), and the reconfiguration is performed based on the context. A reconfigurable device consists of multiple memory bits for each programming bit location. These memory bits can be thought of as multiple planes of configuration information. Only one plane of configuration can be active at a given moment, but the architecture can quickly switch between different contexts of already programmed configurations. An example of the *multi-context* reconfiguration model is depicted in Figure 2.4(b), where only one context of the reconfigurable device is swapped and configured as



(a) Static Reconfiguration



(b) Multi-context dynamic reconfiguration



(c) Dynamic partial reconfiguration

Figure 2.4: Different basic models of runtime reconfigurations in dynamically reconfigurable architectures. The reconfiguration controller is responsible for loading configuration data onto the reconfigurable device (e.g. FPGA) at runtime. The runtime reconfiguration can either be performed at static time, when the device is not active, or at dynamic time, when the device is still running.

required.

– In *partial* reconfiguration [36, 37], configurations do not occupy the full reconfigurable device and only a part of the device can be configured. While the device is still active and critical parts of the design are still operating, the controller can load a partial design into another part of the reconfigurable module. The concept of partial reconfiguration permits to change the structure of only a part of the reconfigurable hardware, while the rest is still running. The *partial* reconfiguration model is depicted in Figure 2.4(c), where only a part of the device is configured, while the rest is still running another configuration.

The emergence of dynamic partial reconfiguration has added new dimensions to the reconfigurable computing domain. However, it has also added a new set of challenges to designers while evaluating such systems. The exploration and partitioning problem for such systems is not only limited to assignment and scheduling of tasks to the set of fixed hardware and software resources. It also has to address the issue of how the reconfigurable space can be efficiently used. This introduces to a number of other problems, such as task allocation and task placement, among others. In the following, the issues involved in the design of heterogeneous dynamically reconfigurable architectures are listed and discussed in more detail.

2.1.3 Hardware-Software Partitioning

One of the major problems with heterogeneous reconfigurable systems is the identification of which application tasks should be implemented onto which resources, in order to improve a particular design metric, such as faster execution, reduced power consumption, and lower hardware cost. Hardware-software partitioning of reconfigurable systems can be categorized into: the *spatial partitioning* problem and the *temporal partitioning* problem.

2.1.3.1 Spatial Partitioning

The spatial partitioning deals with the allocation of application tasks onto different types of processors. The spatial partitioning places the tasks onto differ-

ent physical areas: the GPP (SW tasks) or a reconfigurable hardware unit (HW tasks). The spatial partitioning is driven by various objectives, of which some are the following:

- The basic objective of the spatial partitioning is to achieve a performance gain. One way to achieve this goal is to place tasks with high execution latency onto reconfigurable hardware and to execute tasks with lower execution latency as software. In this way, tasks with higher latency can be accelerated with a hardware implementation in order to increase the performance. Another way to achieve a performance gain is to minimize the communication between hardware and software domains. Minimizing the transfer of data between hardware and software may again achieve a performance increase.

- Another objective of spatial partitioning is to efficiently utilize the reconfigurable hardware resources. For example, by placing only the tasks, those are more beneficial to execute in hardware, on the reconfigurable unit, the tradeoff between hardware cost and performance can be achieved.

The criteria mentioned above are not always orthogonal. For instance, mapping a task with high execution latency to hardware can increase the communication between hardware and software domains. Nevertheless, it may also increase the amount of required resources. Various techniques of different complexities for spatial partitioning of tasks onto heterogeneous reconfigurable architectures can be found in the literature. These techniques focus on achieving one or more of the goals discussed above. A few examples are presented in [38–42].

2.1.3.2 Temporal Partitioning

The reconfiguration capability of modern reconfigurable devices allows mapping application tasks that are larger than the available hardware resources. To implement applications of which the requirements exceed the available hardware capacity, it is necessary to perform temporal partitioning. The temporal partitioning places tasks in separated time slots (temporal independence), sharing the same physical resources on the reconfigurable hardware. It divides a design into mutually exclusive, limited sized segments, such that the logic requirements for implementing a segment is less than or equal to the capacity

of the available hardware. These segments are called hardware *configurations*, and they can be sequentially executed on the reconfigurable hardware. The temporal partitioning is driven by various objectives, of which some are the following:

- The minimization of the communication between different hardware *configurations*. This can reduce the communication overhead between different hardware configurations, which in turn, can speedup the whole application.

- The minimization of the reconfiguration latency. Reconfiguration latency can be avoided by a careful selection of the tasks to load onto the reconfigurable device. This, in turn, can significantly improve the performance of the whole applications.

Nonetheless, these objectives also compete with each other. For instance, minimizing the communication between hardware configurations can decrease the load balancing between these configurations. A number of techniques for temporal partitioning of tasks onto sets of reconfigurable hardware have been presented in [10, 43, 44].

2.1.4 Task Allocation and Placement

With the partial dynamic reconfiguration of dynamically reconfigurable architecture, tasks can be swapped in or out of the hardware individually at runtime, without interrupting other tasks running on the same hardware. Such feature allows better flexibility and device utilization. However, it creates a new challenge for task placement at runtime. Task placement deals with placing a set of HW tasks onto reconfigurable hardware in different stages. The reconfigurable devices are subjects to various constraints such as area, memory requirement and power consumption. As a result, placement of tasks onto such reconfigurable devices involves various challenges in order to satisfy the given system constraints. Several techniques for allocation and placement of tasks onto set of reconfigurable hardware have been presented in [45–48].

2.1.5 Task Scheduling

Task scheduling is another problem which deals with scheduling of various tasks on architecture. The SW tasks, the HW tasks and the communication channels shared by different architectural components (such as the GPP and the reconfigurable hardware) have to be scheduled. This scheduling has to be done at runtime in order to avoid resource conflicts and to meet the execution requirements of the application. The task scheduling becomes complicated in case of partial reconfigurable systems, where few HW tasks can be configured onto the reconfigurable hardware, while other tasks are still running on the same device. In such systems, the efficiency of task scheduling directly impacts the overall system performance. Several mechanisms for task scheduling onto dynamically reconfigurable hardware with partial dynamic reconfiguration have been presented in [49–52].

2.2 Design Space Exploration

The increasing intricacy of the application functionalities and, at the same time, the escalating use of heterogeneous reconfigurable platforms has significantly enlarged the design space of modern systems. Several choices have to be evaluated and judged for executing any decision at each stage of the design. Therefore, it is essential to perform DSE at every design stage, starting from the very early ones, in order to investigate tradeoffs between all possible design goals, and to select the most appropriate solution. As the design progresses, these design choices can be gradually refined at subsequent stages in order to find the final "optimal" solution. DSE is the process of analyzing functionally equivalent alternative design points, either architecture and/or application, in order to identify the "optimal" design point. These design points are determined based on the various system constraints imposed on the system. An example tradeoff for such design can be between the best execution time and the used area.

The main goal of DSE is to strengthen the synergy between the application and the architecture to attain an improvement of a particular design metric, such as area, performance and power. A number of dedicated tools (generic or specific), methodologies and environments can be employed to explore the design space, to select the best architecture and/or application characteristics.

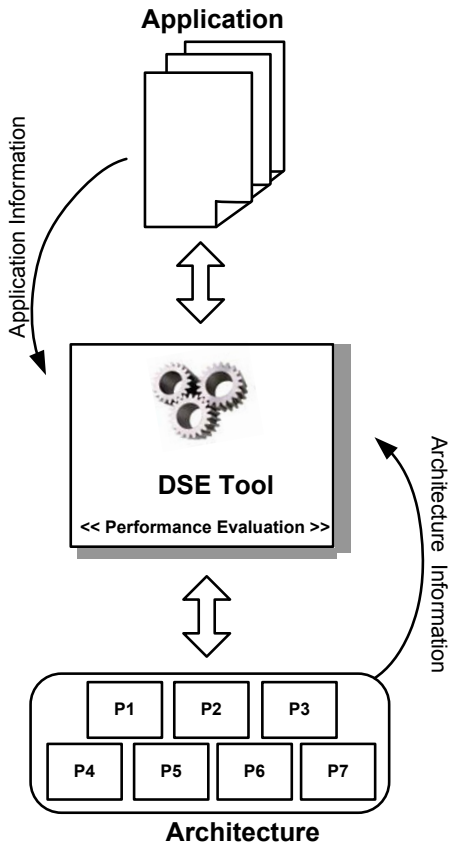


Figure 2.5: Design Space Exploration (DSE) process between application specification and architecture characterization. Such a DSE process contains both the application requirements and the architectural constraints.

The major challenge of such DSE methodologies is to take advantage of the application models and choose the best architecture platform available for that application. As a result, a DSE method has to take into account not only the application information, but also the types of resources in the architecture, their number and their timing information. To enhance the conjunction between the application and the architecture, DSE environments link the application requirements and the architecture constraints together. Figure 2.5 depicts such an instance of a DSE process between an application specification and an architecture characterization. Such DSE environments assist in the rapid performance evaluation of different parameters, such as architectural characteristics, application-to-architecture mapping, scheduling policies and hardware-

software partitioning. Consequently, they enable a designer to identify a design candidate, which satisfies various design attributes, such as performance, chip area, memory and power consumption.

2.2.1 System-level Design and Early Stage DSE

In the traditional design process, designers often start with an informal specification of the system. This is followed by a manual or semiautomatic generation of several alternative designs, which are subjected to a series of time-consuming and, typically, ad-hoc evaluations. At the end of such evaluations, the most suitable design is chosen to be synthesized into an architecture implementation. With the complexities of heterogeneous reconfigurable systems rising almost daily, such traditional design processes, when applied to the design of these systems, results in an inefficient design methodology.

The co-simulation framework that is based on the classical design approach generally starts from a single system specification, which is gradually refined and synthesized into an architectural implementation. In such approaches, typically, different types of simulators are employed for the simulation of the application and the architecture. For example, an instruction-level simulator can be used for the simulation of the application, and the architecture can be defined using VHDL [53] or Verilog [54] and the corresponding cycle accurate simulator. The major disadvantage of using such detailed simulators for performing DSE at an early stage is their high simulation time. Due to the enormous design space of heterogeneous reconfigurable architectures, it is almost impossible to evaluate all possible designs at early design stages even while using the most efficient simulator. Furthermore, it uses a single system specification to describe the application and the architecture, which makes the design process inefficient and incomplete.

To cope with the design complexity of modern heterogeneous and reconfigurable systems, it is essential to have a methodology that can handle the given complexities with increased productivity and decreased time-to-market. An obvious solution to address such issues is to raise the level of abstraction, in order to quickly achieve the solution. By abstracting the specification of the system from the detailed design, we can gain the ability to perform fast simulation and efficient synthesis of complex heterogeneous and reconfigurable systems. This had lead to the emergence of a modern design methodology, called *system-level design methodology* [55], which tries to overcome the aforemen-

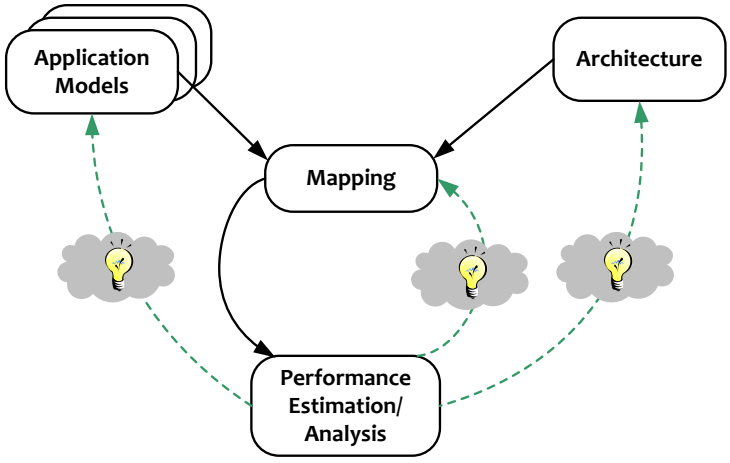


Figure 2.6: The Y-chart design scheme. The scheme shows a strict separation between architecture, applications and mapping/performance analysis.

tioned shortcomings of the classical co-simulation framework. The system-level design methodology aims at raising the abstraction level of the design process in order to simulate the application and the architecture using abstract models. The system-level design methodology incorporates ideas from *Y-chart scheme* [56, 57]. The Y-chart scheme is based on the *separation of concerns* principle, which typically separates various aspects of design. The separation of various aspects of the design allows for more effective exploration of alternative implementations. One fundamental separation in design process suggested by the Y-chart scheme is the separation of the application behavior and the architecture constraints. Additionally, the separation in computation and communication behavior in the application can also be realized using the Y-chart scheme.

The Y-chart design scheme is visualized in Figure 2.6. As it can be seen from the figure, the Y-chart design scheme recognizes a clear separation between an application model and an architectural model. The application and the architectural models are associated with each other using an explicit mapping layer. The application model describes the functional behavior of an application, which is independent of architectural specifics. Similarly, the architecture model defines the architecture resources and captures their timing characteristics. Both models (application and architecture) can be mapped onto each other, by means of well defined mapping methods. The performance and estimation are then obtained through the analysis of different mapping in-

stances, which consist of different applications-architecture pairs. Different instances of this mapping can be evaluated and compared. These estimations are available as a feedback to revisit certain decision through a feedback to any of the three blocks as indicated by light bulbs in Figure 2.6. This may inspire the designer to improve the architecture and/or the application, or change the mapping. The separation of application and architecture modeling, as suggested by the Y-chart, allows designers to use a single application model to map them onto a range of architecture models representing different instances of a single platform, or the same platform instance at various abstraction levels. Such a capability, clearly demonstrates the strength of decoupling the application and the architecture model, encouraging the reuse of both types of models.

With the emergence of the system-level design, it is possible for designers to model the system at the early design stages and to evaluate their performance. System-level design typically uses models that capture the input application, the target architecture and the system constraints, such as latency, throughput and energy dissipation. These high level models minimize the modeling effort and are optimized for faster execution. As a result, they can be applied during early stages of the system design of complex heterogeneous systems. The advantages of the system-level design methodology includes: a) easy to design and validate, b) quick system evaluation, and c) low cost.

In the DSE framework based on system-level design methodology, designers use system-level models to investigate and explore the system. These models are relatively easy and fast to construct, and as a result, designers can apply this to traverse a larger design space at early design stages. System-level DSE helps in the rapid performance evaluation of different parameters, such as architectural characteristics, application to architecture mappings, scheduling policies and hardware-software partitioning. Thus, performing DSE at a higher level of abstraction facilitates design decisions to be made at the very early stages. This, in turn, can significantly reduce overall design time.

Figure 2.7 depicts the abstraction levels in DSE. The multiple design criteria in the system design result in an exponential explosion in the design space. Furthermore, this is also subject to the multi-dimensional objective space. In the higher levels of DSE, a huge design space must be explored in a short period of time. It is, therefore, important to prune the design space using fast (and often heuristic) methods. Models at this level have a higher level of abstraction and these are easier to build, but they are less accurate. As design

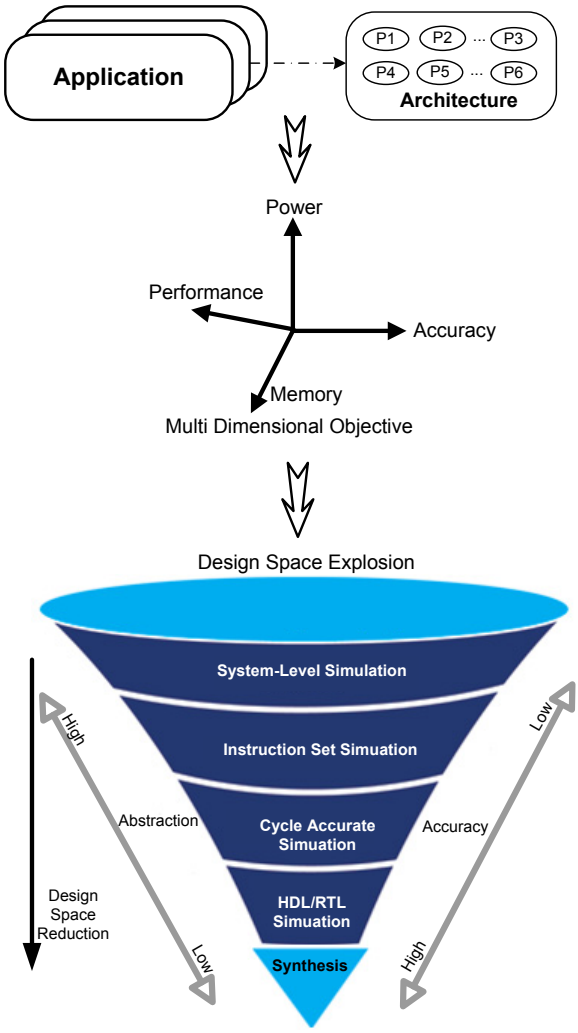


Figure 2.7: Different levels of Design Space Exploration. Models at the top of the funnel are more abstract and less accurate, but they are easy and quick to build. Conversely, models at the bottom are more detailed and accurate, however they are more difficult and time consuming to build [14].

progresses, more refined and accurate (and thus more expensive) evaluation algorithms (e.g. simulation) can be applied to a reduced design space. Models that are used at the lower level incorporate more details, and are more difficult to build. Nevertheless, they are more accurate. After performing initial calculations, designers propose various solutions. These solutions are evaluated

and compared at a high level of abstraction, and a set of candidate solutions is identified. The candidate set can be further analyzed at a lower abstraction level, such as the transaction level and cycle accurate level, and a more refined set of solutions is identified. Later, the refined solution set can further be analyzed at the synthesizable Register Transfer Level (RTL), in order to reach an "optimal" solution. In this way, higher abstract models can be used to explore the large design space at the early stages and more detailed models at the later stages convey more implementation details and subsequently attain better accuracy.

2.2.2 System-Level Simulation and Exploration Tools

Over more than a decade, the system-level design methodology has become an important research field, trying to improve the embedded system design process. Various system-level design environments have been developed in order to provide such improved system design. SystemC [58] is a simulation driven system description language and has become the most popular language for system-level design. It is widely used for modeling both hardware and software at various levels of abstraction in many application domains, such as embedded software, SoC, multicore systems and reconfigurable architectures.

The system-level design framework presented in [59, 60] is a systemC based simulation environment for function/architecture co-design for Network-on-Chip (NoC) system. The framework enables the quantitative evaluation of application-to-platform mappings by means of an executable performance model [61]. The methodology presented in [62] is also a SystemC based design methodology for system-level architectural exploration and performance analysis for hardware-software co-design. Similarly, STepNP [63] is an exploratory network processor simulation environment for exploring applications, multiprocessor network processing architectures, and SoC tools. It is a system-level exploration platform based on SystemC and supports model interactions, instrumentation and analysis of new processors, coprocessors and interconnects. In [64], STepNP has been extended to provide a system-level exploration platform for network applications based on configurable processors.

Ptolemy [65, 66] is a system-level design framework that allows simulation and prototyping of heterogeneous multiprocessor SoC systems. It has well established computational models to express both architectures and ap-

plications at multiple levels of abstraction. Similarly, the Metropolis [67, 68] framework embodies the platform based design methodology integrating modeling, simulation, synthesis and verification tools within a single framework. In the platform-based design methodology, a common platform is specified and shared across multiple applications in a given application domain [69]. It also emphasizes in systematic reuse of Intellectual Property (IP) core in order to reduce development risks, costs and time-to-market.

MILAN [14] is a model based integrated simulation framework for embedded system design and optimization, which employs a model based solution for hardware-software co-design and co-simulation. It integrates various simulation models at different levels of abstraction. The designers formally model the target application, underlying hardware, and the system constraints through the interfaces provided by the framework.

The use of UML in embedded system modeling has also been an active research area. Modeling frameworks that use UML notations for modeling reconfigurable architectures can be found in the literature. In such design frameworks, the design process can begin from use-case models that are gradually refined via set of models towards the implementation. The framework DIPLODOCUS [7, 70] is UML [71] based environment that allows fast simulation of SoC at a high level of abstraction. With DIPLODOCUS, applications are modeled using either a UML class or activity diagrams and architectures/mappings are modeled using UML deployment diagrams. Similarly, in [72], the authors proposes a methodology for system modeling based on a specific UML profile. In the context of their work, a high design abstraction level for modeling and analyzing hardware resource sharing has been defined. Additionally, a systemC based simulator to simulate modeled systems and evaluate their performance has been developed. Their methodology is also based on the DIPLODOCUS framework.

The system-level performance analysis and design space exploration methodology called SPADE [13] provides a means to quickly build models of architectures and applications at an abstract level. The application model consists of Kahn Process Networks (KPN) [28], which is discussed in more detail in Section 3.1. The application model describes the functional behavior of an application. The architecture model defines the architecture resources, and it captures their timing characteristics. Application models can be explicitly mapped onto the architecture models and the performance of the resulting system can be analyzed via simulation. It is based on the Y-chart principles,

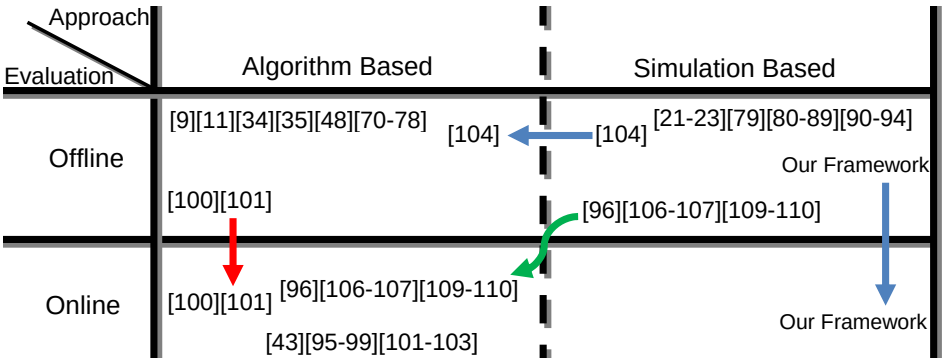


Figure 2.8: General classification of DSE methodologies at the system- and micro-architecture levels. This classification is summarized from the survey presented in [75]

where application models and architecture models are separated via an explicit mapping step. It distinguishes among application specification and architecture characterization, and it uses a trace-driven simulation technique [73] for co-simulation of application and architecture models. Decoupling application and architecture allows the designers to use a single application model to exercise different hardware-software partitioning and to map it onto a range of architecture platforms.

Artemis [12] is also a system-level modeling and simulation environment related to SPADE. Artemis aims at efficiently exploring the design space of heterogeneous embedded systems architectures at multiple abstraction level. Sesame [2,6] is developed within the context of the Artemis project for the efficient system-level performance evaluation and architecture exploration of heterogeneous embedded systems targeting the multimedia application domain. Sesame also uses KPN for modeling application characteristics.

Similarly, the framework defined in [74] is a system-level design framework, and it follows the Y-chart design scheme. In this context, the mapping of applications onto architectures is performed on graph based descriptions and based on the information provided by the designer. In this case, the mapping methodology uses abstract information, such as cycle counts.

2.3 DSE Approaches for Reconfigurable Architectures

In general, DSE methodologies can loosely be categorized based on two criteria: i) methods used to evaluate a single design candidate, and ii) methods used to traverse a design space in order to evaluate more design points [75]. Methods for evaluating a single design candidate can range from purely analytical methods to system-level simulation, RTL simulation and cycle-accurate simulation. Correspondingly, traversing of a design space can also be done in various ways. A trivial method to traverse the design space in order to find the "optimal" solution is to exhaustively evaluate all the design points. For this, various levels of simulation tools can be used, such as system-level simulation, instruction-set simulation and cycle accurate simulation. The design space for a system-level exploration can quickly become large if such exhaustive exploration mechanisms are employed and the tracing of the design space becomes inefficient. Therefore, this method is prohibitive for large design spaces. Other method for traversing the design space includes optimization heuristics, which can be used to iteratively evaluate several design points. Common examples of such methods are genetic algorithms, simulated annealing, ant colony optimization and tabu search (see Section 2.3.1). This approach of traversing the whole design space is usually more effective in terms of time. The general classification of DSE methodologies at the system- and micro-architecture level of computer architecture domain is presented in Figure 2.8. This classification is summarized from the survey presented in [75], which provides an excellent survey for evaluating and covering the design space during early design development.

Similar classification of DSE methodologies can also be identified for the reconfigurable system domains. The state-of-the-art shows a variety of tools and methodologies used for carrying out DSE for dynamically reconfigurable systems. These methodologies can also be categorized based on the aforementioned criteria as shown in Figure 2.8. Here, we focus on two main approaches of performing DSE for dynamically reconfigurable architecture: a) *simulation-based* approaches and b) *algorithm-based* approaches. Note that, these two approaches are not mutually exclusive. This means that any algorithm can also be used together with the simulation-based approach to perform a certain kind of analysis. Furthermore, both approaches can be used to evaluate a single design candidate, or they can also be used to traverse the whole design space. However, there is a clear separation between how these approaches can be used to traverse a design space, such as online and offline. A simulation framework

can traverse certain execution paths in the design space. Typically, the simulation allows an exhaustive traversal of one of more design points in the design space. When combined the simulation approach with certain algorithms, it can sample the design space in a specified way. For instance, when the genetic algorithm is applied together with the simulation, the design space can be randomly traversed. In the rest of this section, these approaches are elaborated in more detail.

2.3.1 Algorithm-Based Approaches

As it can be inferred from Figure 2.9, in an algorithmic approach, algorithms of different complexities are used to evaluate a set of criteria for a set of alternative designs. These algorithmic approaches can obtain the required results quickly for a one-time use. However, they are difficult to recreate and reproduce in case of comparison and/or re-evaluation. For instance, if the application or the architecture changes, the algorithm has to be re-implemented. A large number of different algorithms can be found in literature for DSE to carry out hardware-software partitioning, mapping and scheduling of reconfigurable architecture. For example, dynamic programming [76] branch and bound [77], Integer Linear Programming (ILP) [10], simulated annealing [52, 78], tabu search [79], genetic algorithm [80] and ant colony optimization [81]. Many of these algorithms are used to explore the design space of reconfigurable systems, and they are widely accepted for carrying out automated DSE for reconfigurable systems.

Typically, an algorithm-based approach can either be used to evaluate a single design point, or it can also be used to traverse the whole design space. The exhaustive search algorithms for DSE evaluate every possible combination of design points, and as a result, they are less effective in case of large design spaces. The technique presented in [10] introduces such an exhaustive exploration method based on ILP for resolving temporal partitioning problems. Similarly, the authors in [82] proposed a greedy search technique of DSE for partitioning and scheduling. Chatha et al. [38, 83] utilize an iterative algorithm for DSE based on list scheduling and the methodology presented in [42] employs a maximum flow formulation for DSE solution for spatial partitioning. These algorithms are often implemented to optimize or to evaluate new design alternatives in order to reach the near-optimal design point.

Algorithms, such as the simulated annealing [52, 78], the genetic algo-

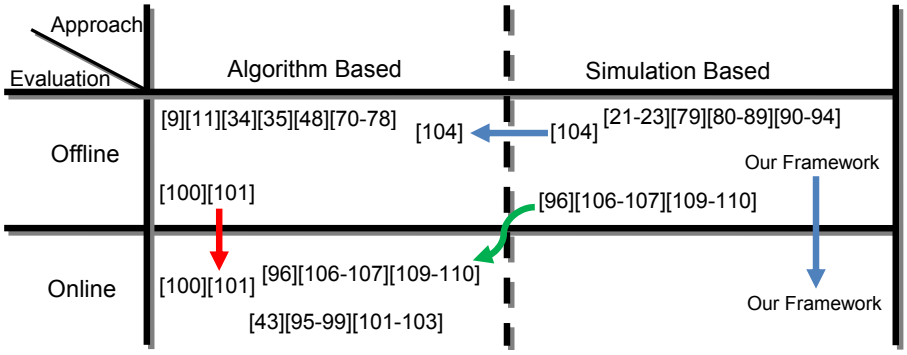


Figure 2.9: Classification of DSE Methodologies for reconfigurable architectures. These methodologies can be classified based on the approach used to perform DSE and the time of the evaluation.

rithm [39] and the tabu search [79], randomly sample the design space in order to traverse the total design space. On the other hand, the ant colony optimization algorithm [81] does not randomly sample the design space, rather incorporates some knowledge of the design space to analyze the design point. Wang et al. [81] present an approach based on ant colony optimization, in which collection of agents co-operate using distributed and local information to effectively explore the search space. The authors in [80] perform DSE based on multi-objective evolutionary algorithms, which introduce performance improvements for genetic algorithms. The technique in [8] also describes the automated DSE methodology for area timing tradeoffs utilizing evolutionary multi-objective algorithms. Many DSE techniques also employ the combination of one or more of such algorithms. A combination of the tabu search and the list scheduling is presented in [84], a combination of the genetic algorithm and the list scheduling algorithm is presented in [80], and a hybrid strategy for DSE based on the genetic algorithm and the tabu search is presented in [79].

2.3.2 Simulation-Based Approaches

In the simulation-based approaches, a design candidate is evaluated with a defined set of stimuli. Typically, in such an approach, a standardized simulation framework is used to model, simulate and explore reconfigurable systems' behavior at various design levels. Such a simulation framework offers modeling methodologies and simulation tools to evaluate different design criteria. The simulation-based approach can also be used to evaluate a single design point,

as well as, to exhaustively traverse the given design space. Advantages of simulation-based approaches include higher model (component) re-usability and an easy customization of the design. These models are easier to construct and validate. Therefore, they allow quick design and validation of the system. The literature shows a variety of simulation frameworks for reconfigurable architectures developed by various research groups (See Figure 2.9). Typically, these frameworks include system-level simulation, RTL simulation and cycle-accurate level simulation. In this research, we focus only on the system-level simulation tools. In the following, a number of such system-level simulation frameworks are summarized.

The simulation framework presented in [85, 86], allows a system-level cycle accurate performance evaluation of hybrid reconfigurable processors, at low architectural level. Within the context of the presented framework, the processor behavior is modeled by an extension of the SimpleScalar simulator [87] and the reconfigurable unit is implemented in VHDL. Another system-level modeling and co-simulation environment for reconfigurable architecture is presented in [88]. This co-simulation approach is designed for power performance tradeoffs in task scheduling, and it is based on clock-gating and frequency-scaling. The approach used in this framework is based on an object-oriented system, where discrete event classes and objects are used to model all the necessary behaviors of reconfigurable systems. Likewise, the modeling approach presented in [89] exploits a graph based approach for modeling a hardware-software partitioning problem for architectures including reconfigurable hardware. The application is modeled as a process graph, and the architecture is modeled as an architecture graph. Each instance of a partition consists of a process graph, an architecture graph and an explicit communication channel.

The ADRIATIC project [90] proposes a SystemC based design flow methodology for DSE, simulation and synthesis of reconfigurable systems. With the proposed methodology, reconfigurable components are modeled as classes, and these classes are transformed into modules, which provide the implemented functionality. These modules are called Dynamically ReConfigurable Fabric (DRCF), and they may be created from multiple classes. Similarly, the DSE scheme of [91, 92] presents a system-level performance estimation approach for SoC featuring reconfigurable logic. This approach is related to the methodology presented in the ADRAITIC project. In this context, the reconfigurable components are used as hardware accelerators.

The SyCERS project [26] is also a SystemC design exploration framework for SoC reconfigurable architecture. The framework can be used by system designers to study and verify reconfigurable system specifications. It is built on top of the SystemC library, and it allows the specification of both architecture, and system models. Architecture models are implemented as a class, which are derived from a set of common interfaces provided in the framework. The system models use the interfaces, which are implemented in the architecture model, in order to access the hardware resources. As a result, a model can be tested in any architecture implementing the same interface set.

Rissa et al. [25, 93] have presented a system-level approach based on OCAPI-*xl* [58] for modeling and implementing hardware-software systems that contain runtime reconfigurable systems. OCAPI-*xl* is a modeling language similar to SystemC. Similarly, Tiensyrjä et al. [94] present a system-level design methodology for reconfigurable SoC based on SystemC and OCAPI-*xl*. The work presented in [95] also uses SystemC based co-simulation scenarios to model reconfigurable cores in multiple-context representation of the different functionalities.

Perfecto [24, 96] is another SystemC based design space exploration framework for dynamically reconfigurable architectures. It allows a designer to perform rapid exploration of reconfigurable design alternatives, and to detect system performance bottlenecks. Given an architecture model and an application model, Perfecto uses SystemC transaction level models to automatically simulate the system design alternatives. Numerous hardware-software partitioning, scheduling and placement algorithms, can be embedded into the framework for the system analysis.

The objective of the OverRSoC project [97] is to provide a complete design framework for exploration and validation of embedded real time operating systems for reconfigurable SoC. OverRSoC is a system-level modeling framework for the design of a complete reconfigurable SoC platform including processor(s), dynamically reconfigurable architecture and operating system services. The method provided in the project is based on abstract and modular SystemC models that allow to explore, simulate and validate the distribution of operating system services on reconfigurable platforms.

Tseng et al. [98] and Chun-Hsian et al. [99] proposed a UML-based hardware-software co-design platform for dynamically reconfigurable computing systems targeting network security systems. The proposed design flow

takes a UML-based application model and facilitates the co-synthesis and rapid prototyping of dynamically reconfigurable computing systems. Similarly, Kangas et al. [100] describes a complete design flow based on UML for multiprocessor SoC covering the design phases from system-level modeling to FPGA prototyping.

We can categorize DSE approaches for reconfigurable architectures also based on the time of the evaluation: *offline* evaluation and *online* evaluation. This categorization is well displayed in Figure 2.9.

2.3.3 Offline Evaluation

Offline evaluation refers to a condition where a design candidate is evaluated for fixed system constraints. This evaluation is often performed at design-time. No changes in the system (application, architecture and/or environment) are given as feedback to the evaluation process. Therefore, the evaluation simply returns its results without such dynamic consideration. As a result, such methods are not appropriate for dynamic system conditions. Offline evaluation can generally be faster, but it may be less accurate as the runtime behavior of a system is mostly captured by offline (static) estimations and predictions. Examples of such evaluation can be found in [39–42, 83].

2.3.4 Online Evaluation

Online evaluation refers to the runtime evaluation of the design candidates where a design candidate is evaluated for (dynamically) varying system constraints. Any changes in the system (application, architecture and/or environment) are given as feedback to the evaluation process. As a consequence, the design parameters are adjusted during the evaluation based on the changes encountered by the system. The runtime evaluation of a design candidate can provide better accuracy as it takes into account some feedback from the system. Nevertheless, such evaluation is typically hard, due to the enormous size of the search space generated by the runtime system parameters.

Vahid et al. [101] present a simple approach for online evaluation of the task mapping in which a mapping module evaluates the most frequently executed tasks at runtime and maps them onto a reconfigurable hardware com-

ponent. This work [101], however, has a focus on the lower level and targets only loop kernels. A similar approach for high-level runtime evaluation of application mapping is presented in [102] for multiprocessor SoC containing fine-grain reconfigurable hardware tiles. This approach details a generic runtime resource assignment heuristic that performs fast and efficient task assignment. In [103], the authors define the dynamic coprocessor management problem for processors with FPGA and provide a mapping to an online optimization based on cumulative benefit heuristics, which is inspired by a commonly used accumulation approach in online algorithm work. In the same way, Compton et al. [104] present runtime resource allocation and scheduling heuristic for a multi-threaded environment based on the status of the reconfigurable system. Correspondingly, Ghaffari et al. [47] presents a dynamic and online DSE method for task mapping, task scheduling and task allocation for reconfigurable architectures. The proposed method consists of dynamically adapting the architecture to the processing requirements. The authors in [105] also present a runtime optimization targeting the speedup of applications running on a reconfigurable platform. In this context, an online adaptive decision algorithm has been proposed. The algorithm is used to determine whether a task should be executed as a hardware task or a software task.

Likewise, authors in [106, 107] present online resource management for heterogeneous multi-processor SoC systems, and the authors in [108] also presents a runtime mapping of applications to a heterogeneous reconfigurable tiled SoC architecture. The approach of [108] consists of a mapper, which determines a mapping of application(s) to an architecture, using a library at runtime. The approach presented by authors in [109] performs mapping of streaming applications, with real-time requirements, onto a reconfigurable MPSoC architecture. In the same way, Faruque et al. [110] present a scheme for runtime agent based distributed application mapping for on-chip communication for adaptive NoC based heterogeneous multi-processor systems.

The classification presented in Figure 2.9 shows that the trend of most current DSE research efforts for reconfigurable architectures focus either on the second row or the second column of the table. Note that, the mentioned categories in the classification are not strictly orthogonal to each other, and there is no absolute separation between the cells, in the row and column. As a matter of fact, few DSE efforts overlap each other (see the arrows in Figure 2.9). The DSE methodology presented in [111] combines an algorithmic approach with a simulation approach for the algorithm and architecture explorations. They use an estimation based algorithm for DSE for reconfigurable

architectures [112] together with a simulation modeling framework. In a similar way, the approaches in [102, 113, 114] combine a design time exploration together with a runtime management to tradeoff faster exploration with accuracy. The approach of [114] proposes a customized runtime management to map an application onto the platform. This customized runtime management is a pareto-based approach combining the design time application mapping and platform exploration with a low complexity runtime manager. Similarly, the approach described in [3, 115] provides a DSE framework for enabling and supporting runtime resource management for MPSoC. The framework combines the design-time and runtime methodology to provide an "optimal" tradeoff in terms of various parameters, such as power and performance.

The categorization, however, pointed out that there is still a huge gap for tools and methodologies for online evaluation of reconfigurable architectures, which fit the last cell in the classification diagram (see Figure 2.9). To the best of our knowledge, there is no existing framework for DSE which allows designers to model and evaluate the reconfigurable systems' behavior at runtime. Due to this lack, various research groups rely on their custom-built proprietary models/simulators for evaluating architectures and algorithms. As a result, the evaluation procedure is complex, and the comparison between various evaluations is difficult if not impossible. Therefore, it is crucial to have a common modeling and simulation framework that can be re-used between research groups or even industry. This can provide a standard platform, which can allow easy comparison between various evaluations, and hence, it can also be used as a reference tool for future research. This can add a tremendous value in the area of reconfigurable systems by providing an excellent vehicle for research. Furthermore, we believe, providing such a standardized framework can also provide an invaluable insight to future generations of (partially) dynamically reconfigurable systems from the industrial perspective. A parallel example can be found in the field of micro-architecture with the revolution brought by the SimpleScalar [87] in the academic, as well as the industrial research practice. Towards this end, in order to fill this gap, the focus of this dissertation is to provide a generic system-level modeling and simulation framework, which can explore and evaluate reconfigurable systems both statically and at runtime.

2.4 Conclusions

This chapter served as a simple introduction to the concepts associated with the reconfigurable architecture and the system-level DSE. In this chapter, we discussed the heterogeneous reconfigurable systems, and their challenges in the context of the embedded system domain. Several challenges encountered while designing such systems are hardware-software partitioning (e.g. spatial and temporal partitioning), task allocation, task placement and task scheduling. Moreover, the chapter also presented a description of the DSE and its importance at the early design stage. Early stage DSE allows to investigate and analyze tradeoffs between all possible design goals, and to select the most appropriate solution at early design stages. As a result, performing such exploration enables decisions to be made quickly, and it can significantly reduce the overall design time. Furthermore, the chapter also presented the study of system-level design methodology. System-level design methodology uses abstracted models to capture the application, the architecture and the system behavior. As a result, it allows easy and rapid exploration of the various design alternatives at early design stages. There are several design frameworks supporting system-level design in the context of embedded system domain, which have also been reviewed in the chapter. Additionally, we presented a classification of DSE tools and methodologies for reconfigurable architectures. These tools are classified based on the approach used for performing DSE (such as algorithm-based approach and simulation-based approach) and the stages they are used for carrying out DSE (such as offline and online evaluation).

In the next chapter, we propose a two-level exploration mechanism for reconfigurable systems, in order to address the various challenges that are encountered while designing such systems. The proposed mechanism allows exploration and evaluation of reconfigurable system, both statically and at runtime.

Note. Some contents of this chapter is based on the following article:

K. Sigdel, M. Thompson, C. Galuzzi, A.D. Pimentel, K.L.M. Bertels, **rSesame - A Generic System-Level Runtime Simulation Framework for Reconfigurable Architectures**, *Proceedings of the International Conference on Field-Programmable Technology (FPT'09)*, Sydney, Australia, December 2009, pp. 460-464.

3

Runtime Mapping Exploration

Dynamically reconfigurable systems can evolve under various conditions due to changes imposed either by the architecture, or by the applications, or by the environment. The architectural behavior can change due to various reasons - e.g. processing elements shut-down in order to save power or extra processing elements are added in order to meet the real time constraints. Similarly, the application behavior changes due to the dynamic nature of the application - e.g. the application load can change due to the arrival of sporadic tasks. In such systems, the design process becomes more sophisticated as all the design decisions have to be optimized in terms of runtime behaviors and values. Runtime mapping exploration allows to explore reconfigurable systems at runtime to optimize application-to-architecture mappings[1] in order to adapt to the changing behavior of the application(s), the architecture or the environment. Performing such runtime explorations, the system efficiency can be increased in terms of various system attributes, such as performance, chip area, memory requirement, and power consumption.

This chapter presents an overview of the runtime mapping exploration of reconfigurable architectures and various issues involved with it. The chapter is organized as follows. Section 3.1 summarizes the Kahn process network, which is used as the model of computation for representing applications in this research. Section 3.2 presents the problem definition together with the tool flow for the runtime mapping exploration. Section 3.3 and Section 3.4 discuss the static and the runtime application mapping respectively. Section 3.5 proposes the two-level mapping exploration approach for reconfigurable architectures, while section 3.6 presents an overview of the runtime mapping

[1] Application-to-architecture mapping is sometimes referred simply as mapping in this dissertations.

manager. Finally, Section 4.5 concludes the chapter.

3.1 Kahn Process Networks

The main focus of this research is to develop a generic system-level framework for performing DSE of dynamically reconfigurable architectures, targeting streaming applications from the multimedia domain (e.g. JPEG, MJPEG and MPEG codecs). The main characteristic of these applications is that they are data-flow oriented applications, i.e. large streams of data have to be processed. A Kahn Process Network (KPN) [28] has simple operational semantics, which can conveniently specify *stream-oriented* data processing behavior of such multimedia applications. As a result, they are suitable to model the parallel and streaming nature of these applications. Therefore, in this research, we specify applications as graphs at the granularity of task or function level KPNs. In the following, we describe KPN in more detailed.

The KPN model of computation consists of a network of concurrent processes. These processes run autonomously and communicate with each other over unbounded FIFO channels, in a point-to-point fashion, using a blocking-read synchronization module. In the KPN, each of these processes consists of a sequential program that executes concurrently with other processes. When these processes are mapped onto hardware, such as an FPGA, they are synchronized via a blocking read and non-blocking write synchronization protocol. The KPN characteristics can be summarized as follows:

- The KPN network is *deterministic* in nature, which implies that for any input there exists the same output. In other words, the output does not depend on the process execution order. As a matter of fact, it produces same output for the same set of input, irrespective of the schedule chosen to evaluate the process networks. This makes KPN easier to be mapped onto different processors in the architecture.

- The KPN models have a *distributed control* without a global scheduler, and as a result, a partitioning of a KPN over multiple processes is a simple task. This makes KPNs ideal for mapping onto concurrently available resources, such as FPGAs. Furthermore, with KPN models, the exchange of data is distributed over the FIFO channels without having a global memory structure, and hence resource contention does not oc-

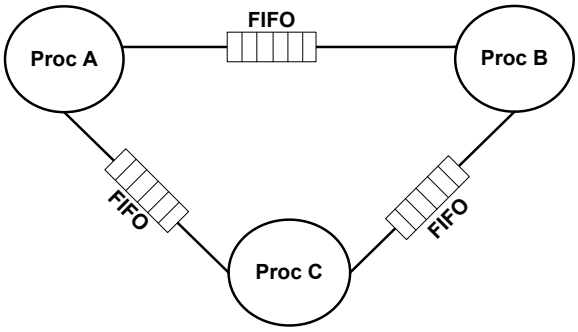


Figure 3.1: An example of Kahn Processor Network (KPN). Proc A, Proc B and Proc C are autonomous processes that communicate with each other using FIFO channels.

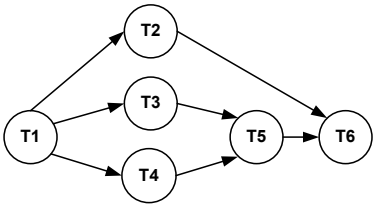
cur. However, this highly depends on the memory/buffer structure in the underlying architecture model, and how FIFOs are mapped onto this structure.

- The processes in a KPN are *self-schedulable*. The inter-processor synchronization is done by blocking reads and non-blocking writes. This synchronization mechanism can be easily realized in both hardware and software.

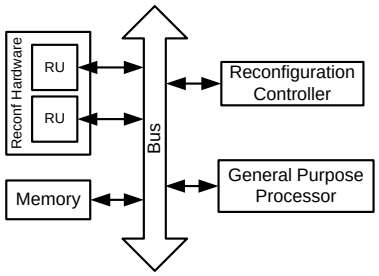
Figure 3.1 depicts an example of a KPN, where Proc A, Proc B, Proc C are autonomous processes that communicate with each other via FIFO channels. KPN models are widely used for modeling the parallel nature of streaming applications as presented in [12, 13, 21]. In [116], KPNs are extended with the notion of time behavior. In this case, they are used to describe packet processing workloads, and to enable the time dependent behavior. These models can be manually derived, or they can also be automatically converted from sequential C/C++. The automatic generation of KPN models from sequential code has been addressed in [117], [118].

3.2 The System Model

Let us consider an example of a KPN as depicted in Figure 3.2(a). Additionally, let us consider an example of a dynamically reconfigurable architecture as shown in Figure 3.2(b). The KPN consists of six different tasks having different requirements, and they are suitable for mapping onto different architectural



(a) A KPN that consist of six different tasks having different requirements and suitable for mapping onto different architectural components.



(b) A dynamically reconfigurable architecture that consists of a GPP, a reconfigurable hardware, a memory and a reconfiguration controller, all connected to a peripheral bus.

Figure 3.2: An example of a KPN and a dynamically reconfigurable architecture considered in the system model.

components. The architecture consists of a GPP, a reconfigurable hardware, a memory, and a reconfiguration controller, all connected to a peripheral bus. The reconfigurable hardware consists of one or more Reconfigurable Units (RUs). The application to be run on the reconfigurable hardware is loaded onto RUs. The reconfiguration controller is responsible for loading and configuring the application onto the RUs at runtime. Finally, let us consider an execution of the application given in Figure 3.2(a) on a dynamically reconfigurable architecture as the one depicted in Figure 3.2(b). In the following, we discuss the mapping in more detail.

There are three different types of tasks to be specified in a system: *software* tasks, *hardware* tasks and *pageable* tasks:

- *software* tasks (SW Tasks) are the tasks always executed as software on the GPP;
- *hardware* tasks (HW Tasks) are the tasks always executed as hardware on the reconfigurable hardware;
- *pageable* tasks (pageable Tasks) are the tasks that can switch between the GPP and the reconfigurable hardware for their execution.

Given an acyclic KPN and different types of tasks, the primary goal of task partitioning is to assign each task to a given task set (HW, SW or pageable).

From the KPN given in Figure 3.2(a), let us consider, as an example, the following task sets:

$$\begin{aligned}&\text{SW Tasks} = (T_1)_{SW}\\ &\text{HW Tasks} = (T_4, T_5, T_6)_{HW}\\ &\text{pageable Tasks} = (T_2, T_3)_{\text{pageable}}\end{aligned}$$

This is called *spatial partitioning*. The spatial partitioning identifies a set of tasks that belong to a particular task type. Given the partitioning, task mapping binds each task to a resource either the GPP or the reconfigurable hardware. The mapping of HW and SW tasks is already known, and therefore, they are simply be coupled with their corresponding resources. The HW tasks are coupled with the Reconfigurable Hardware (RH), and the SW tasks are coupled with the GPP. The mapping of pageable task is, however, not known beforehand. Therefore, pageable tasks are coupled with any of the available resources depending on the system conditions. In the mapping phase, particularly, the pageable task set attains a resource binding.

For the previously mentioned partitioning, we can find, for example, the following mappings.

$$\begin{aligned}&\text{Mapping1} : (T_1, T_2)_{GPP}, (T_3, T_4, T_5, T_6)_{RH}\\ &\text{Mapping2} : (T_1)_{GPP}, (T_2, T_3, T_4, T_5, T_6)_{RH}\end{aligned}$$

The example shows that in the mapping phase, tasks T_2 and T_3 from the pageable task set attain their resource bindings with the GPP and the RH respectively in Mapping1. In Mapping2, both tasks (T_2 and T_3) attain their resource binding with the reconfigurable hardware only.

Reconfigurable hardware is limited by its physical hardware size. As a result, not all the tasks mapped onto the hardware can be executed at the same time. Depending on the size of the reconfigurable fabric, these tasks have to be divided into different mutually exclusive hardware *configurations*. Each configuration can therefore be sequentially executed onto the hardware at each configuration time. This process is called *temporal partitioning*.

In the example, let us assume a limitation of two tasks that can be executed on

the given reconfigurable hardware at once. In this case, the execution of tasks T_2, T_4, T_5 and T_6 in Mapping1 and the tasks T_2, T_3, T_4, T_5 and T_6 in Mapping2 are not possible at once. These tasks therefore have to be divided into different hardware configurations. For example, we can consider the following configurations.

Mapping1: $(T_1 , T_2)_{GPP}$, $((T_3 , T_4)_{C_1} , (T_5 , T_6)_{C_2})_{RH}$;

Mapping2: $(T_1)_{GPP}$, $((T_2, T_3)_{C_1} , (T_4, T_5)_{C_2} , (T_6)_{C_3})_{RH}$

where C_1, C_2 and C_3 are different hardware configurations.

In Mapping1, since tasks T_2 and T_4 can fit on the reconfigurable hardware at the same time, these tasks can run in parallel as there is no task dependency between them according to Figure 3.2(a). On the other hand, though tasks T_5 and T_6 are in the same configurations, these two tasks cannot run at the same time due their dependency. Under such circumstance, task T_6 can be configured and made ready for execution on the hardware, while task T_5 is still executing. In this way, the configuration time of one task can be overlapped with the execution time of the other task. In similar ways, the configuration hiding is possible also for the tasks T_4 and T_5 in Mapping2. This mechanism is also discussed with an example in Section 3.3.

Definition 1: Given a task set T: $\{ T_1, T_2, ..., T_N \}$ with $N > 0$ number of tasks in a KPN, the mapping can be defined as M = { HW, SW }, where

$HW, SW \in T$ and
$SW \cup HW = T$
$SW \cap HW = \phi$

Definition 2: Given a task set T: $\{ T_1, T_2, ..., T_N \}$ with $N > 0$ number of tasks in a KPN, the goal of the runtime mapping exploration is to find a set of SW and HW, such that M is "optimal" according to the given criteria.

These criteria can be defined based on the various design objectives and the design constraints imposed on the system. The designers can specify different kinds of design constraints, such as area, communication bandwidth and power consumption. Additionally, they can also indicate different design objectives, such as the maximization of the system performance, the minimization of the communication bandwidth and the minimization of the power consumption. The runtime mapping exploration takes these objectives and constraints into

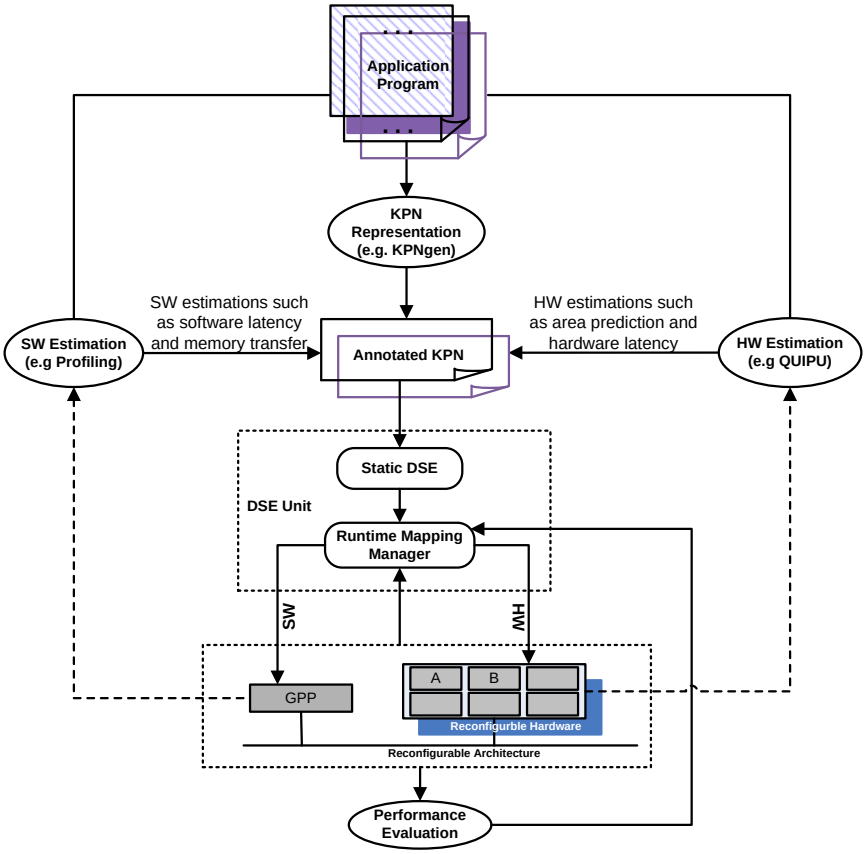


Figure 3.3: The conceptual framework for the runtime mapping exploration. The KPNs are annotated with HW and SW estimates and passed to the Design Space Exploration (DSE) unit. The DSE unit performs mapping exploration and identifies various task mappings for the given reconfigurable architecture.

consideration, and it tries to find the sets of HW and SW tasks at runtime, such that the mapping identified is "optimal" in some predefined ways.

3.2.1 Runtime Mapping Exploration Framework

The conceptual framework for the runtime mapping exploration is shown in Figure 3.3. In the first phase, the application program is transformed into a KPN model. This transformation can be performed either manually or automatically from sequential C/C++ code by using an automatic KPN generation

tool, such as the one presented in [117].

As a preprocessing phase, the application is profiled to gather various software estimates. Well established methods and tools (software profilers such as *gprof* [119]) can be used for analyzing the application, statically and/or dynamically, in order to determine relevant information, such as the execution time, the memory size, and the number of times a task is executed. For these tools, the worst-case behavior is a reasonable assumption for the system evaluation. As a result, the quantitative measure obtained through such analysis is, typically, a worst-case estimate. The profiling of the application provides various software estimates for a task when it is executed on a microprocessor. We are mainly interested in the following software estimates:

- **the software latency**, which is the quantitative measure of the total execution time for a task during its execution as software on the microprocessor. The software latency for a task T is denoted as T_{SW};

- **the communication bandwidth**, which is the quantitative measure of the total number of bytes exchanged (written or read) through a FIFO channel between two processes in a given KPN. It is computed as the product of the total number of tokens sent through a FIFO channel and the size (in bytes) of each token sent through that particular channel. The communication bandwidth between tasks T_i and T_j over a FIFO channel k is given as:

$$CL_{ij} = \sum_{k=1}^{N} n_{ijk} \cdot m_k, \qquad i, j = 1 \ldots N$$

  where n_{ijk} is the number of token sent through channel k between tasks T_i and T_j, m_k is the token size of channel k and N is the total number of FIFO channels between T_i and T_j.

Similarly, hardware cost prediction models can be used to determine the hardware attributes of a task. A hardware cost prediction model (e.g. the Quipu model [120]) can predict different hardware attributes, such as hardware area, interconnect delays and hardware latency for a task when it is executed on a given reconfigurable hardware. We are interested in the following hardware estimates:

- **the hardware latency**, which is the quantitative measure of the total

execution time for a task executed onto the reconfigurable hardware. The hardware execution time for a task T is denoted as T_{HW}.

- **the area occupancy**, which is the quantitative measure of the area occupied by a task on a given reconfigurable hardware. Area can be measured in slices or it can also be expressed in terms of percentage of the total hardware area. The area prediction for a task T is denoted by A_T.

Each task in a KPN is annotated with software and hardware attributes resulting in an *annotated* KPN. As shown in Figure 3.3, the *annotated* KPN is then passed to the DSE phase, where several exploration strategies can be applied. The DSE unit collects a variety of architectural information, such as the free resources and timing information. Based on the architecture information, the application information, and using the DSE strategy implemented, the DSE unit generates several mappings for the underlying architecture. In this case, DSE is performed in two phases.

In the first phase, a static exploration is performed to compute an initial set of mappings. In this case, only the static condition of the system is considered. As a result, the application and the architecture constraints are pre-determined and fixed. Each mapping consists of a HW and a SW task sets (see Definition 1). Out of these task sets, the designers can specify few tasks to be pageable. In the second phase, in order to optimize the mapping, these pageable tasks are optimized. Note that, designers can guide this decision using their own knowledge about the application. For instance, designers can explicitly decide, for certain tasks, to be either *software* or *hardware* or *pageable*.

In the second phase of DSE, the mappings from the first phase are optimized to accommodate the runtime condition of the system. This implies that, if there is any change in the architecture or the application, then this is also given as a feedback to the decision unit, which adjusts the mapping accordingly. As depicted in Figure 3.3, the Runtime Mapping Manager (RMM) is responsible for performing such optimization. If any change in the architecture and/or the application is reported, pageable tasks may be relocated from the GPP to the reconfigurable hardware and vice versa, resulting into a different set of mappings. The performance evaluation unit monitors the performance impact of each mapping for a particular system function for the given reconfigurable resources. Furthermore, the evaluation results from the performance evaluation unit can also be used as a feedback to the DSE unit. Such information can assist the RMM for further decision making while identifying various

other mappings. At the end of the exploration, an "optimal" set of mapping is obtained, which satisfy the given system condition.

3.3 Static Application Mapping

The traditional application mapping approaches for reconfigurable architectures and embedded systems, in general, use static methodologies for mapping an application onto an architecture [39, 42, 77]. These classical mapping approaches typically rely on offline information of an application and/or architecture as primary information for application mapping. Such application and/or architecture information are recorded under static system conditions using various techniques such as profiling. *Static conditions, here, refer to the condition of the system where the application, the architecture and the environment do not change.* Under static conditions, the application behavior is steady, the architecture requirement is fixed, and system constraints are predetermined. Based on static information, various heuristics for application mapping are applied to execute an application onto the architecture. As a result, a static set of HW tasks and SW tasks are identified for the architecture mapping. In the following, an example showing a static mapping behavior for an application is discussed.

Let us consider the KPN shown in Figure 3.2(a). Let us assume that while using the mapping heuristic to execute parallel tasks onto the reconfigurable hardware, the static mapping approach identifies the two sets of tasks: SW = { T_1, T_5, T_6 } and HW = { T_2, T_3, T_4 }. The HW task set is then mapped onto the reconfigurable hardware and the SW task set is mapped onto the GPP. This mapping is determined statically, thus, each task is mapped onto only one resource during the entire execution of the application. The mapping behavior for this example is shown in Figure 3.4(a). As it can be seen from the figure, the mapping of tasks T_2, T_3 and T_4 onto the reconfigurable hardware, and the mapping of tasks T_1, T_5 and T_6 onto the GPP, are fixed during their entire execution (see period I, II and III in the figure). In this example, we assume that the given reconfigurable hardware can fit only two tasks at the same time. As a result, although task T_4 can run in parallel with tasks T_2 and T_3, it can be executed only after the execution of these tasks.

The mapping depicted in Figure 3.4(a) also takes the reconfiguration overhead into account. Each hardware task is configured before its execution. The

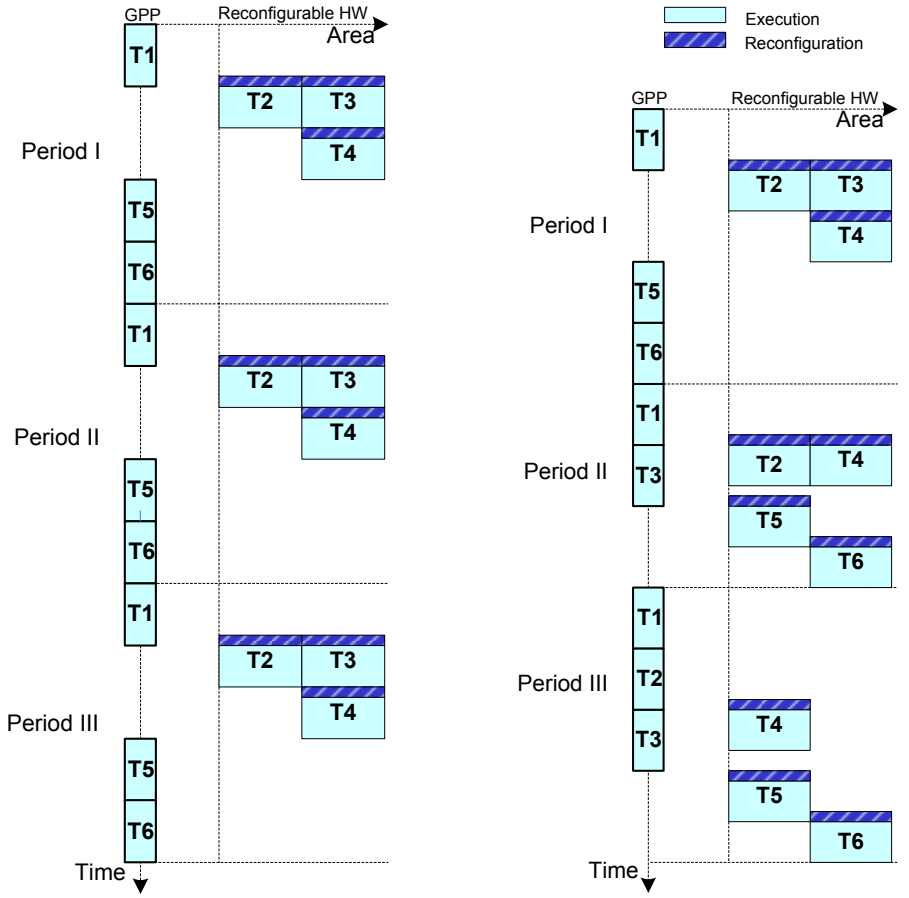
reconfiguration time for each task in Figure 3.4(a) is shown with shaded lines. In this example, we assume that all the tasks have the same hardware, software and reconfiguration latency. Additionally, the example shows that the configuration of task T_2 is overlapped with the execution of task T_1 in order to gain performance. We call this as *reconfiguration hiding*. Performance can be significantly improved by hiding reconfigurations. Note that, reconfiguration hiding is possible only in case of partially reconfigurable hardware, which allows part of the hardware to be configured while other parts are executing other tasks.

3.4 Runtime Application Mapping

Dynamic systems can change under various conditions while maintaining both functional consistency and non-functional design attributes, such as power consumption, performance and redundancy. These changes can be imposed and initiated either by architecture, or by an application, or by the environment. For example, an architecture can change under the following conditions: an increase or decrease of the available or connected resources, a requirement to temporarily switch-off one or more parts of the hardware to reduce the power consumption, or a necessity to achieve high fault tolerant behavior for certain tasks. At the same time, the application itself can change to maintain a specified Quality of Service (QoS) for variable processing load of one or multiple applications, or due to the arrival of additional sporadic tasks, or to uphold the load balancing.

Applications with varying execution intensities have dynamic tasks. The processing requirement for such dynamic tasks is not fixed. As a result, when the application requirements for such tasks change, it can be advantageous to change their mapping as well. For instance, if the processing requirement for a task significantly decreases, it may be more profitable to change its execution from the reconfigurable hardware to the microprocessor, such that the hardware can be used to accelerate other tasks. The static mapping methodologies however do not cover any of the circumstances discussed above. *To cope with all these situations, it is necessary to support runtime application mapping.*

Runtime application mapping allows a task to be executed on various resources at different time intervals during the execution of the application without interfering with the execution of the other tasks. As a result, task mapping



(a) Static Application Mapping

(b) Runtime Application Mapping

Figure 3.4: Tasks mapped onto the GPP and the reconfigurable hardware with the static and the runtime application mapping. With the static mapping, the same set of tasks is mapped at different mapping intervals (I, II, III), while with runtime application mapping, different task sets are mapped at different mapping intervals.

can be changed whenever required to adapt to the changes imposed either by the application, or by the architecture, and/or by the environment. In the following, an example showing a runtime mapping behavior for an application is presented.

Let us consider the KPN given in Figure 3.2(a). An example of runtime mapping behavior for the application is shown in Figure 3.4(b). The runtime

mapping approach identifies different sets of tasks at different intervals. For time intervals I to III, different sets of tasks are identified: $SW_I = \{T_1, T_5, T_6\}$, $SW_{II} = \{T_1, T_3\}$, $SW_{III} = \{T_1, T_2, T_3\}$ and $HW_I = \{T_2, T_3, T_4\}$, $HW_{II} = \{T_2, T_4, T_5, T_6\}$, $HW_{III} = \{T_4, T_5, T_6\}$ respectively. As it can be inferred from the figure, these HW and SW tasks are mapped onto the reconfigurable hardware and the GPP respectively at the corresponding execution intervals. As a result, in this example, tasks T_2, T_3, T_5 and T_6 change their mapping during the execution from the GPP to the reconfigurable hardware and vice versa. As in the static mapping case, since the given reconfigurable hardware can fit only two tasks at the same time, although task T_4 can run in parallel with tasks T_2 and T_3, the former can be executed only after the execution of the latter two tasks. Furthermore, with this mapping, the reconfiguration hiding is possible between task T_1 with T_2 and T_3, task T_3 with T_5 and task T_2 with T_4, respectively.

3.4.1 Pageable Tasks

In addition to the classical sets of HW tasks and SW tasks, the runtime mapping recognizes another set of tasks, which is called *pageable* task set. Pageable tasks are those tasks that can change their execution resources during their execution. These tasks can be mapped onto the GPP or the reconfigurable hardware, depending on the runtime status of the system. A pageable task can be executed as a traditional software program on the microprocessor at one point of the execution and it can be executed as a hardware circuit on the reconfigurable hardware at another point, thus providing a solution, which is a perfect blend of flexibility and performance. As discussed in the runtime application mapping example, it can be observed from the example of the runtime application mapping in Figure 3.4(b) that task T_2, T_3, T_5 and T_6 change their mapping from the GPP to the reconfigurable hardware and vice versa. As a result, these tasks are considered as pageable tasks.

Pageable tasks provide great flexibility with their implementation migration at runtime between the GPP and the reconfigurable hardware. Each pageable task must be available in both implemented versions (configuration data for the reconfigurable hardware and software object code for the microprocessor). Whenever decided by the system, a pageable task may be executed by picking its corresponding version and loading it into the target execution environment. Moreover, during execution, a pageable task may be relocated

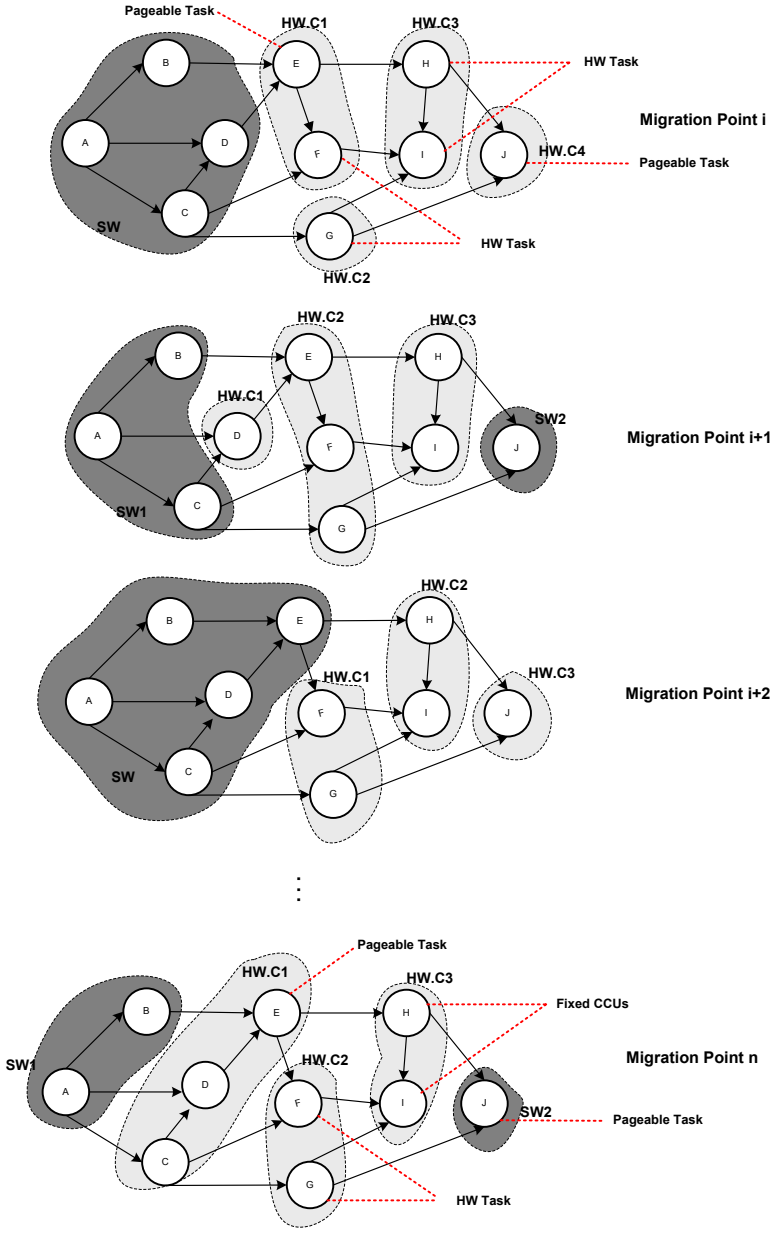


Figure 3.5: An example showing how pageable tasks change their mapping and adapt to the changing system condition by using different resources. Tasks J, D, E, C change their mapping from HW to SW and vice versa at different migration points.

to another execution environment. Such task relocations can be performed in different ways. . For example, in one way of performing such relocation, the task is frozen first, and then its counterpart version is loaded into the target executing environment, the context data is migrated and finally the task resumes its execution in the new execution environment. For instance, when the execution requirement of such tasks decreases, they can be executed on the GPP and when the requirement increases, they can execute on the reconfigurable hardware. In this way, pageable tasks can utilize computation-per-resource cost more efficiently.

By adding the notion of pageable tasks, dynamism is added to the mapping methodology, which allows on demand application mapping onto different resources as required. Nevertheless, pageable tasks also have penalty associated with them. Changing the mapping for a pageable task at runtime involves various issues, such as saving the state of the task, context switching, and address space transfer. All these can incur overhead. Typically, a tradeoff has to be made between the cost and the benefits of changing the mapping for a pageable task.

The degree of dynamism in the mapping can be decided by the number of pageable tasks in the system. By allowing all the tasks to be pageable, the degree of dynamism in the application mapping is maximum. At the same time, it can have the maximum overhead of changing the mapping. As a result, fixing certain tasks as HW tasks and/or SW tasks allows a good tradeoff between area, performance and the overhead associated with them. Typically, tasks with constantly higher processing demands can be fixed as HW tasks such that they can be accelerated with reconfigurable hardware. On the other hand, tasks with lower processing demands can be fixed as SW tasks. In the same way, a task with variable intensities can be considered as a good candidate for a pageable task - as the intensity changes, the task can be mapped onto different resources. Several profiling and program analysis tools can be employed to make such decisions. Nonetheless, this can also be subjectively decided by the system designers depending on the application and on the architecture.

By allowing a system to have a mechanism to fix few tasks as HW, SW and pageable, a tradeoff can be accomplished, for instance between **resources** and **performance**. The execution of a task *always* on the reconfigurable hardware consumes expensive resources, while its execution *always* on the GPP has a slow performance compared to the hardware execution. A benefit can be made by changing the execution of tasks between the GPP and the reconfigurable

hardware depending on the requirement of the task. This is where a pageable task plays an important role.

Figure 3.5 depicts an example, which shows how pageable tasks change their mapping behavior and use different resources for the execution. As it can be inferred from the figure, tasks J and D change their mapping at each period. Task E changes its mapping from HW to SW only once in period i+2, and it executes as HW again in the last period. Similarly, Task C changes its mapping only once. This implies that tasks J and D have more flexibility to be executed on both platforms compared to tasks C and E. Nevertheless, they also have more overhead associated with their mapping change.

Let us consider the total execution time for a pageable task is denoted by t_{page}. This execution time can be calculated as:

$$t_{page} = \sum_{i=1}^{n} (t_{HW_i} + t_{Recon_i}) + \sum_{j=1}^{m} t_{SW_j} + p \cdot \rho \tag{3.1}$$

where, given a task t, t_{HW} is its hardware execution latency, t_{SW} is its software execution latency, and t_{Recon} is its reconfiguration latency. The task t executes n times on the reconfigurable hardware, m times on the GPP and it changes its mapping p times. ρ is the overhead of changing the mapping for the given task.

The change of mapping for a pageable task from a SW set to a HW set is beneficial only if the factor of accelerating it with hardware is higher than its relocation overhead such that:

$$\rho < r \cdot [t_{SW} - (t_{HW} + t_{Recon})] \tag{3.2}$$

where r is the number of times the task is executed on the hardware instead of executing it on its software counterparts after changing its mapping. Note that if a task configuration is already present in the hardware, the task is not configured, in those cases, t_{Recon} in Equation 3.2 is set to 0.

With runtime application mapping of tasks onto the reconfigurable architecture, the total application execution time can be calculated as:

$$T_{runtime} = \sum_{i=1}^{m} [t_{HW_i} + t_{Recon_i}] + \sum_{j=1}^{n} t_{SW_j} + \sum_{k=1}^{p} t_{page_k} \tag{3.3}$$

where m, n and p are the number of HW tasks, SW tasks and pageable tasks respectively. As a result, the total application execution time of tasks for the static application mapping onto the reconfigurable architecture is:

$$T_{static} = \sum_{i=1}^{m} [t_{HW_i} + t_{Recon_i}] + \sum_{j=1}^{n} t_{SW_j} \tag{3.4}$$

As the set of pageable task is empty, only HW and SW sets exist in static application mapping. Let us assume that the performance in this case is measured in terms of speedup. The corresponding speedup with static and runtime application mapping can be calculated as follows:

$$\text{Performance (runtime)} = \frac{T_{runtime}}{T_{SW}} \tag{3.5}$$

$$\text{Performance (static)} = \frac{T_{static}}{T_{SW}} \tag{3.6}$$

where T_{SW} is the total application execution time when all the tasks are executed as software. Let us assume that the average area utilization with the static and the runtime application mapping is A_{static} and $A_{runtime}$ respectively. The cost of the design is proportional to the area it consumes. The ratio of performance per cost can be calculated as follows:

$$\text{Ratio (static)} = \frac{\text{performance (static)}}{A_{static}} \tag{3.7}$$

$$\text{Ratio (runtime)} = \frac{\text{performance (runtime)}}{A_{runtime}} \tag{3.8}$$

This ratio can be used to compare static application mapping and runtime application mapping in terms of cost and performance. Moreover, it can also help to explore if identical performance can be achieved with the runtime mapping instead of the static mapping with a lower utilization of resources. The runtime application mapping can utilize computation-per-resource cost more efficiently than the static mapping. A mapping with higher performance-per-cost ratio is better as it better utilizes the area. Obviously, there is a performance area tradeoff. The interesting point, however, is to find out the "optimal" cost that can incur in order to achieve a particular performance. Another

important issue is to measure the resource penalty that has to be paid in order to increase the performance of the application by a certain factor. Pageable tasks play an important role for exploring such tradeoffs.

3.5 Two-Level Mapping Exploration

The static exploration of a system alone is not sufficient for any kind of architectural exploration, due to changing runtime conditions of the system. Such conditions can occur in the system with respect to, e.g., user requirements, or having multiple simultaneously executing applications competing for platform resources. Static exploration is carried out often under offline system conditions without considering any change in the system (application, architecture and/or environment). Such exploration can generally be faster. However, it is less accurate as it does not consider the runtime behavior of the system. With offline system condition, applications that run in parallel and their respective user requirements are unknown. Thus, while performing static exploration, such runtime behavior of the system is mostly captured by static estimations and predictions. As a result, the precision of static time exploration is compromised.

On the other hand, with runtime exploration, a design candidate is evaluated for (dynamically) varying system constraints. Any change in the system (application, architecture and/or environment) is given as a feedback to the evaluation process. As a consequence, the design parameters are adjusted during the evaluation based on the changes encountered by the system. The runtime exploration of a design candidate can provide better accuracy compared to the static exploration, as it takes into account certain feedback from the system. Nevertheless, such evaluation is typically hard to obtain due to the enormous size of the search space generated by the runtime system parameters.

In order to benefit from both static and runtime exploration, we propose an exploration approach, which combines a static mapping exploration together with a runtime mapping exploration. At first, the static mapping exploration performed under static conditions leads to a set of static mappings. Each mapping is characterized by a set of HW, SW and pageable tasks, and it is subject to the combination of used architectural resources, costs and constraints. After that, the runtime mapping exploration performs a high quality exploration of these task sets to address any runtime change in the application, in the archi-

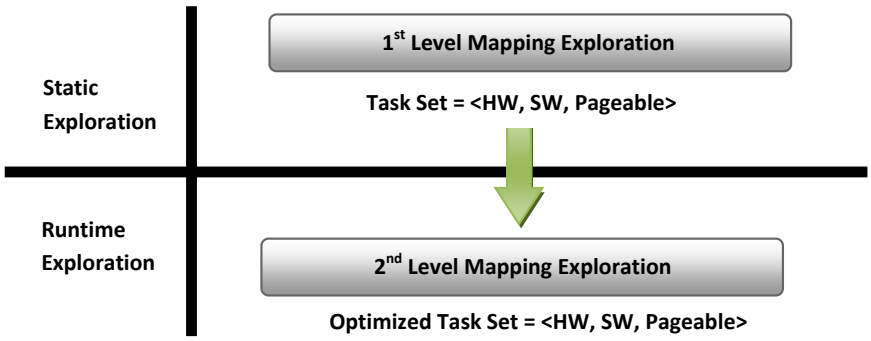


Figure 3.6: The two-level mapping exploration. The first level of mapping exploration performed statically, identifies a mapping that consists of a set of the HW, the SW and the pageable tasks. The second level of mapping exploration performed to address the runtime system condition, optimizes that mapping based on the runtime system conditions.

tecture, or in the environment. In this way, with a faster static exploration, it is possible to evaluate large set of different mappings, and further on, the detailed runtime mapping exploration can evaluate a set of selected mappings for better accuracy.

The two phases of the two-level mapping exploration are depicted in Figure 3.6. The static mapping exploration is carried out at the first level, and it results in static sets of tasks. These sets consist of HW, SW and pageable tasks. With the runtime exploration, these task sets are optimized at runtime. Such runtime exploration, typically, consists of a low complexity runtime mapping manager that is responsible for performing runtime mapping decisions in order to optimize mappings based on runtime changes in the system (the application, the architecture or the environment). The detailed discussion of such a runtime mapping manager is provided in Section 3.6. In the following, the two-level mapping exploration is discussed more in detail.

3.5.1 Static Mapping Exploration - The First Level

Static mapping exploration is performed at the first level of the exploration. At the first level, we address the exploration of the system under *static* conditions. Static, here, refers to the condition of the system where applications, the architecture and the environment do not change during the exploration. The system constraints imposed in such conditions are fixed and known during the

exploration. For instance, a condition where a single application is mapped onto a fixed architecture is an example of a static condition. In this case, the exploration is performed to address static system condition by iteratively evaluating different mappings until the most suitable mapping satisfying the given system constraints are identified. If multiple mappings are found, one or more mappings are selected. This set of mapping is considered "optimal" in a predefined way. Static exploration can be carried out using any static exploration methodology, such as the ones presented in [24, 90, 91]. Each mapping identified in this case is associated with a set of HW and SW task sets. This is used as a starting point for the runtime exploration in the second level of the exploration. At the end of the static exploration process, certain tasks out of HW and/or SW sets are considered as pageable, which can be optimized with the runtime mapping exploration.

There are several advantages of performing the static exploration. First, it allows faster exploration. Second, it is applicable in every embedded system environment, as it does not require runtime information. Additionally, it provides a good coverage for the exploration of an enormous size of the search space. One of the major disadvantages of static exploration is that the mapping decisions made under static system conditions can often only cover certain scenarios, and it fails in efficiency when hard-to-predict system scenarios occur. This process becomes more difficult especially in dynamic systems, where changes in the user requirements and architectural parameters are often unpredictable in advance.

3.5.2 Runtime Mapping Exploration - The Second Level

At the second level, mapping exploration is performed to address the runtime conditions of the system. In this case, the pageable task set identified with static exploration is optimized in order to address the runtime conditions of the system. This is done by dynamically changing the mapping, for such tasks, from one resource to another at runtime, such that the given design constraints are satisfied. For example, in order to meet the real-time constraints of a sporadic task with faster execution, few tasks can be migrated from the reconfigurable hardware to the GPP, in order to give priority to the sporadic task. Similarly, if a sudden increase in the application load is detected, performance can be enhanced by moving tasks from the GPP to the reconfigurable hardware.

The main advantage of performing runtime exploration is its accuracy while exploring different mappings. As it considers real runtime system scenarios, it can be efficiently used to explore dynamic and hard to predict system conditions. Nevertheless, runtime exploration is slower compared to static exploration, when it is used to explore the enormous search space. The main objective of the runtime mapping exploration is to allow pageable tasks identified in static exploration to change their mapping at runtime, such that mapping can be optimized to satisfy any change in the system.

As mentioned before, with the static exploration, tasks are classified onto HW, SW and pageable sets. In the runtime mapping exploration, these tasks are refined using the knowledge from any change in the system. To perform such refinement, we change the mapping of tasks onto different resources at the runtime. As a result, at the end of the runtime exploration, a refined task set is obtained together with the policy that is used to perform mapping change at runtime. Note that the policy for changing mapping at runtime is bound to the classification of the tasks onto different task sets. On the contrary, different policies implemented for changing the application mapping, results with different refinements.

The runtime mapping exploration involves various issues, which are discussed in the following.

- The first issue that needs to be addressed while performing runtime mapping exploration is to understand the criteria to decide if a current mapping is not adequate and it has to be changed. Most importantly it is necessary to identify whether the current mapping is sufficient to satisfy the given system constraints or not. For this purpose, several pre-defined system criteria can be used based on various design parameters, such as performance, resource utilization and power consumption. Based on these criteria, the system can decide to change the mapping for one or more pageable tasks at runtime.

- A system needs to support task migration from one resource to another to allow a pageable task to change its mapping at runtime. Task migration between various resources has always been seen as a way to perform dynamic load distribution, to ensure fault resilience, facilitate system administration and to enhance data access locality [121–124]. In this case, the migration of pageable tasks is possible from the GPP to the reconfigurable hardware and vice versa. For instance, if the reconfigurable

hardware cannot accommodate the current HW task, a set of pageable tasks can be migrated from the reconfigurable hardware to the GPP, thus freeing resources for its execution. Furthermore, if a part of the reconfigurable hardware is left idle, and if there are not enough HW tasks, the pageable tasks must be migrated from the GPP to the reconfigurable hardware to promote the use of the hardware. Moreover, whenever user requirements change (e.g. switching to another resolution in a video application) or in case of hardware failure, the runtime application can use runtime task migration to re-allocate resources to overcome the change.

- Once the criteria for pageable tasks have been identified to migrate to a different resource, another issue that needs to be addressed is to decide for which pageable tasks to change the mapping. This involves several issues, such as *which* pageable task to migrate and *where* to migrate it. To tackle this, a system needs to implement certain policies for decision making. These decision making policies have to be chosen based on various pre-defined system criteria. Typically, such strategies are chosen based on the tradeoff between the solution space search mechanism and the computation requirement for that search. At runtime, a fast, light weight algorithm that comes up with a reasonable good solution is generally preferable over an algorithm that comes up with an "optimal" solution requiring a lot of computations. A number of algorithms can be used for making such runtime decisions. These algorithms are discussed in Section 2.3.

- Once the migration criteria are known and the mapping decisions are identified, the mapping decision can be made and the pageable tasks can be migrated to another resource. Nonetheless, another issue that needs to be addressed at this point, while migrating tasks at runtime, is how to handle the task state while migrating the task and how to ensure functional and communication consistencies of the system. For instance, to maintain the system consistency, the tasks can be migrated at pre-defined execution points. Performing such migration at predefined execution points, the task state can be saved and the context can be shifted to another resource. In this way, the functional as well as the communication consistencies can be ensured. In another way, the task can be migrated when it is not actually running but in the waiting state. The task can be in the waiting state if it is waiting for the resource to be free or due to the data dependency on another task. Nevertheless, task migration always incurs some overhead. A tradeoff has to be made between the cost and

benefits of the migration. A number of techniques have been devised to alleviate this migration cost, as discussed in [121–124].

3.5.3 Two-Level Mapping Exploration Illustration

To illustrate the two-level mapping exploration, let us consider a scenario with two applications App1 and App2. App1 is a regular application that runs on the given reconfigurable architecture. Application App2, in this case, is sporadic and, as a result, it can go in and out of the system at random time forcing the application to behave dynamically. The first level exploration identifies HW, SW and pageable task sets for each application separately. In this case, we assume that each application is separately mapped onto the given architecture separately. This exploration starts with an initial random solution and different task sets are identified by searching the design space with iterative simulations. Let us denote the most suitable task sets for application App1 and App2 as $App1_{SW}$, $App1_{HW}$, $App1_{page}$, $App2_{SW}$, $App2_{HW}$ and $App2_{page}$.

In the second level of the exploration, the application model App3 is considered as a combination of App1 and App2. Therefore, the initial task sets for the second level of the exploration can be defined as follows:

$$App3_{HW} = App1_{HW} \cup App2_{HW}$$
$$App3_{SW} = App1_{SW} \cup App2_{SW}$$
$$App3_{page} = App1_{page} \cup App2_{page}$$

In order to address the "dynamic behavior" of the application model App3, a number of explorations have to be carried out at runtime in order to find the most suitable mapping for the pageable task set. With the new system condition, when application App2 starts to execute in the system, the tasks in $App3_{page}$ have to compete for resources. As a result, subsets of $App3_{page}$ may predominantly be executed either on hardware (P) or on software (Q). The task sets then can be changed accordingly, as follows:

$$\begin{aligned} App3_{HW} &= App1_{HW} \cup App2_{HW} \cup P, \\ App3_{SW} &= App1_{SW} \cup App2_{SW} \cup Q, \\ App3_{page} &= (App1_{page} \cup App2_{page}) - P - Q, \end{aligned}$$

where $P \geq 0, \quad Q \geq 0.$

Note that, although not all the tasks in the HW task set may fit on the reconfigurable hardware at the same time, all tasks can be executed with a delay after reconfiguring them onto the hardware. In such cases, in order to avoid the reconfiguration delay, few tasks from the HW task set can also be moved to a pageable task set. At any point during the exploration process, designers can influence the process by manually setting up these task sets using their domain knowledge, or by using heuristics. For instance, the designer may decide to manually fix certain tasks from one task set to another (e.g. when there is no hardware implementation available for a certain task), in order to reduce or enlarge the design space for iterative simulation.

System designers can also define, a priori, which tasks are mapped statically onto hardware or software. At the same time, the designers can additionally define a set of pageable tasks. In such case, the runtime search space is the largest when all the tasks are defined to be pageable. However, it also has the maximum overhead for changing their mapping. Typically, system designers have knowledge about the system (application and/or architecture) and they can specify certain tasks as hardware to run on the reconfigurable hardware (eg. computation intensive kernels) and certain tasks as software to run on the GPP (eg. control dominated tasks or tasks for which no RP implementation is available). The rest of the tasks can be specified as the pageable task set so that their mapping can be dynamically optimized based on the system behavior. In this way, system designers can tailor the exploration process to suit their purpose.

3.6 The Runtime Mapping Manager

In a reconfigurable system, in order to address the previously discussed runtime task mapping issues, and to perform runtime mapping exploration, there is a need for a runtime decision making component. The *Runtime Mapping*

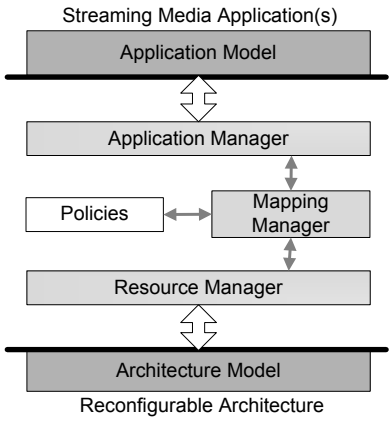


Figure 3.7: The structure of a typical Runtime Mapping Manager (RMM). A RMM consists of an application manager, a resource manager, a mapping manager and task mapping policies.

Manager (RMM) is such a decision making component that can be employed to make *intelligent* mapping decisions in order to address the runtime system conditions. Typically, the RMM resides in between the application and the architecture, and it is responsible for making any kind of mapping decisions at runtime. These mapping decisions can be made based on the dynamic system conditions, such as the application, the architecture or the system environment. In general, for dynamic systems the RMM has to identify whether the current mapping is sufficient to satisfy the given system constraints or not. If not, then it should also be able to identify for which tasks to change the mapping. More importantly, the RMM should identify *which* tasks to migrate and *when* to change the mapping to perform the migration. For this, various types of information about the runtime conditions of the system can be used - application information such as task priority, real time constraints, and architecture information, such as free resources and timing information.

Figure 3.7 outlines the structure of a typical RMM. The main responsibility of the RMM is to decide what services should the RMM provide with respect to the application and the architecture in order to satisfy various system constraints. Therefore, it is crucial for the RMM to understand both the application requirements and the architecture behavior. As it is depicted in Figure 3.7, there are four basic functionalities contained within the RMM: the application manager, the resource manager, the mapping manager and the mapping policies. This structure of the RMM is the basis for the runtime map-

ping exploration, which we believe to be generic enough to match many other industrial and academic platforms that support runtime mapping management. In this case, we consider that the RMM is placed between the application and the architecture (see again Figure 3.7). In the actual system implementation, the RMM entity may be part of the application, middleware, operating system or even implemented as a hardware component. By separating the application management from the resource management and by creating a set of well defined interfaces between them, we enable reuse and interchangeability of the runtime management components. In addition, such modularity conforms to the *separation of concerns* principles and also complies with the platform based design paradigm. For this particular reason, the RMM is structured with functionalities distributed among different modules. In the following, we present a general description of each component involved in the runtime mapping management and its respective roles and responsibilities.

3.6.1 The Application Manager

The application manager interacts with the application layer and monitors any change in the application, such as change in user requirement, arrival of sporadic tasks, priority of tasks and real time constraints. It deals with these task requirements and administers QoS management, e.g., in case to ensure the execution at a particular frame rate to maintain the required image quality. The application manager is a platform independent component and it is only responsible for dealing with applications. In case of any change in the application, it notifies the mapping manager for optimizing the mapping decision. Other application characteristics, such as real time application constraints and task priorities are also handled by the application manager.

3.6.2 The Resource Manager

The resource manager is a platform dependent component and it gathers information about the architecture platform. It administers the architecture behavior, and it provides various types of architectural information to the mapping manager, such as free resources and timing information. In case of any change in the architecture (e.g. component failure, power safe mode), it reflects these changes to the mapping manager for optimizing the mapping decisions. Figure 3.8 illustrates the structure of a typical resource manager. It consists of

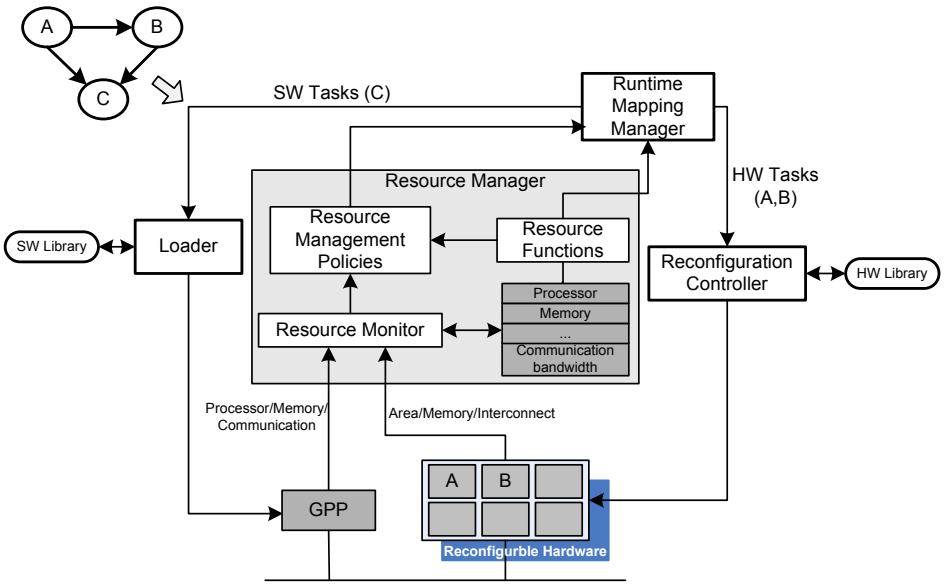


Figure 3.8: The structure of a typical resource manager that consists of a resource monitor, resource management policies and resource functions. The resource manager interacts with a runtime mapping manager, a reconfiguration controller and an application loader.

platform dependent components, the resource monitor and resource functions, and a platform independent component, the resource management policies.

The resource monitor unit systematically keep track of all the resources available in the architecture. These resources are stored in a *list*, which can also be accessed by the runtime mapping policies unit. If any change in the architectural resources is noticed, the resource monitor updates its entry in the list. The implementation of the resource monitor is done in a platform dependent way. As a result, the functionality of the resource monitor can change based on the reconfigurable platform used. The resource functions are predominantly derived from the resource list. Consequently, based on the type of resources considered in the system, the resource functions change, making them platform dependent components. These resource functions are used by the mapping manager unit to determine the task mapping. Moreover, the resource functions can also be used to determine the overall cost or quality of the mapping. The resource management policy, on the other hand, determines how the resource space is searched.

Note that, in this case, there are two management units, one dedicated to the resource management and another one dedicated to the mapping management. In other cases, the resource management unit can be coupled with the runtime mapping policy. This depends on the choice of the policy and implementation of the system. The key point however is to separate the management decision from the platform dependent component to achieve flexibility and modularity.

The reconfiguration controller manages the reconfiguration process and the configuration state of the reconfigurable hardware. The main task of this unit is to load the compile-time prepared hardware implementation of a HW task.This involves accessing the hardware library and configuring the appropriated task onto the hardware.

The loader, on the other hand, is responsible for loading SW programs from the software library onto the GPP. Loading a program involves reading the contents of an executable file, loading the file containing the program text into memory, and finally it involves carrying out other required preparatory tasks to run the executable onto the GPP.

3.6.3 The Mapping Manager

The mapping manager is responsible for making actual mapping decisions. For executing these decisions, the mapping manager has to employ one or more mapping mechanisms. A mechanism describes a set of actions, the order in which they need to be performed, and their respective preconditions. In order to get such information, the mapping manager collaborates with the resource manager and the application manager (see Figure 3.7). Based on application information from the application manager and architecture information from the resource manager, the mapping manager finds the mapping at runtime. The mapping manager can be designed to learn from its execution history, to predict future requirements, or can employ several mapping policies to optimize the mapping. These mapping policies are implemented as a modular component. As a result, a variety of policies can be easily plugged in and out of the system.

3.6.4 Mapping Policies

The runtime mapping manager employs various strategies for decision making. Different types of algorithms can be used in this context depending on, e.g., the tradeoff of mapping speed versus mapping quality. Optimal mapping policies often require longer time for finding a "best" mapping compared to heuristics based solutions that use few simple rules to determine a "good" solution. These decisions can be categorized as static and dynamic based on when the task assignment is performed. More information about different types of mapping algorithms has been presented in Section 2.3. The task mapping decision of such policies is also influenced by the platform-specific resource functions. The mapping manager can choose one or more of these policies to optimize the mapping based on the system condition. Note that, in this context, there is a dedicated component for carrying out the mapping decisions. Nonetheless, this decision making process can be implemented in the actual system in any number of different ways, e.g., in the application, in the hardware or in the operating system as a system policy.

3.7 Conclusions

In this chapter, we presented an overview of the runtime mapping exploration of reconfigurable architectures. Within the context of the dissertation, the applications are represented by using the KPN model of computation. We identified three types of tasks in the system: *software* tasks, *hardware* tasks and *pageable* tasks. The chapter provided a detailed discussion of static and runtime application mapping. Static mapping is performed under static system conditions, where tasks are mapped onto only one resource during their execution, while runtime application mapping allows the tasks to be executed on different resources during their execution. Based on the static and the runtime application mapping, the chapter outlined the two-level mapping exploration mechanism, in which both static and runtime mapping explorations are performed. At the first level, a static exploration is performed, which identifies a set of candidate mappings that is obtained by considering only the static condition of the system. These mappings consist of HW, SW and pageable tasks sets. After that, at the second level, the runtime exploration performs a high quality exploration at runtime to optimize these mappings. More importantly, the mappings are optimized by allowing pageable tasks to be executed on dif-

ferent resources during their execution, such that the any change in the system (the application, the architecture or the environment) is satisfied. Finally, the chapter elaborated the definition and the structure of the RMM. The RMM is a key component, which performs an intelligent mapping decisions at runtime, by deciding *where* and *when* to change the mapping of pageable tasks, based on the imposed system condition. The RMM consists of a resource manager, an application manager, a mapping manager, and mapping policies.

In the next chapter, we propose a framework that can realize the proposed two-level mapping exploration. The proposed framework can be used to carry out both the static and the runtime mapping exploration of reconfigurable architectures. Additionally, the framework can model and simulate reconfigurable architectures, and it can also perform system-level DSE with respect to hardware-software partitioning, task mapping, task allocation and task scheduling.

Note. The content of this chapter is based on the the following articles:

K. Sigdel, M. Thompson, A.D. Pimentel, C. Galuzzi, K.L.M. Bertels, **System-Level Runtime Mapping Exploration of Reconfigurable Architectures**, *Proceedings of the Reconfigurable Architectures Workshop (RAW'09)*, Rome, Italy, May 2009, pp. 1-8.

K. Sigdel, M. Thompson, A.D. Pimentel, K.L.M. Bertels, **Towards System Level Runtime Design Space Exploration of Reconfigurable Architectures**, *Proceedings of the Annual Workshop on Circuits, Systems and Signal Processing (ProRISC'08)*, Veldhoven, The Netherlands, November 2008, pp. 1-8.

4

rSesame Framework

In the previous chapter, we proposed the two-level mapping exploration methodology, which can explore reconfigurable architectures statically as well as at runtime. In this chapter we develop a sytem-level framework, called *rSesame*, which can implement the two-level mapping framework described in the previous chapter. The rSesame framework is a generic modeling and simulation framework, which can explore and evaluate reconfigurable systems at the early design stages. The framework can be used to perform system-level Design Space Exploration (DSE) by performing rapid investigation of several parameters such as, architectural characteristics, application-to-architecture mappings, scheduling policies and hardware/software partitioning, both statically and at runtime. It provides various important system attributes, such as execution time, number of reconfigurations, time-weighted area usage, percentage of hardware-software mapping, percentage of reconfiguration, and hardware reusability efficiency. The main features of the rSesame framework include flexibility, ease of use, fast performance, and its applicability to a wide range or reconfigurable systems.

The rSesame framework has been implemented on top of the Sesame framework. Sesame [2, 21] is a modeling and simulation platform for system-level DSE. To this end, the Sesame framework is extended to model and simulate reconfigurable architectures at runtime. In this chapter, we will demonstrate the modeling methodology behind the rSesame framework with an example of a generic reconfigurable architecture. More specifically, we discuss how the Sesame framework is extended to model reconfigurable architectures both statically as well as at runtime. We performed two major extensions on the Sesame framework. Firstly, it is extended to model and simulate the behavior of reconfigurable architectures. In the second extension, we extended

the Sesame framework to allow runtime exploration of various architecture-application-mappings of such architectures. The rSesame framework consists of the aforementioned extensions.

The chapter is organized as follows: Section 4.1 presents an overview of the Sesame modeling and simulation infrastructure. Section 4.2 discusses the necessary extensions in the Sesame framework to model reconfigurable architectures, and Section 4.3 discusses a necessary extension in the Sesame framework to facilitate the runtime application mapping of such architectures. Section 4.4 presents the rSesame framework, which consists of the aforementioned extensions. Finally, Section 4.5 provides the summary of this chapter.

4.1 The Sesame framework

In this section, we provide background information about the Sesame framework. The Sesame modeling and simulation environment [6,21,125] is geared towards fast and efficient exploration of embedded multimedia architectures, typically those implemented as heterogeneous MPSoCs. Using Sesame, a designer can construct system-level performance models, map applications onto these models, and explore the design space through high-level system simulation. For this purpose, Sesame uses a technique called *trace-driven co-simulation* [21] to map application models onto architecture models. Sesame adheres to a transparent simulation methodology, where the concerns of the application and the architecture modeling are separated via an intermediate mapping layer. An application model describes the functional behavior of an application. An architecture model defines the architectural resources and constraints. In the following, each layer of the Sesame framework is discussed in more detail.

The application layer

Sesame uses the Kahn Process Network (KPN) model of Computation [28] for application modeling. A KPN model consists of concurrent Kahn processes that communicate using blocking read/non-blocking write synchroniza-

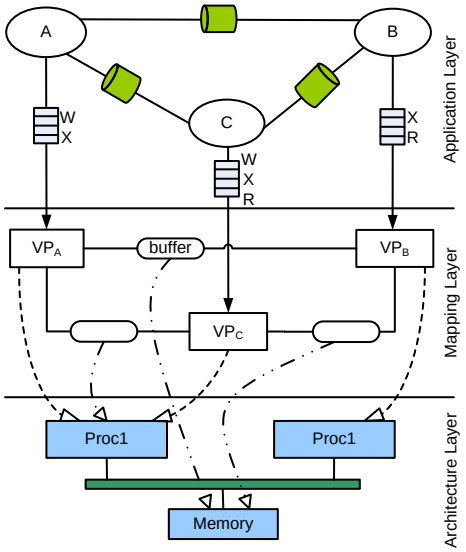


Figure 4.1: The three layers in the sesame infrastructure. The application layer and the architecture layer are separated via an intermediate mapping layer that consists of virtual processors.

tion over unbounded FIFO channels. These models are suitable for modeling stream-based (multi-media) applications. Additional information on KPN models has been presented in Section 3.1.

Sesame employs trace-driven co-simulation. To this end, the code of each Kahn process is equipped with annotations that describe the action of each process. When such a process is executed, it generates its own trace of events which represent the application workload imposed on the architecture by that specific process. These events are coarse-grained computation and communication operations, such as read (R), write (W) and execute (X). The events R and W describe FIFO channel communication and the event X describes the computations performed by a Kahn process (typically a function). These events are actual primitives that drive the architecture simulation. They are collected into event traces that are mapped, using an intermediate mapping layer, onto an architecture model (see Figure 4.1).

The mapping layer

In the Sesame framework there is an intermediate layer between the application

model and the architecture model called *mapping layer*. The mapping layer acts as an interface for mapping Kahn processes from the application layer onto the architecture model components. This layer is also responsible for the scheduling of application events at runtime when multiple Kahn processes are mapped onto a single architecture component.

The mapping layer consists of Virtual Processors (VPs) and FIFO channel components of a bounded size, which are connected using the same network topology as the application model (see Figure 4.1). The main purpose of the VPs is to forward the event traces to components in the architecture model and, hence, to drive the architecture model component for co-simulation. This forwarding is done according to a user-specified mapping of application processes and communication channels. Application processes are mapped onto processors and communication channels are mapped onto communication structures. The components in the mapping layer simulate the synchronization of communication events in such a way that forwarded events are "safe", meaning that they do not cause any deadlock due to unmet data dependencies when application processes are mapped onto shared resources. This mechanism also ensures deadlock free scheduling when application events from different event traces are merged. Unlike the application model, which is un-timed, the mapping and architecture layers are modeled in the same timed simulation domain.

The architecture layer

The architecture model is constructed from generic building blocks provided by a library, which contains template performance models for processors, co-processors, memories, buffers and buses. The architectural timing consequences of the events are modeled in the architecture model. This requires each Kahn process and channel of the KPN to be mapped onto one component of the architecture model. The trace events generated by each Kahn process are routed via the mapping layer to their corresponding architecture component, which then models the appropriate timing consequences.

The processor components of the Sesame framework model the processor utilization of the simulated architecture. Similarly, interconnection and memory components model the utilization and the contention caused by communication events. For modeling the processor utilization, the processor component uses a lookup table that associates the computational (X) events to execution

latency. By changing the latency assigned to these events, a generic processor model component can simulate the timing behavior of a particular simulated processor, which can either be a reconfigurable processor or a dedicated hardware implementation. These latency values may be obtained from existing literature, hardware measurements, rough estimates, predictions tools, such as the one presented in [120], or even from more detailed simulators, such as the one described in [2].

The initial high level of abstraction of the Sesame models allows for very fast simulation, typically in the order of 1 system-level application/architecture co-simulation per second. By extending the Sesame framework with a structured exploration environment as done in the context of Daedalus [126]), it is possible to efficiently perform early DSE.

4.2 Extension of the Sesame framework to model Reconfigurable Architectures

In this section, we describe an extension of the Sesame framework for modeling reconfigurable architectures. The extensions listed in this section are designed to be as generic as possible, and applicable to any type of reconfigurable architecture. However, to describe some extensions, we assume a typical reconfigurable architecture. For this purpose, we consider the dynamically reconfigurable architecture as depicted in Figure 4.2. The architecture consists of a core processor that is a GPP, and a reconfigurable hardware. The reconfigurable hardware consists of one or more Reconfigurable Units (RUs). The GPP, RUs, a memory and a reconfigurable controller, all are connected using a shared bus. In order to speedup the execution of a program by using reconfigurable hardware, parts of the program code can be implemented on these RUs. The reconfiguration controller configures these task onto RUs at runtime.

To model the correct behavior of the reconfigurable architecture with the Sesame framework, the following behaviors have to be modeled.

- The main feature of dynamically reconfigurable architecture includes task configuration at runtime. When a task is mapped onto RUs, the task is configured, which adds a certain reconfiguration delay. Such a reconfiguration delay has to be modeled.

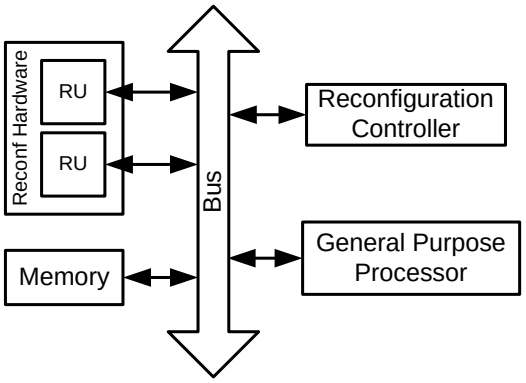


Figure 4.2: A generic reconfigurable architecture with a GPP and a reconfigurable hardware, which consists of one or more RUs. The GPP, RUs, the reconfiguration controller and the memory, all are connected to a peripheral bus.

- The tasks mapped onto the RUs consume certain amount of resources in the reconfigurable hardware. Such resource occupancy has to be modeled.
- Similarly, the reconfigurable hardware is limited by the resources. When the tasks are mapped onto the hardware, these resources have to be managed efficiently at runtime. Such resource management behavior also has to be modeled.

In order to accommodate the above listed behaviors of the reconfigurable architecture, we implemented extensions in each layer of the Sesame framework. In the following sections, we describe these extensions in more detail.

4.2.1 Construction of the Reconfigurable Architecture

As in conventional heterogeneous MPSoC architecture design, the reconfigurable architecture also consists of many different types of architectural components, such as processors, memories and interconnects. To model reconfigurable architectures, first utmost requirement is to be able to construct an architecture that consists of these components. Architecture models in the Sesame framework can be constructed from generic building blocks provided as a library, which contains templates for processors, memories, buses and on-chip networks. Several features have to be added to these components, in

order to accommodate the additional features of reconfigurable architectures as discussed before. With these features added in these components, a reconfigurable architecture can be easily constructed using the Sesame templates.

As mentioned before, reconfigurable architectures are generally composed of two kinds of architectural components: the GPP and RUs. The RUs are connected via a common bus to a shared memory or a different network component (e.g. a crossbar). A task to be executed in the hardware is mapped onto these RUs. With the Sesame framework, we intend to explore reconfigurable architectures at a high abstraction level. At this level of abstraction, the characteristics of the GPP and RUs are comparable, except that RUs have a few additional attributes, such as resource occupancy and the reconfiguration delay. As a result, in the rSesame framework, the GPP and RUs can be represented by one and the same modeling component for a processor. In this case, the RUs are given extra attributes for resource occupancy and reconfiguration delay, in addition to the general attributes, such as task execution latency. Since the functionality of the RU changes depending on the task mapped onto it, there can be as many RUs as the number of tasks mapped onto the reconfigurable hardware, and each RU can have different attributes. In Figure 4.3, an example of a reconfigurable architecture constructed from the Sesame framework is depicted together with the mapping of an application onto it.

4.2.2 Modeling of the Reconfigurable Behavior

Dynamic reconfiguration provides the ability to change the hardware configuration during the execution of the application. Partial dynamic reconfiguration allows the overlapping of computations with reconfigurations to significantly reduce the reconfiguration time overhead. Through partial reconfiguration, tasks can be reconfigured in hardware individually, without interfering with other tasks already running on the same hardware in other stages. This enables a larger percentage of the application to be accelerated on the reconfigurable hardware, hence reducing the overall execution time. However, the reconfiguration of the hardware introduces a reconfiguration delay (e.g. bitstream loading time for an FPGA), which can negatively affect the performance, and it can also increases the power consumption [16, 32].

The main feature of the dynamically reconfigurable architecture is task configuration at runtime. Typically, the reconfiguration controller is responsible for performing such a functionality in the reconfigurable architecture as

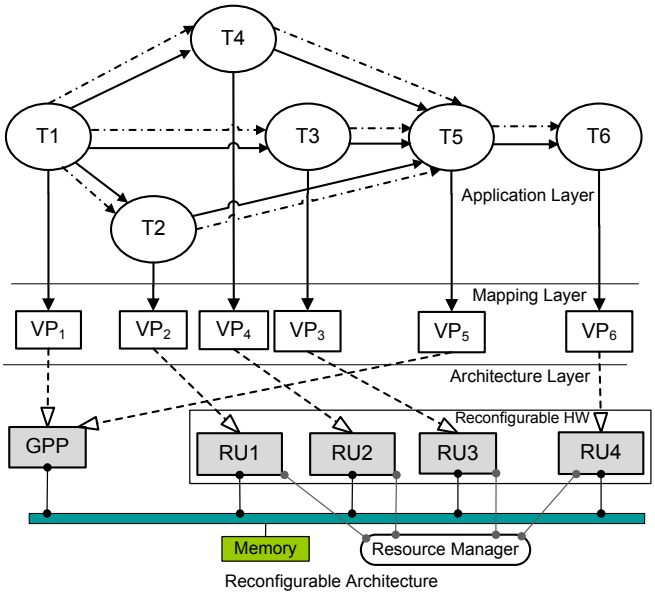


Figure 4.3: The three layers in Sesame's infrastructure for reconfigurable architectures. The architecture layer consists of a GPP and one or more RUs. The dashed lines in the application layer are the token channels added for synchronization purposes.

shown in Figure 4.2. Depending on the availability of the reconfigurable resources (e.g. area slices, memory components, lookup tables, registers, DSP elements, wire segments and interconnects) RUs can be configured and unconfigured on a particular reconfigurable hardware at runtime. To model this behavior with Sesame, we added an extra component in the architecture layer, called the Resource Manager (RM). The RM is responsible for keeping track of RUs and the resources consumed by them, and to configure/un-configure them on the reconfigurable hardware as required. When configuration data is loaded on the hardware for a particular task, the corresponding RU has to be configured on the reconfigurable hardware and the reconfiguration delay has to be modeled for that particular RU. The RM also allows performing such functionality. All RUs are connected to the RM, and the RM keeps track of them.

In summary, the responsibility of the RM is twofold. First, it is responsible for configuring and releasing RUs based on the resource availability in the reconfigurable hardware. Second, it is responsible for managing the resources on the reconfigurable hardware. In the following, we describe this process with

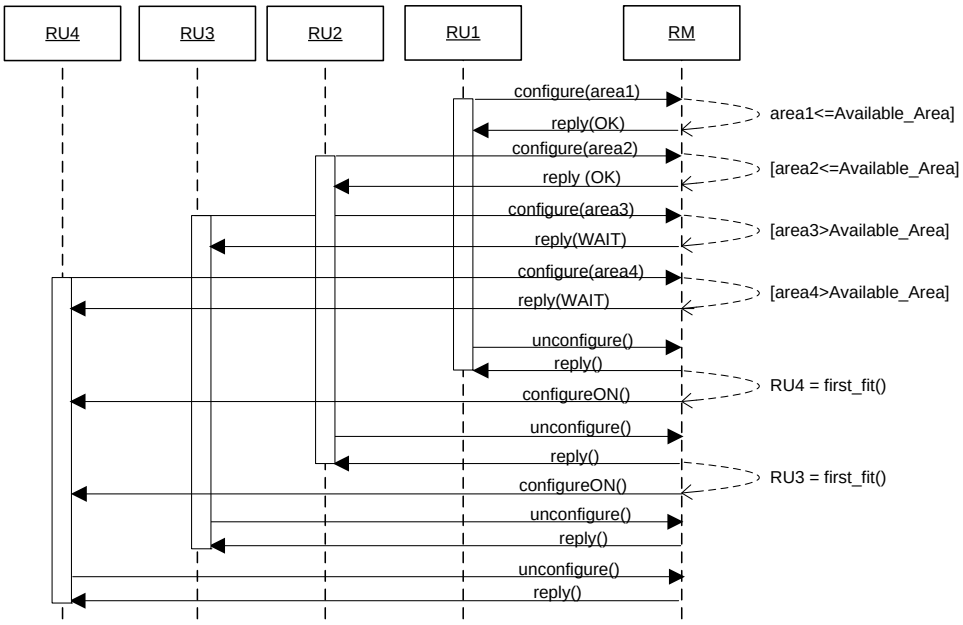


Figure 4.4: The sequence diagram showing an interaction between RUs and the Resource Manager (RM). The RM synchronizes various RUs and keeps tracks of architectural resources.

an example. Figure 4.4 shows the interaction between RUs and the RM. As it can be seen from the figure, when a task needs to be mapped onto a RU, the latter sends a request to the RM to be configured. The RM checks for the availability of resources[1] on the reconfigurable hardware, and it decides whether or not to configure a particular RU. If there is enough area available, the RU is configured immediately (RU1 and RU2 in Figure 4.4), otherwise it is blocked until sufficient area becomes available (RU3 and RU4 in Figure 4.4). Once the necessary area is available, the RU is configured and blocked to model the reconfiguration delay before it initiates the real execution of the events. This blocking time is added to the total simulation time of the system. In this way, the effects of the reconfiguration delay on the system performance are modeled. In the example shown here, a simple `first_fit` algorithm is used to schedule RUs onto the reconfigurable hardware, according to which, the first RU that fits onto the available hardware area is configured first. However, more refined task placement and scheduling algorithms for the reconfigurable

[1] In this particular case, we use area as a quantitative measure indicating the amount of resources used by the RU on the reconfigurable hardware, typically an FPGA.

hardware, as described in [45, 47–49, 122, 127], can also be implemented as a plug-in to the reconfiguration manager.

Note that, in Figure 4.3, the RM is modeled as a part of the architecture layer, assuming that the management of RUs is performed in the architecture layer. However, in other reconfigurable systems, this management can also be defined in other fashions, such as at operating system or at user-level. In those cases, the functionality of the RM can be modeled in any other layer of the framework, such as in the mapping layer. A detailed discussion of a typical RM has been presented in Section 3.6.

4.2.2.1 Addition of the Reconfiguration Event

To indicate possible reconfiguration points, an extra execute event (X_p), is added in the application layer. This reconfiguration point can be defined as a possible point for the reconfiguration of a task onto the reconfigurable hardware. Note that, a task is not always configured when mapped onto a reconfigurable hardware. If the hardware configuration for the task is already present in the hardware, the task is not reconfigured. Therefore, based on the X_p event, the RM can decide when to configure and/or un-configure a particular task.

As discussed in Section 4.1, in a Sesame application model, a Kahn process generates events when reading (R) or writing (W) to/from a channel, and while performing computations (X). Often a Kahn process reads data, performs computation on that data and writes the result to another process. Since our Kahn processes are typically a part of a streaming application, each process usually consists of a loop that iteratively performs computations on certain data items, which are passed on through the network. A single iteration of the loop can be considered as a complete and atomic operation to be performed by the process.

With the aforementioned assumption, the X_p event is added at the end of each iteration of our Kahn process, in order to enable the reconfiguration point. By adding the X_p event, at the end of each iteration, reconfiguration is allowed only after a Kahn process performs a complete iteration of R, X and W events. In our Kahn process, this assumption about atomic execution makes sense because un-configuring the RU, once the data has been read, would simply cause the data to be lost. Similarly, the RU should not be un-configured before writing the result otherwise the computed result could be lost. In this

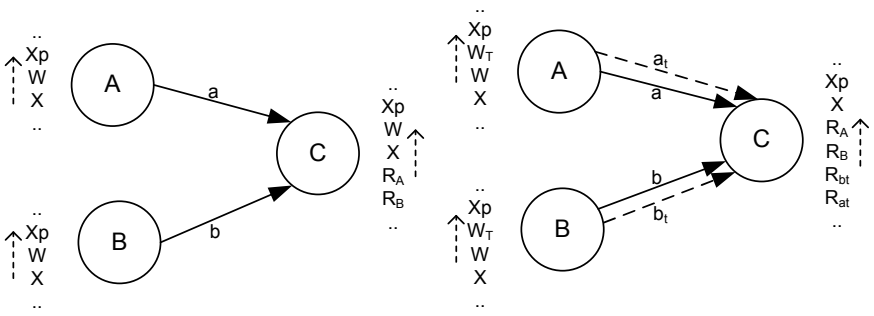
way, a process guarantees that it generates X_p event only after it finishes a set of events for that iteration. As a consequence, its corresponding RU is automatically required to complete a part of the application execution before being un-configured from the reconfigurable hardware. The implementation is performed in such a way that whenever an X_p event is encountered, the RU can always request the RM for its un-configuration. However, the RU can always request for its configuration (if not already configured), whenever it receives any other event.

4.2.2.2 Managing Reconfigurable Resources at Runtime

The basic Sesame models are self-schedulable. This implies that R, X and W events from the application models are automatically scheduled to the architectural component without having an explicit (global) scheduler. Unlike simple processors that model only the simulation time, the RUs in the reconfigurable architectures, should model the simulation time and the hardware resources they consume. As a result, R, X and W events associated with RUs should also be able to model the simulation time and the resources they consume in the hardware. Therefore, adding one or more RUs on the reconfigurable hardware without considering their resources requirements can create a deadlock situation and jeopardize the self-schedulable property of the Sesame model.

Reconfigurable architectures are limited by the available resources. If not enough reconfigurable resource is available to load a task, the latter cannot be executed on the hardware. Such a resource constraint imposed by the reconfigurable hardware can add dependencies between the tasks mapped onto the RUs. As a result, if there are not enough resources available to load a RU, the latter has to wait until another RU executing on the hardware finishes its execution. While modeling reconfigurable architectures with the Sesame framework, in some cases, such resource dependencies between RUs can lead to deadlock situations in the architecture model. To avoid such deadlocks, we have restricted the KPN in the application layer to be acyclic. Additionally, to make sure that only safe events are forwarded from the mapping layer to the architecture models, we extended the application model to support an additional token channel, such that, each two communicating Kahn processes are also connected with a token channel. This can be better explained with an example in the following.

Let us consider a KPN as shown in 4.5(a). In the example, let us assume



(a) Without token channel added (b) With token channel added

Figure 4.5: An example of a KPN with and without token channel. The corresponding event sequences for each process are listed in the figure. The event sequence with token channel has two extra read R_{a_t} and R_{b_t} for the token channel a_t and b_t.

that each of the processes A, B and C requires more than half of the total area on the reconfigurable hardware for the execution. This implies that two processes cannot be executed on the reconfigurable hardware at the same time. As a result, there is an implicit resource dependency between these processes. Let us assume that process A configures on the hardware first, and it executes a sequence of events as shown in the figure[2]. The task executes a function (denoted by the event X), followed by writing the channel **a** (denoted by the event W) and finally the task is un-configured (denoted by the even X_p).

At this point, both processes B and C could start their configuration on the reconfigurable hardware at the same time. Due to the parallel nature of KPN processes, as soon as the data for C is available in the channel **a** written by A, C could also start its configuration on the hardware. If B starts its configuration, the scheduling is automatic, and there is no deadlock. Nonetheless, if C configures on the hardware before B executes and C starts executing its event sequence (X_P, R_b, R_b ... X_P), a deadlock situation incurs while reading channel **b**. Since process B has not been yet executed and there is not enough area to load B on the reconfigurable hardware, there is no data written in the channel **b**.

By adding token channels, as shown in Figure 4.5(b), we ensure that all the processes are configured only when all the input data are available for them. These token channels are ordinary Kahn channels that are used to read and write tokens. As a result, each process reads the token channel before reading

[2]The event sequences are read in the figure in the bottom up manner.

any data channel and writes onto the token channel after writing all the data channels. For example in Figure 4.5(b), A and B write token channels, a_t and b_t respectively, after writing data to their respective data channel **a** and **b**. In this way, C can only proceed after reading tokens from A and B, which guarantees that the data is already written by A and B. This enforces an implicit scheduling between the tasks, forcing task A and B to be executed before C, and hence, the aforementioned deadlock situation can be avoided.

The application model can be easily extended to accommodate the token channels. As mentioned earlier, in the Sesame framework, Kahn processes typically consists of a loop that first reads application data, performs execution and finally writes output. In this case, Kahn processes can easily be extended to produce token channels to read/write an extra token channel at the end of each iteration.

4.2.2.3 Resource Management Strategies

To manage architectural resources on the reconfigurable hardware at runtime, the RM can employ a variety of algorithms. The identification of "optimal" mapping and scheduling strategies for reconfigurable architectures is a well known NP-hard problem [127]. Therefore, many different heuristics have been proposed to address task mapping and scheduling on the reconfigurable hardware when the complexity of the problem does not find a solution in a feasible time [45, 46, 48, 49]. Such heuristics, however, need access to the global state of the system status, in order to perform efficient mapping. The RM can keep track of such information as required by the system for making mapping decision. One of the main goals of the RM is to utilize hardware resources in an efficient manner. Other goals include the minimization of the scheduling time, the reduction of the waiting time and the minimization of the communication between two tasks on the hardware. The choice of such an algorithm can be made based on the design objective. Based on the requirements, any kind of task placement and scheduling algorithm for the reconfigurable hardware can be easily implemented in the model as a plug-in to the RM.

4.2.2.4 Virtual Processors Extensions

In the Sesame framework, the VPs forwards the events (R, X and W), from the application model to the corresponding architectural components, as soon as their dependencies are met. To avoid any deadlock that can occur due to the reconfigurable nature of RUs, as mentioned before, Kahn processes in the application layer, are extended to support an additional token channel. To accommodate such changes in the application layer, extensions are implemented also for VPs in the mapping layer. Towards this end, the VPs are extended in such a way that a VP first checks the availability of all its input data by checking a special token channel for all of its inputs. When all data is available, a VP proceeds as normal and it forwards all R, W and X events to the corresponding architectural components. For instance, X events are forwarded either to the GPP, or to the RU, and R and W events are forwarded to the corresponding memory component. Similarly, X_p events are forwarded the corresponding RUs. After completing this process, a VP writes a token to all of its output token channels to notify all subsequent nodes about the data availability. Since VPs do not have any knowledge about the structure of an application, they cannot autonomously determine when all inputs are available or when to notify about the output availability to other nodes. The reading and writing of token channels is, therefore, managed explicitly by the application model, and the events created by the special reads (R_T) and writes (W_T) are used by the VPs, in order to perform the extra synchronization in the timed simulation domain. Note that, these synchronization events (R_T and W_T) are not forwarded to the architecture model. As a result, only the timing consequences of normal R, W or X events are modeled in the architecture. Hence, the simulation result is not affected by extra events.

4.2.2.5 Self-Schedulable Property Preservation

The modifications to the application model essentially allow the mapping layer to dynamically determine a valid and deadlock-free schedule for application events, which is needed to successfully drive the underlying reconfigurable architecture model. The architecture and mapping models in the Sesame framework do not contain any structural information about the application. They just receive and process events (R, W and X). Therefore, as discussed before, the application model is changed in such a way that the mapping layer can send special tokens to the architecture to indicate when *all* inputs are available. This

provides an indication for a particular RU to be configured when sufficient resources are available. The subsequent events (R, W and X) that arrive at the RU component are part of the atomic execution block, which are annotated with the special event (X_p). The X_p is sent to the RU indicating that it can be un-configured, and its resources on the hardware can be released to be used by another RU.

This approach is relatively easy to implement because the functionality of the architecture layer is rather simple, as the synchronization is performed in the mapping layer. The architecture layer, for instance, *only* keeps track of the available area, and it maintains a list of RUs that are waiting until resources are available. Furthermore, the model is self-schedulable, meaning that there is no need to provide an external scheduler and the self-scheduling automatically works for all mappings. Furthermore, this self-schedulability is also required when runtime change in mapping has to be supported, which is discussed in the next section.

The disadvantage of this approach, however, is that it requires modifications of the application model. These modifications may limit the class of KPNs that can be run, because not all KPNs can be easily extended with the required token channels. For instance, KPNs with highly irregular communication pattern or KPNs with cyclic communication patterns may not be easily extended based on the loop iteration as explained before. In these cases, the Kahn application may require to be rewritten by merging certain processes.

Note that, this behavior of the reconfigurable architecture can also be implemented with the Sesame framework in many other ways. We choose this approach for its simplicity. The choice of the approach depends on the trade-off between low modeling effort and the flexibility of the model. It is up to the designer to decide where and how to implement such functionalities.

4.3 Runtime Application Mapping Modeling

In the previous section, we explained about the extension of the Sesame framework to model dynamically reconfigurable architectures in general. In this section, the extension of the Sesame framework to model the runtime application mapping of the reconfigurable architecture is described in detail. With runtime mapping, changes in the application mapping are allowed at runtime, in order

to accommodate any change either in the application, or in the architecture, or in the environment. The task can change its mapping from one architectural resource to another. In this particular case, the task can change its mapping from the GPP to the RU or vice versa at runtime. The detailed description about runtime application mapping has been presented in Section 3.4.

In order to allow the modeling of the application mapping at runtime, an additional component, called Runtime Mapping Manager (RMM), has to be modeled, which can perform the mapping decision at runtime. The detailed discussion of a typical RMM has been presented in Section 3.6. Before discussing on how to model the behavior of a typical RMM with the Sesame framework in Section 4.3.2, in the following, we first talk about the runtime mapping behavior.

4.3.1 Runtime Mapping Behavior

The modeling of a system that allows runtime mapping of tasks to processing resources can be divided into two parts: the *spatial* mapping behavior and the *temporal* mapping behavior. *Spatial* mapping is the process of identifying which part of the application can be implemented on the reconfigurable hardware (HW tasks), and which part should be executed as software (SW tasks). Not all HW tasks may fit on the reconfigurable device at the same time. Therefore, a logical *configuration* has to be defined for a set of HW tasks for which the functional logic has been loaded on the reconfigurable device at a given moment. *Temporal* mapping is the process of determining a sequence of these configurations at runtime, such that all HW tasks can run efficiently. In summary, spatial mapping determines *where* to map a task and temporal mapping determines *when* to map a task.

With the rSesame framework, a task can be modeled either as a HW task or as a SW task. A task modelled as HW is always mapped onto the hardware component of the architecture and a task modelled as SW is always mapped onto its software counterpart. Task assignment for the SW and HW categories is done a priori. At runtime, these tasks are mapped onto their corresponding resources based on time, architectural resources and conditions of the system. If more than one task is mapped onto the reconfigurable hardware, and if all of them do not fit on the hardware at once, they are divided into logical configurations and executed sequentially. In this way, the temporal mapping is addressed at runtime with the rSesame framework.

In order to model the spatial mapping behavior at runtime, the framework supports a third task type: the *pageable* task. Unlike HW and SW tasks, a task modelled as pageable does not have a fixed spatial mapping. Basically, a pageable task can be either a HW or a SW task and it can be mapped onto any of the computing resources. For pageable tasks, the spatial mapping decision is made at runtime. These tasks are mapped either as HW or SW depending on the runtime conditions of the system (e.g. resource availability).

For HW and SW tasks, only the decision of *when* to map is made at runtime, and for the pageable tasks the decisions of *when* and *where* to map are made at runtime. Note that, by eliminating pageable tasks, only the temporal mapping behavior is supported at runtime, while with pageable tasks, both temporal and spatial mapping are addressed at runtime. In other words, without pageable tasks, only the temporal mapping behavior takes place at runtime, whereas with pageable tasks, both temporal and spatial mapping take place at runtime. Designers can perform various explorations with these task sets depending on the system requirements and their evaluation purpose, such as static exploration and runtime exploration. In this way, rSesame can be used to model the task mapping both statically and at runtime, and to realize the two-level mapping exploration as discussed in Section 3.5. To perform such mapping decisions at runtime, the rSesame framework rely on the RMM component. In the following, we discuss the modeling of the behavior of the RMM with Sesame.

4.3.2 Modeling of the Runtime Mapping Manager

The RMM is the central intelligent component, which performs runtime mapping decisions. The complexity of RMM implementations can vary from relatively simple to extremely complicated. As explained in the case of the RM, in the actual system implementation, the RMM entity may be a part of the application, middleware, and operating system or even implemented as a hardware component. In this particular implementation, we model the case when the RMM is placed between the application and the architecture. Therefore, the RMM is modeled as a mapping layer component that resides between the VP and the RM. VPs can be considered as a distributed form of the application manager since they bring application information (e.g. priority of tasks, real time constraints) to the RMM. The RM is a component in the architecture layer, which brings the architectural information (e.g. resources, timing

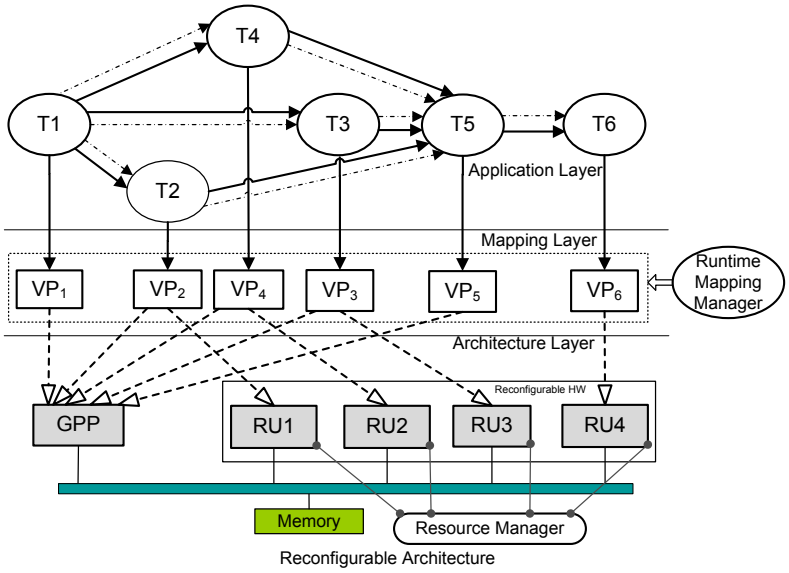


Figure 4.6: The three layers in the Sesame infrastructure for modeling runtime application mapping on reconfigurable architectures. The Virtual Processors (VPs) are extended to connect to the GPP and to the RU.

information) to the RMM. The RMM uses this information as inputs for its decision making. In order to make any mapping decisions, the RMM employs any available method or heuristic to perform the mapping decision. The RMM also bears the ability to learn from its environment, from its previous data, or from the current situation. Based on these values and behaviors, the RMM can make mapping decisions for each task using the current application behavior, the past execution or even the predicted dynamic conditions of the system. Since the runtime mapping policy is implemented as a separate component, it is easy to plug-in diverse range of RMM policies for different experiments.

In order to facilitate runtime application mapping with the Sesame framework, the VPs can be connected to multiple processor resources. The trace events from the application model can now be forwarded to any connected processor in the hardware layer. Figure 4.6 shows an example where a virtual processor VP_2 is connected to the GPP and RU1. Hardware or a software task requires only one connection to a RU or GPP respectively. However, a pageable task has a connection to both processors. In many scenarios, it is conceivable that there are multiple RU implementations for one pageable task. However, for the sake of simplicity, we currently restrict it to one. Moreover,

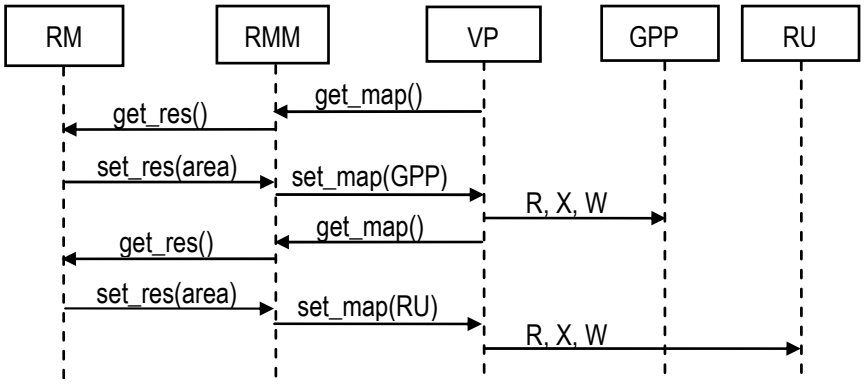


Figure 4.7: The sequence diagram showing an interaction between the Resource Manager (RM), the Runtime Mapping Manager (RMM), the application Virtual Processor (VP) and the GPP/RU to enable runtime application mapping. The RMM resides between the Resource Manager (RM) and the application VP, which performs mapping decisions at runtime based on the policy implemented.

in many heterogeneous architectures that include more than two types of processors, this connection can be established with all those processors. The VP forwards events, but does not make the mapping decision by itself. This is done by the RMM to which all VPs are connected. Before forwarding events from the application to the architecture, a VP asks the RMM on which processor to execute the event. Based on the policy implemented, the RMM returns a target processor identifier (either a RU or the GPP) and the VP forwards events accordingly.

Components Interaction

To summarize the interaction between different components (such as the RMM, the RM, the VP, the GPP and RUs), let us consider Figure 4.7. It shows an example of time-interaction diagram of the components that are involved in processing application trace events for a GPP and a RU. As soon as the VP receives an event to forward it to the architecture layer, it requests the RMM to obtain a mapping for that event (`get_map()`). Before making any mapping decision, the RMM checks the resource availability in the architecture layer by contacting the RM (`get_res()`). The RM provides the necessary architecture information to the RMM (`set_res(area)`). If there are available resources on the reconfigurable hardware, the RMM maps the task onto the RU

(`set_map(RU)`) otherwise to the GPP (`set_map(GPP)`). The RMM may request additional information about the system or the environment. Based on the policy implemented, the available resources and the system conditions, the RMM decides which event to forward to which processor component. The VP gets this decision in form of a target process identifier (either RU or GPP) and the VP forwards the event accordingly. Note that, the delay for these components can also be modeled in the system. Such delays can be easily provided as a parameter in each of these components.

4.4 The rSesame Framework Characteristics

The Sesame framework is extended to model and simulate reconfigurable architectures. These extensions are discussed in Section 4.2.2 and Section 4.3.2. Basically, with these extensions, the framework can realize the two-level mapping explorations of reconfigurable architectures as discussed in Section 3.5. As it can be observed from Figure 4.6, without the implementation of the RMM, the framework can be used to perform static exploration of reconfigurable architectures. With the implementation of an application mapping policy in the RMM, it can be used to evaluate the application mapping at runtime. In this way, the framework can be exploited to perform both the static and the runtime mapping explorations. Such a framework strives for various characteristics, such as a the one described in the following.

- **Generic Framework:** The rSesame is a generic framework in the sense that it is not restricted to the modeling of one type or class of reconfigurable systems. Instead, it can be deployed to model and evaluate any kind of reconfigurable architecture running a wide set of streaming applications from the multimedia domain. Using the rSesame framework, a designer can instantiate a model for the given architecture and any additional architectural specifics can be augmented in the model as required. Figure 4.8 presents a scenario where several models are instantiated from the rSesame framework for different types of reconfigurable architectures running different sets of applications. In the next chapter, we discuss an example of one such instance for a specific architecture running a particular application set.

- **Flexibility by Modular Design:** In the presented model, an application model is independent of the architectural specifics. As a result,

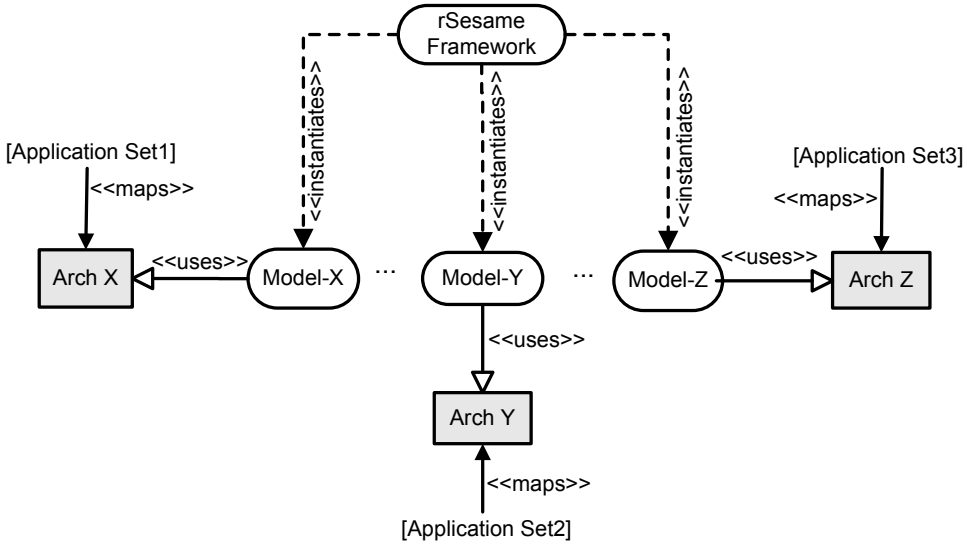


Figure 4.8: Instantiation of the rSesame framework for various architectures. Models (Model-X, Model-Y, ...Model-Z) are instantiated from the framework for different architectures (Arch-X, Arch-Y, ...Arch-Z) running different set of applications.

application and/or architecture models can be re-used and altered without affecting each other. This separation of concerns makes easier to accommodate any kind of modification to the model, permitting design variations and even completely different architectures to be modeled with ease. Moreover, a well defined structure of the RM and the RMM makes the model flexible. Therefore, any kind of mapping policy can be plugged into the model and evaluated without affecting other components.

- **Performance by abstraction:** The applications are modeled at the granularity of tasks where the application behavior is abstracted as read (R), write (W) and execute (X) events. The model operates based on discrete-event simulation of these events leaving out all the minute details that might, otherwise, hinder the model's performance. Therefore, we provide easy construction of the models and fast simulation. There is always a tradeoff between detailed modeling and fast model performance and, in this case, we compromise details for performance. As a result, the accuracy of the resulted model is also compromised. This is a fair tradeoff for a system-level model, which is targeted at very early design stages. At this level, where the design space is extremely large,

a quick exploration is more important than a detailed and very accurate exploration.

- **Ease of Use:** The modular design and the higher abstraction level together facilitate the ease of use of the model. In case of application modeling, the Kahn application models can be automatically converted from sequential C/C++ code using tools, such as the one presented in [117]. Similarly, any kind of architecture model can be constructed from generic building blocks provided by a library, which contains templates for processors, memories, buses, on-chip networks and so on. Moreover, any policy can be implemented with a minimal effort and without having detailed knowledge of all parts of the model. Thus, the learning curve is rather moderate.

- **Input/Output:** The input given to the model consists of different architecture parameters, such as hardware and software latencies, area and reconfiguration delay. The model simulates the application characteristics, architecture responses and the runtime spatial and temporal mapping behavior. As a result, it produces various system evaluation attributes, such as performance, speedup, number of reconfigurations, mapping behavior of a task, area utilization, number of HW and SW tasks, percentage of hardware/software execution, percentage of reconfiguration and hardware reusability efficiency.

4.5 Conclusions

In this chapter, we presented the rSesame framework. rSesame is a generic system-level modeling and simulation framework which can explore and evaluate reconfigurable systems both statically and at runtime. The main features of the rSesame framework include flexibility, ease of use, fast performance, and applicability. The rSesame employs the Sesame modeling and simulation framework as a system-level simulation platform. The Sesame framework allows efficient system-level performance evaluation and exploration of heterogeneous embedded multimedia architectures. We extented the Sesame framework to perform the modelling and simulatation of recofigurable architectures at runtime. To model reconfigurable architectures with the Sesame framework, the following extensions have been performed:

- an X_p event has been added to steer configuration of tasks onto the reconfigurable hardware, and scheduling of RUs;
- a Resource Manager (RM) is employed to manage reconfigurable resources at runtime;
- token channels haven been added to preserve self-schedulability and deadlock-free properties of the model.

Additionally, the chapter presented the discussion on modeling reconfigurable architecture at runtime with the Sesame framework. In order to facilitate the runtime change in the application mapping with Sesame, the mapping layer of the framework is extended by adding the RMM component, which allows changing in the application mapping at runtime by employing diverse mapping policies.

In the next chapter, we present a case study, where the rSesame framework is deployed to perform the system-level DSE of a reconfigurable architecture. In particular, the rSesame framework is used to perform rapid evaluation of several application-to-architecture mappings of a reconfigurable architecture, both statically as well as at runtime. A model is instantiated from the rSesame framework for an existing reconfigurable architecture, and the instantiated model is used to show how the framework can be used to perform such explorations.

Note. The content of this chapter is based on the following articles:

K. Sigdel, M. Thompson, C. Galuzzi, A.D. Pimentel, K.L.M. Bertels, **rSesame - A Generic System-Level Runtime Simulation Framework for Reconfigurable Architectures**, *Proceedings of the International Conference on Field-Programmable Technology (FPT'09)*, Sydney, Australia, December 2009, pp. 460-464.

K. Sigdel, M. Thompson, A.D. Pimentel, T. P. Stefanov, K.L.M. Bertels, **System-Level Design Space Exploration of Dynamic Reconfigurable Architectures**, *Proceedings of the International Symposium on Systems, Architectures, MOdeling and Simulation (SAMOS'08)*, Samos, Greece, July 2008, pp. 279-288.

5

Molen Architecture : A Case Study

The rSesame framework presented in the previous chapter can be deployed for performing system-level Design Space Exploration (DSE) of reconfigurable architectures by evaluating several parameters, such as architectural characteristics, hardware-software partitioning, and scheduling strategies, both statically as well as at runtime. The main goal of this chapter is to introduce a case study to show the characteristics of the rSesame framework tested on an existing and well known reconfigurable architecture by performing system-level DSE based on various design attributes. The aim of this case study is twofold. First, we instantiate a model from the rSesame framework for the Molen reconfigurable architecture [22, 23]. The Molen reconfigurable architecture is established based on the Molen architectural paradigm incorporating a microprocessor and a reconfigurable processor. The instantiated model is used to demonstrate how the rSesame framework can be deployed to perform two-level mapping exploration (static and runtime mapping exploration) of the Molen architecture. Second, we show that the model can be efficiently used to perform exploration of various design parameters such as execution time, area usage, number of reconfigurations and percentage of hardware and software execution.

The remainder of the chapter is organized as follows. Section 5.1 presents an overview of the Molen reconfigurable architecture, which is used as a test bench to perform the proposed case study. Section 5.2 deals with the model instantiation from the rSesame framework for the Molen reconfigurable architecture. Section 5.3 presents the application model considered for the case study together with the necessary experimental setup required. In Section 5.4, the instantiated model is used to perform mapping exploration of the given application onto the Molen reconfigurable architecture both statically as well as

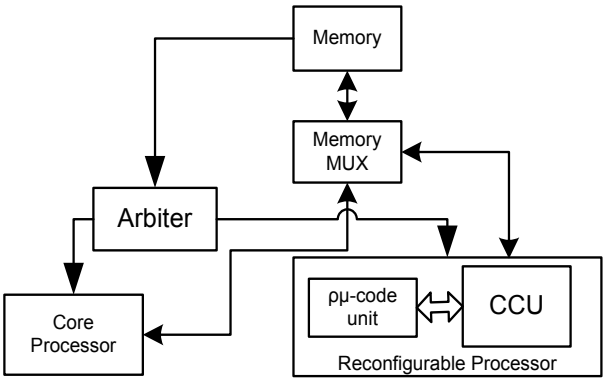


Figure 5.1: The machine organization of the Molen reconfigurable architecture. The architecture consists of a General Purpose Processor (GPP) and a Reconfigurable Processor (RP), which are coordinated by an arbiter.

at runtime. In Section 5.5, we demonstrate that the rSesame framework can be efficiently deployed to explore various design attributes. Section 5.6 presents the detailed analysis and the comparison of the results obtained from the static exploration and the runtime exploration. Finally, Section 5.7 summarizes the major contributions and concludes the chapter.

5.1 Molen Architecture

The Molen polymorphic processor is established on the basis of the tightly coupled co-processor architectural paradigm [22, 23]. It consists of two different kinds of processors: the core processor, which is a General Purpose Processor (GPP), and the Reconfigurable Processor (RP). Figure 5.1 depicts the machine organization of the Molen reconfigurable architecture. The reconfigurable processor is further subdivided into the reconfigurable microcode ($\rho\mu$-code) unit and *Custom Computing Unit* (CCU). The CCU is executed on reconfigurable hardware, e.g., an FPGA, and is intended to support additional functionalities, which are not implemented in the core processor. In order to speed up the program execution, parts of the code running on a GPP can be implemented on one or more CCUs.

The GPP and the RP are connected to an arbiter. The arbiter controls the co-ordination of the GPP and the RP by directing instructions to either of

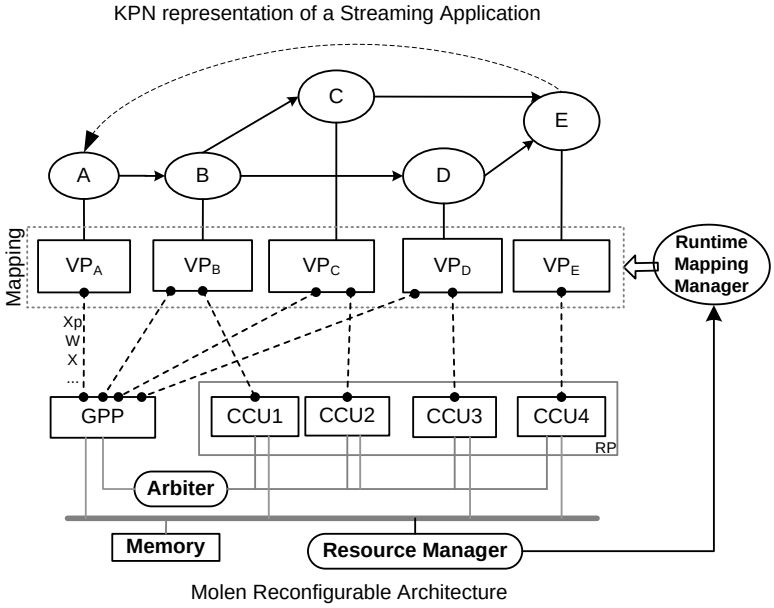


Figure 5.2: A model instantiated from the rSesame framework that can facilitate static and runtime exploration of the Molen reconfigurable architecture. The arbiter is modeled as a part of the architecture layer, which co-ordinates the GPP and CCUs.

these processors. The code to be mapped onto the RP is annotated with special `pragma` directives. When the arbiter receives the `pragma` instruction for the RP, it initiates an "enable reconfigurable operation" signal to the reconfigurable unit, gives the data memory control to the RP and it drives the GPP into a waiting state. When the arbiter receives an "end of reconfigurable operation" signal, it releases the data memory control back to the GPP and the GPP can resume its execution. An operation executed by the RP, is divided into two distinct phases: *set* and *execute*. In the *set* phase, the CCU is configured to perform the supported operations and in the *execute* phase the actual execution of the operation is performed. The decoupling of `set` and `execute` phase, allows the set phase to be scheduled well ahead of the execute phase and thereby hiding the reconfiguration latency.

5.2 Model Instantiation for the Molen Architecture

In order to perform exploration of different application-to-architecture mappings onto the Molen architecture by using the rSesame framework, we instantiate a *Molen model* using the rSesame framework [1]. This model is depicted in Figure 5.2, which consists of the Molen architecture. In the Molen architecture, the task to be accelerated by executing with the reconfigurable hardware is implemented in the CCU. Therefore, there can be as many CCUs as the number of tasks mapped onto hardware. As a result, the Molen architecture depicted in Figure 5.2 consists of more than one CCU. In the remainder of this section, the modeling of the different components of the Molen architecture will be discussed.

Figure 5.2 depicts that in the Molen model, the CCUs and the GPP are modeled as architectural layer components. The GPP is a core processor, thus, is simply instantiated from rSesame's processor class. A CCU represents a RP's custom computing unit, and it bears the ability to execute task the same way the processor does. As a consequence, a CCU is also instantiated from the processor component. In addition to the normal processor class, a CCU is also provided with additional parameters, such as area occupancy and reconfiguration delay. The pragma directive is modeled as a special execute event (X_{pragma}) in the application layer. Based on the special event X_{pragma}, a CCU can be configured/unconfigured on the RP. The X_{pragma} event corresponds to the X_p event defined in Section 4.2.

The Molen architecture exhibits a tightly coupled co-processor paradigm, and it requires CCUs to run as a co-processor, which adds control dependencies between the GPP and CCUs. In some cases, this added dependencies can lead to a deadlock situation in the architecture model. To avoid this deadlock, we have restricted the KPNs in the application layer to be static. Additionally, to make sure only safe events are forwarded from the mapping layer to the architecture models, we modified the application by adding a Kahn channel from the application's output (or sink) node to its source node(s). This means, for a streaming application, such as depicted in Figure 5.2, after node A has written data to node B, the former has to wait for the token from the sink-node E, before it can write a new data item to the stream. To achieve this, sink and source nodes have been slightly adapted to read and write the

[1] Now onwards we address the Molen reconfigurable architecture simply as the Molen architecture

additional token. This way the pipeline parallelism is removed from the application, and avoids two data-dependent tasks to be active simultaneously in the architecture model. This extension is necessary to prevent the deadlock situation in the model that can occur due to the co-processor behavior of the Molen architecture. It is important to note that the sink-to-source channel does not remove all the parallelism in the application. Particularly, "fork-and-join" parallelism still remains available between tasks that are not data dependent, such as between the task pair (C,D).

5.2.1 Modeling the Arbiter

The arbiter has been modeled as a component in the architecture layer which controls the execution of the GPP and CCUs (see Figure 5.2). The arbiter coordinates the co-processor behavior of the Molen architecture by granting exclusive control to either the GPP or the CCUs. In order to model this behavior, the arbiter implements a synchronization primitive, which allows to execute instructions for either the GPP or the RP. In the following sections we describe the GPP/CCUs synchronization mechanism to model the co-processor behavior of the Molen architecture.

As mentioned before, the `pragma` annotations that mark the hardware implementation of tasks have been modeled as a special execution event (X_{pragma}) in the application layer. Together with other events, such as R, X and W, the X_{pragma} event is also passed to the architecture model. When a processor (GPP or CCU) receives the X_{pragma}, it requests for the execution control from the arbiter. The arbiter coordinates between these processors by granting control only to one processor (either GPP or CCU) at a time.

Figure 5.3 illustrates the interaction between the GPP, CCUs and the arbiter. The figure shows these interactions in the case where GPP and CCUs want to execute at the same time. In this particular case, GPP gets the lock to execute at t0. At time t1, CCU1 also requests for an execution. Since the GPP is still executing, CCU1 goes to a wait mode. When the GPP finishes its execution, it returns the lock at time t2 and execution is granted to CCU1. At time t3, CCU2 requests execution. Since both CCU1 and CCU2 operate in parallel on the FPGA, CCU2 also gets the lock and can start execution. At time t4, CCU1 finishes its execution, but CCU2 is still executing on the reconfigurable hardware and only finishes at time t5. At time t4, if the GPP is to request the lock for execution, then it has to wait until time t5. In this way, the

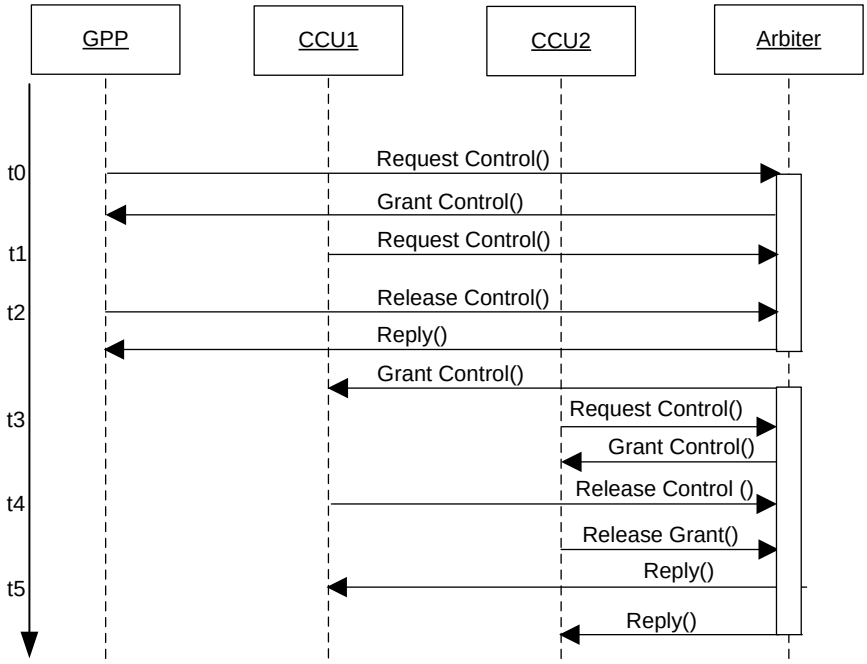


Figure 5.3: The sequence diagram showing an interaction between the GPP, CCUs and the arbiter. The arbiter co-ordinates between the GPP and CCUs by granting the execution control.

arbiter guarantees that all the CCUs finish their execution before it releases the lock to the GPP. This is the interaction between the GPP and CCUs when they behave in mutual exclusive way. In this case, the arbiter acts as a mutex. The model can be easily extended to support the parallel operation of the GPP and the CCUs. In such cases, the arbiter grants control also to CCU1 and CCU2 together with the GPP. The control is granted whenever it is asked for following simple scheduling algorithms, such as first-come-first-serve. Depending on the behavior of the architectural components, the functionality of the arbiter can be modified and additional functionalities can be easily included in the arbiter.

5.2.2 The Runtime Mapping Manager

The Runtime Mapping Manager (RMM) is the central intelligent component of the rSesame framework, which performs runtime mapping decisions. It is modeled as a mapping layer component, and resides between the applications

Virtual Processors (VPs) and the Resource Manager (RM). As a result, RMM can receive both the application and the architectural information. In the actual system implementation, the RMM entity may be a part of the application, middle-ware, operating system or even implemented as a hardware component. The RMM can employ various strategies for performing runtime task mapping decisions.

The RM that is a part of the architecture layer in the rSesame framework, provides co-ordination between various reconfigurable units. In the Molen model, the RM co-ordinates between the various CCUs. The RM also keeps track of the information on the available resources (i.e. available area) in the underlying Molen architecture, by monitoring which CCUs are *configured* on the RP at any given time. The RM, in this case, is also responsible for configuring and un-configuring CCUs for executing them on the RP.

In order to perform the aforementioned actions, the RM implements two modules for mapping and un-mapping CCUs onto the RP. A CCU invokes the `mapping()` module when it initiates its execution on the RP, and it invokes the `unmapping()` module when it completes its execution. The pseudo-code for each module is specified in Algorithm 5.1 and Algorithm 5.2 respectively. The resource management on the RP is implemented such that, a CCU can preserve its hardware configuration on the RP, until it is overwritten by another CCU. The rest of this section discusses the resource management of the RM in more detail.

We define three states for a given task: a `WAITING` state, a `MAPPED` state, and a `RUNNING` state. A task is in the `WAITING` state, if it is waiting to be mapped onto the RP. Thus, a task in `WAITING` state does not occupy any hardware resources. A task is in the `MAPPED` state if it is already configured on the RP, but it is not being executed. The task may be re-executed later. The configuration of such a task is saved on the RP, thus it occupies hardware resources. When the task in `MAPPED` state is required to be re-executed, it can directly start processing without reconfiguration. Finally, a task is in the `RUNNING` state, when the task is actually processing data.

Algorithm 5.1 presents the pseudo-code that describes the functionality of the `mapping()` module for task T_i. CCU_i represents the hardware implementation of task T_i. Therefore, Algorithm 5.1 is explained in terms of the CCU_i. If the total available resources [2] of the RP is not enough to accommo-

[2] In this case study, we use area as a factor to measure the amount of resources consumed by

Algorithm 5.1 Pseudo-code for the mapping method

Require: CCU_i
1: **if** $CCU_i \geq$ Total_AREA in RP **then**
2: CCU_i mapped onto GPP.
3: **else**
4: {*Task already mapped on the RP, reuse it.*}
5: **if** CCU_i == *MAPPED* **then**
6: CCU_i . *STATE* $\leftarrow$ *RUNNING*;
7: **else**
8: {*CCU not mapped on the RP, configure the CCU.*}
9: **if** *AREA* $\geq$ CCU_i . *AREA* **then**
10: configure(CCU_i);
11: CCU_i . *STATE* $\leftarrow$ *RUNNING*;
12: **else**
13: $j \leftarrow 0$
14: {*Not enough area, remove the mapped CCUs.*}
15: **while** *AREA* $\leq$ CCU_j . *AREA* and $j < N$ **do**
16: **if** CCU_j == *MAPPED* **then**
17: $CCU_j \leftarrow$ *WAITING*;
18: *AREA* = *AREA* + CCU_j . *AREA*;
19: **end if**
20: **end while**
21: {*Enough area, configure the CCU.*}
22: **if** CCU_i . *AREA* $\leq$ *AREA* **then**
23: configure(CCU_i);
24: CCU_i . *STATE* $\leftarrow$ *RUNNING*;
25: **else**
26: {*Not enough area, store the CCU in CCUList.*}
27: Store(CCU_i , *CCUList*);
28: **end if**
29: **end if**
30: **end if**
31: **end if**

date any CCU, the latter is mapped onto the GPP (line 1 to 3 in Algorithm 5.1). Similarly, if a CCU is already mapped on the RP, the CCU can start processing data without its configuration (line 5 to 7 in Algorithm 5.1). However, if the

the CCUs on the RP.

Algorithm 5.2 Pseudo-code for the un-mapping module

```
Require: CCU_i
1: {Save the configuration for next time.}
2: CCU_i.STATE ← MAPPED;
3: while CCUList != empty do
4:    CCU_k ← FirstFit(CCUList);
5:    {Configure the CCU that fits in the available area.}
6:    if AREA ≥ CCU_k.AREA then
7:       configure(CCU_k);
8:       CCU_k.STATE ← RUNNING;
9:    end if
10: end while
```

configuration of a given CCU is not currently present on the RP, and there is sufficient area to execute that CCU, then the given CCU is mapped on the RP. In this case, it is necessary to configure the CCU, and it can only start processing the data after its configuration (lines 8 to 11 in Algorithm 5.1). Finally, if the RP does not have an adequate amount of area to execute the given CCU, a set of CCUs out of total N CCUs in RP, whose area is sufficient to accommodate the given CCU, is overwritten by the current CCU. Note that, the CCUs that are currently running cannot be overwritten (line 13 to 20 in Algorithm 5.1). In case, there is not enough area accommodated by overwriting the previously mapped CCUs on the RP, the current CCU is stored to be configured later when area becomes available (lines 22 to 27 in Algorithm 5.1).

In the `unmapping()` module, as depicted in Algorithm 5.2, a CCU is unmapped from the RP. When the `unmapping()` module is called, the state of a CCU_i is changed to MAPPED. This implies that the hardware configuration for that CCU is saved on the RP, and if the CCU is re-executed, there is no need for its reconfiguration (line 1 to 2 in Algorithm 5.2). At the end of the `unmapping()` module, a set of CCUs from the waiting list is evoked to be executed on the RP. The policy implemented to select these CCUs is rather simple. The first CCU from the waiting list that fits on the currently available hardware area is configured first (line 3 to 9 in Algorithm 5.2).

Typically, the `mapping` and the `un-mapping` modules are called in a subsequent order. When a CCU requires to process its execution traces, the `mapping` module is invoked, and when the X_{pragma} directive is encountered, the corresponding `un-mapping` module is invoked. The X_{pragma} directive in

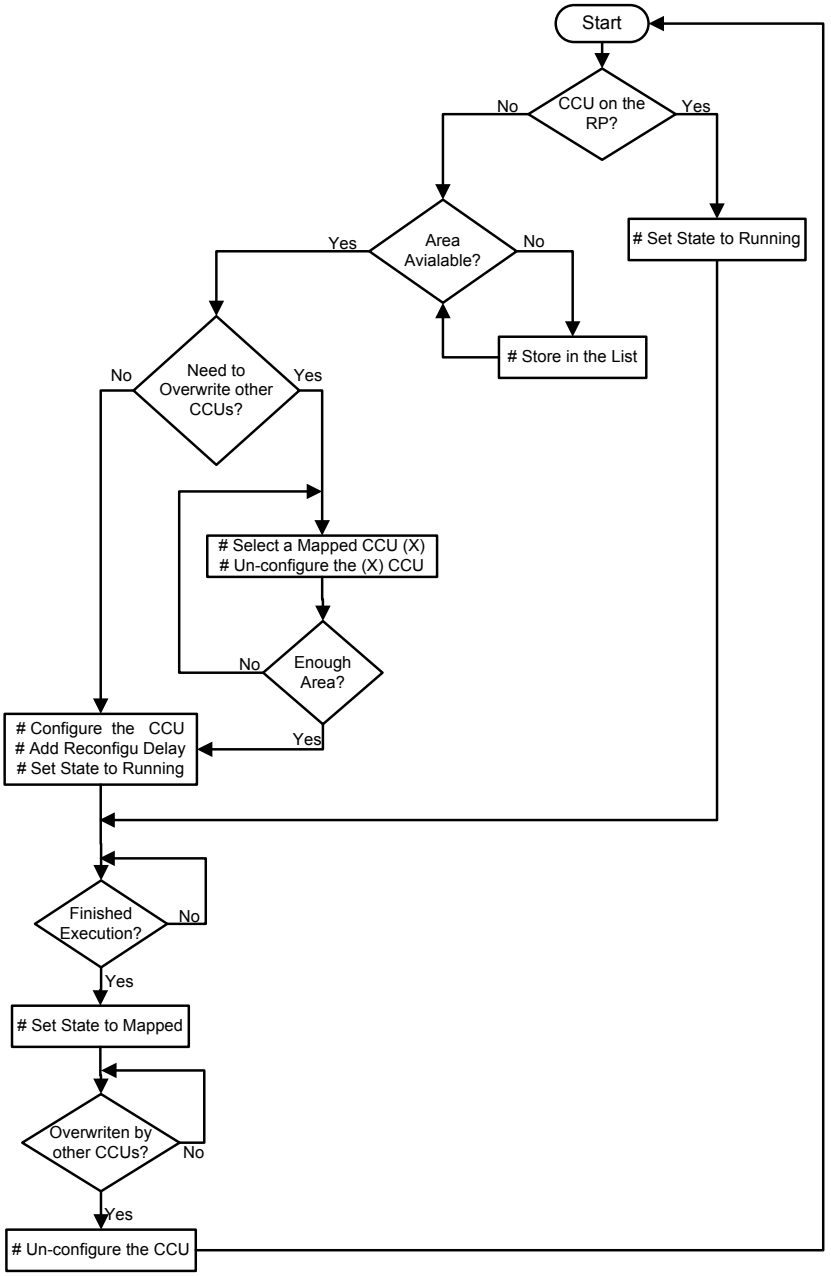


Figure 5.4: The flowchart showing the mapping/un-mapping of a CCU onto the reconfigurable processor (RP). If the CCU is already mapped, the configuration is reused, otherwise, the CCU is configured.

a typical execution trace is shown in Figure 5.2. The complete phase involved in mapping and un-mapping CCUs onto the RP is depicted in the flowchart presented in Figure 5.4. The flowchart illustrates that the configurations of the CCUs already mapped on the RP, are re-used. A CCU is configured on the RP when sufficient area is available, and its state is set to `MAPPED` when its execution is completed. This way, the configuration of a CCU is saved for future re-use. When another CCU executes on the RP, if additional area is required, all the CCUs with `MAPPED` state can be overwritten to accommodate that CCU.

5.3 Experimental Setup

We consider a Motion-JPEG (MJPEG) encoder application as a case study. The corresponding KPN is shown in Figure 5.5. The application model consists of two implementations (MJPEG1 and MJPEG2) of the MJPEG application. MJPEG1 operates on the pixel blocks (partially) in parallel (see the 4 DCT/Q streams in the upper part of Figure 5.5), whereas MJPEG2 operates on the blocks sequentially (lower part of Figure 5.5). MJPEG1 and MJPEG2 are combined together in order to create an example of a dynamic application. MJPEG2 can be considered as a sporadic application that appears in the system randomly and competes with MJPEG1 for the resources. This behavior is implemented in such a way that, at a certain point in time, MJPEG2 starts encoding a frame simultaneously with MJPEG1. The inputs for MJPEG1 and MJPEG2 consists of 8 and 4 picture frames of 128×128 pixels respectively.

To demonstrate the mapping exploration at runtime with the Molen model, we incorporated a simple strategy, called As Much as Possible (AMAP), to perform the runtime mapping decision. This strategy is implemented as a task mapping policy for the RMM. With the implemented policy, the area is considered as the only factor for performing task mapping decisions by the RMM. AMAP tries to maximize the use of the RP area as much as possible. As a result, tasks are executed to the RP if sufficient area is available, otherwise they are executed to the GPP. This straightforward policy can be used as a simple resource management strategy in various domains. This is also used as a default runtime mapping strategy in the rSesame framework.

We instantiated a Molen model with 18 CCUs, one for each task. This allows us to make the most suitable use of the parallelism available in the

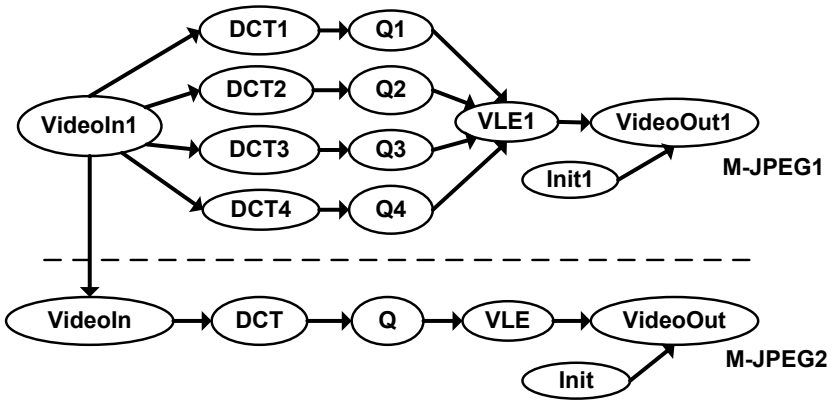


Figure 5.5: The application model used for the case study which combines two implementations of Motion-JPEG (MJPEG) together.

application, by mapping each of the DCT and Q tasks onto a CCU. A CCU represents an implementation of a Kahn process. Hence, the one-to-one mapping of the Kahn process onto the CCU represents the correct behavior of the CCU in the Molen architecture. Note that, the number of CCUs is a parameter that a designer can define based on the number of pageable and HW tasks. For this case study, we consider all tasks as pageable to fully exploit the runtime mapping by deciding *where* and *when* to map them at runtime depending on the system condition. We also assume that no task can have a size larger than the total RP area. Additionally, all CCUs may not fit on the RP at once due to its area constraints. Nonetheless, they can execute on the RP after the reconfiguration.

The main purpose of this case study is to demonstrate that the rSesame framework can be used to perform both static and runtime mapping decisions. As a result, we use estimated values of the computational latency, the area occupancy (on the RP) and the reconfiguration delay for each CCU. The computational latency values that the GPP model component associates with the computational events, are initialized using estimates obtained from literature [2, 126] (but non-Molen specific). We estimated area occupancy for each process mapped onto the CCU using the Quipu model [120]. Quipu predicts FPGA resources from a C-level description of an application using Partial Least Squares Regression (PLSR) and Software Complexity Metrics (SCM). Kahn processes contain functional C-code together with annotations that generate events such as R, X and W. As a result, Quipu can estimate area occupancy of each Kahn process.

For this case study, we considered a Xilinx Virtex XC4VFX60 [36]-based implementation of the Molen architecture. We assume the Processor Local Bus (PLB) of the FPGA is 4 bytes wide, and the Internal Configuration Access Port (ICAP) functions at 100 MHz, thus, its configuration speed is considered at 400 MB/sec [128]. Based on the reconfiguration delay of the considered FPGA, and the estimated area occupancy of each CCU, we computed the reconfiguration delay of each CCU using the following equation:

$$T_{Recon} = \frac{\text{CCU slices}}{\text{FPGA slices}} \cdot \frac{\text{FPGA bitstream}}{\text{ICAP bandwidth}} \tag{5.1}$$

where CCU slices is the total number of area slices a CCU requires, FPGA slices is the total number of slices available on a particular FPGA, FPGA bitstream is the bitstream size in MBs of the FPGA and ICAP bandwidth is the ICAP configuration speed. As a final remark, we assume that there is no delay associated with the runtime mapping such as task migration and context switching.

5.4 Two-Level Mapping Exploration with rSesame

In this section, we describe a case study using the aforementioned Molen model to perform an experimental validation of the two-level mapping explorations as presented in Section 3.5, which combines the static mapping exploration together with the runtime mapping exploration. In the static mapping exploration, the exploration is carried per application basic. As a result, a set of static mappings, which consists of HW, SW and pageable tasks, is identified. In the runtime mapping exploration, a high quality exploration of pageable tasks is performed to address any change in the application, architecture or the environment. In the remainder of this section, we will show how the Molen model can be used to perform the static and runtime mapping exploration of the Molen architecture.

5.4.1 Static Exploration with rSesame

At first-level, the Molen model described above is used to perform static mapping exploration. To perform the static exploration, MJPEG1 and MJPEG2 are

Mappings	HW Tasks	SW Tasks
M1	None	All tasks
M2	DCT1	DCT2-DCT4, Q1-Q4, VidIn1, VidOut1, Init1, VLE1
M3	DCT1,DCT2	DCT3, DCT4, Q1-Q4, VidIn1, VidOut1, Init1, VLE1
M4	DCT1-DCT3	DCT4, Q1-Q4, VidIn1, VidOut1, Init1, VLE1
M5	DCT1-DCT4	Q1-Q4, VidIn1, VidOut1, Init1, VLE1
M6	DCT1-DCT4, Q1	Q2-Q4, VidIn1, VidOut1, Init1, VLE1
M7	DCT1-DCT4, Q1,Q2	Q3, Q4, VidIn1, VidOut1, Init1, VLE1
M8	DCT1-DCT4, Q1-Q3	Q4, VidIn1, VidOut1, Init1, VLE1
M9	DCT1-DCT4, Q1-Q4	VidIn1, VidOut1, Init1, VLE1

Table 5.1: Different mappings used to perform static exploration of MJPEG1 onto the Molen architecture. In each successive mapping one task (DCT or Q) from the GPP is mapped onto its corresponding CCU.

separately mapped onto the given architecture, and a range of different mappings are evaluated. In particular, we observed the impact of different task mappings on the total execution time in terms of simulated clock cycles. Since the changes in the system are not considered while performing mapping decisions, with the static mapping exploration, a fixed set of tasks is mapped onto the GPP and CCUs. As a result, there are either HW tasks mapped onto the RP or SW tasks mapped onto the GPP, which means the set of pageable tasks is always empty.

In the first experiment, we mapped the MJPEG1 onto the Molen architecture. At first, all tasks from the application are mapped onto the GPP and, in each successive mapping, the mapping is changed by moving one task (either DCT or Q tasks) from the GPP to the CCUs. The mappings used to perform static mapping exploration of the MJPEG1 are reported in Table 5.1. The second column in the table lists the CCUs that are mapped onto the RP in the corresponding mapping, while rest of the tasks are mapped onto the GPP. For instance, in M1, all tasks are mapped to the GPP. In M2, DCT1 is mapped to its corresponding CCU, and rest of the tasks are mapped to the GPP. In M3, DCT1 & DCT2 are mapped to their corresponding CCUs, and rest of the tasks are mapped to the GPP. Similarly, in M9, all DCT and Q tasks are mapped to their corresponding CCUs, and the rest of the tasks are mapped to the GPP.

Figure 5.6 depicts the results of executing all different mappings listed in Table 5.1 with the Molen model. The primary y-axis (left) in the graph

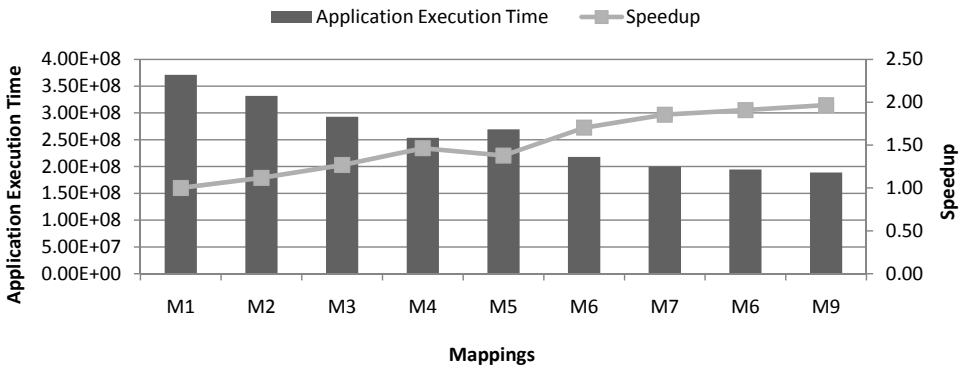


Figure 5.6: The application execution time and the corresponding speedups for the MJPEG1 application when mapped onto the Molen architecture. This result is obtained from the static exploration.

represents the application execution time measured for each mapping, and the secondary y-axis (right) in the graph represents the application speedup for each mappings compared to the software only execution. The x-axis lists all different mappings considered in this experiment as reported in Table 5.1. A first observation that can be noticed from the figure is in terms of application execution time. Due to the lower execution latency of CCUs as compared to the GPP, we might expect the system performance to significantly increase with more tasks being mapped onto the RP in each row in the table. However, the results show that in fact there is a non-linear tradeoff. This is because, moving the tasks to the CCUs, adds to the latency for reconfiguring the CCUs.

At the end of the first experiment, the most suitable mapping, while mapping MJPEG1 onto the Molen architecture, is obtained. The mapping M9 has the best speedup as depicted in Figure 5.6, and it consist of the following set of tasks: HW_1 = {DCT_1, DCT_2, DCT_3, DCT_4, Q_1, Q_2, Q_3, Q_4}, SW_1 ={VideoIn1, VLE1, VideoOut1, Init1} and $Page_1$ = {Ø}.

Similarly, in the second experiment, we mapped MJPEG2 onto the Molen reconfigurable architecture, and a wide range of different mappings are evaluated using the Molen model. We also recorded the most suitable mapping in terms of speedup at the end of the second experiment. This mapping has the following set of tasks: HW_2 = {DCT, Q}, SW_2 = {$VideoIn_1$, VLE_1, $VideoOut_1$, $Init_1$} and $Page_2$ = {Ø}. In the second level of exploration, these task sets can be optimized, such that, an efficient set of tasks can be found at runtime. This can possibly even improve the performance by avoiding the

Mappings	HW Tasks	Pageable Task	Tasks added to the SW task set after the Runtime Exploration
M1	DCT2-DCT4, DCT, Q1-Q4, Q	DCT1	None
M2	DCT3-DCT4, DCT, Q1-Q4, Q	DCT1-DCT2	None
M3	DCT4, DCT, Q1-Q4, Q	DCT1-DCT3	DCT2
M4	Q1-Q4, DCT, Q	DCT1-DCT4	DCT2
M5	Q1-Q4, Q	DCT1-DCT4, DCT	DCT2, DCT4
M6	Q2-Q4, Q	DCT1-DCT4, Q1	DCT2, DCT4
M7	Q3-Q4, Q	DCT1-DCT4, Q1-Q2	DCT2, DCT3, DCT4
M8	Q4, Q	DCT1-DCT4, DCT, Q1-Q3	DCT2, DCT3, DCT4, Q1
M9	Q	DCT1-DCT4, DCT, Q1-Q4	DCT2, DCT3, DCT4, Q1, Q2
M10	None	DCT1-DCT4, DCT, Q1-Q4, Q	DCT2, DCT3, DCT4, Q1, Q2, Q3

Table 5.2: The execution behavior of the given application model (MJPEG1 & MJPEG2) when mapped onto the Molen architecture at runtime.

reconfiguration delay for a task running on the GPP. In the following section, we show the mapping exploration at runtime by optimizing these task sets.

5.4.2 Runtime Exploration with rSesame

In order to carry out the second level of mapping exploration at the runtime, the combined application model shown in Figure 5.5 is considered. The initial task sets used as input for this exploration is the most suitable task sets obtained from the static exploration as discussed in the previous section. In this case, these task sets are formed as the combination of the task sets obtained for MJPEG1 and MJPEG2, when they are separately mapped onto the Molen architecture while performing static exploration. The task sets considered as input in this case are the following: HW = $\{DCT_1, DCT_2, DCT_3, DCT_4, Q_1, Q_2, Q_3, Q_4, DCT, Q\}$, SW = $\{VideoIn_1, VLE_1, VideoOut_1, Init_1, VideoIn, VLE, VideoOut, Init\}$ and Pageable = $\{\emptyset\}$.

To perform the runtime exploration with the rSesame, few tasks from the given HW set are considered as pageable tasks, and the behavior of these pageable tasks during the runtime is observed. To perform this experiment, in each

successive simulation run, one additional task from the HW task set is marked as a pageable task, and the behavior of the mapping is evaluated at runtime. Table 5.2 reports the results of this observation. The second and the third column in the table show the HW and the pageable tasks considered for each mapping, used to perform the runtime mapping exploration. The SW task set, in this case, is fixed, and it consists of the following tasks, VideoIn_1, VLE_1, VideoOut_1, Init_1, VideoIn, VLE, VideoOut and Init. The last column in the table lists the pageable tasks, which change their execution behavior, and become SW tasks after performing the runtime exploration. For these tasks, the runtime exploration decided that executing them *always* on the GPP is necessary. However, few other tasks than those listed in the last column in Table 5.2, either move to the GPP or to the RP based on the available resources in the latter.

This process is highly affected by the number of pageable tasks specified in the system. When there are less pageable tasks, these tasks can still be mapped onto the RP. Nonetheless, when this number increases, many of these tasks are only mapped onto the GPP. In M1 listed in Table 5.2, only DCT_1 from the HW task set is considered as a pageable, the rest of the tasks are considered as HW. In this case, the DCT_1 stays pageable, and executes on the GPP and the RP. Despite of many HW tasks listed in the second column of the corresponding mapping, DCT_1 is also mapped onto the RP during the runtime exploration. This is due to the arrival sequence of DCT_1. DCT_1 executes in parallel with other DCT tasks. If it arrives first for execution, and at that instance, if there are no HW tasks running on the RP, it can be mapped onto the RP. However, this may not be applicable for all DCT tasks. For instance, in M10, when all the tasks from the HW set are considered as pageable tasks, many tasks *always* mapped onto the GPP, due to the limited area on the RP.

Figure 5.7 depicts the results of executing different mappings listed in Table 5.2 with the rSesame framework in terms of application execution cycles. The HWonly (SWonly) execution is performed when all the tasks are mapped onto CCUs (GPP). The static mapping is performed when tasks listed in the first column of Table 5.2, are executed as fixed SW tasks, and the runtime mapping is performed when these tasks are considered as pageable as given in the table. As it can be inferred from Figure 5.7, the HWonly execution performance is higher than the SWonly execution performance due to the lower execution latency of CCUs as compared to that of the GPP. With the runtime mapping, the performance has ranged between these two values. We may expect the performance improvement with the runtime mapping. However, in

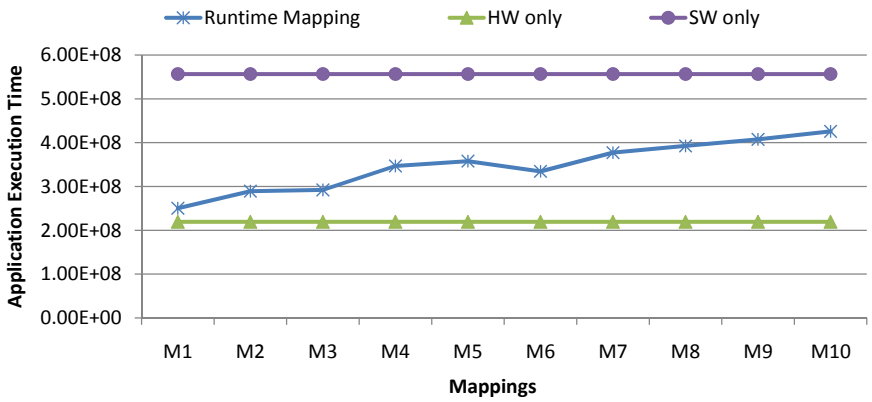


Figure 5.7: Comparison of the application execution time for the considered application model (MJPEG1 and MJPEG2) as given in Figure 5.5. The runtime mapping has better performance than the static and the SWonly mapping.

this case, we do not see any improvement of the runtime mapping. This is due to the higher SW latency for tasks as compared to their HW counterparts. Furthermore, the policy implemented by the RMM is very simple and it does not take into account the performance evaluation for making the mapping decision (it only looks at available area).

Although the outcome advices negatively against the use of runtime mapping, it does show that the rSesame can be used by a designer to evaluate such tradeoffs. We believe that real performance improvement can be obtained when the RMM implements intelligent task mapping policies. Although moving tasks onto the GPP with runtime exploration decreases performance, it may improve the hardware cost. In Section 5.6, we present the detailed analysis of the results obtained from the case study. The analysis shows indeed that with the runtime task mapping, we can evaluate area and performance tradeoffs.

5.5 Architecture Exploration Design Parameters

In the previous section, we showed that the Molen model instantiated from the rSesame can be employed to perform mapping exploration both statically and at runtime. In this section, we show that the model can be efficiently used to perform exploration of various design parameters. In particular, we show what kinds of design parameters can be obtained from the rSesame framework. The

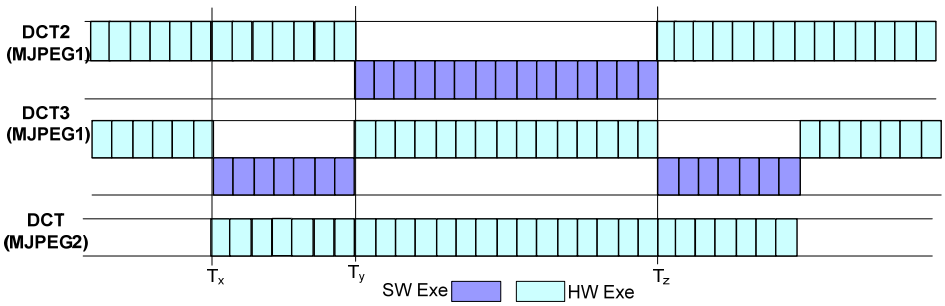


Figure 5.8: DCT execution snapshot in MJPEG1 and MJPEG2 while performing runtime mapping of the given application model onto the Molen reconfigurable architecture. DCT2 and DCT3 switched their mapping onto the GPP in order to accommodate DCT on the RP.

rSesame framework provides various useful design parameters to the designer. These includes the total execution time (in terms of simulated cycles), area usage, number of reconfigurations, percentage of reconfiguration, percentage of HW/SW execution and reusability efficiency. These are very important design parameters for architectural exploration. Based on these parameters, various characteristics of reconfigurable architectures, hardware-software partitioning algorithms and task mapping heuristics can be evaluated and compared. In the following part of this section, these design parameters are described in more detail.

5.5.1 Runtime Design Parameters

The Molen model provides various design parameters that describe the runtime behavior of the application and the architecture. These parameters provide an useful insight into the characteristics of the architecture and the efficiency of the task mapping. The runtime information provided by the model is recorded as a trace during the model execution. In the following, we describe these runtime parameters in more detail.

5.5.1.1 Spatial behavior of a task

A pageable task can change its behavior from HW execution to SW execution and vice versa, depending on various constraints imposed in the system,

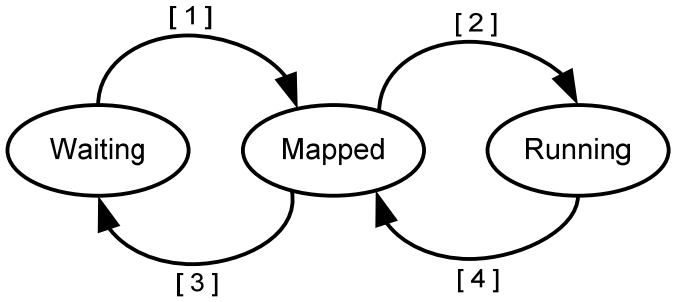


Figure 5.9: A Finite State Machine (FSM) showing the temporal behavior of a HW task.

such as the available resources on the RP. The spatial behavior of a task provides an indication on whether a pageable task is running as a HW or SW task. This information is vital to check the correctness of the spatial mapping behavior. Figure 5.8 captures a snapshot of such behavior for three different tasks - DCT2, DCT3 (from MJPEG1) and DCT (from MJPEG2), recorded for the runtime mapping of the given application onto the Molen architecture. The figure shows that, in order to accommodate DCT on the RP, DCT2 (at time T_y) and DCT3 (at times T_x and T_z) switched their mapping to SW providing DCT enough area to execute. The figure shows a snapshot of one specific time interval of the execution. The mapping behavior for all the tasks for the entire execution time-line can be retrieved from the model.

5.5.1.2 Temporal behavior of a task

As mentioned before, a HW task can further show various behaviors depending on its execution. It can either be in a waiting state, in a mapped state, or in a running state. A HW task is in a waiting state if the task is waiting to be mapped onto the RP. This happens, for example when there is no area available on the RP, or in case it has a task dependency with other tasks. A HW task is in a mapped state, if it is already configured on the RP, and it is not currently executing, but it may execute again. A HW task is in the running state when the task is actually busy performing execution.

Figure 5.9 depicts a Finite State Machine (FSM) showing different states of a HW task, where the numbers 1 to 4 refer to the following state transitions:

1. as soon as area becomes available or task dependency ends,
2. the task execution starts,
3. when other tasks need to be executed, and
4. the task execution finishes, but the task may execute again.

The mapped state has a reconfiguration delay associated with it. If a task transits from a waiting state to a running state, this delay is considered. However, if the task is already in the mapped state then this delay is ignored. The performance can be significantly improved by avoiding the former transition. A HW task may or may not enter the waiting state depending on the system conditions, such as available resources. To avoid a task entering the waiting state due to a lack of area on the RP, the task can be mapped onto the GPP. Moving a task onto SW not only has the slower execution of the GPP, but it also has task migration and context switching delay. However entering the waiting state also has reconfiguration penalty. The decision is up to the specific policy implemented by the RMM and/or by the RM. If the waiting state is due to a data dependency, it cannot be avoided.

Table 5.3 shows a snapshot of the temporal behavior of each HW task during a small period of the application run recorded for the runtime mapping of the given application onto the Molen architecture. At each execution, the behavior of each HW task is noted as R, M and W which refer to the **R**unning, **M**apped and **W**aiting state respectively. In each row, the state of all the HW tasks is recorded at each execution. As it can be inferred from the table, HW tasks change their state (R, M and W) with time when the system behavior changes. The first row shows that DCT2 and DCT3 are in the running state, DCT1, DCT4 and VideoOut1 (VOut1) are in the mapped state, while the other tasks are in the waiting state. The RP has limitated area, and as a result, only few tasks can be in the mapped/running state. Moreover, all the Q tasks, VLE1 and VOut1 have a data dependency with DCT in the application. As a result, other tasks are in the waiting state, and in the successive executions, these in turn, are mapped and run.

As it can be inferred from Table 5.3, when DCT4 changes its state from M to R, the reconfiguration is avoided. However, in the case of DCT1, when the state changes from W to R (as it has to pass through the mapped state), the reconfiguration delay is added. In the latter case, by saving the first M state (see first row for DCT1 in Table 5.3) for three more executions, this delay

Time

VideoIn1	DCT1	DCT2	DCT3	DCT4	Q1	Q2	Q3	Q4	VLE1	VideoOut1	DCT
					...						
W	M	R	R	M	W	W	W	W	W	M	-
R	W	R	R	**M**	W	W	W	W	W	M	-
M	W	R	R	**R**	W	W	W	W	W	M	-
R	**W**	R	R	R	W	W	W	W	M	M	-
M	**R**	R	R	R	W	W	W	W	W	W	-
R	R	R	R	R	W	W	W	W	W	W	-
					- - - - - - - - - - - - MJPEG2 starts - - - - - - - - - - - -						
SW	R	R	M	R	W	W	R	W	W	W	W
SW	R	R	W	R	W	R	R	W	W	W	R
SW	R	W	W	R	W	R	R	W	W	W	R
SW	R	W	W	W	W	M	R	R	W	W	R
SW	R	W	W	W	W	W	R	R	M	W	M
SW	R	W	W	W	W	W	M	M	R	M	M
SW	R	W	W	W	W	W	M	M	M	R	M
					...						

Table 5.3: A snapshot of the temporal behavior of HW tasks for the given application model. HW tasks can have three execution stages, such as R = **R**unning, W = **W**aiting and M = **M**apped state

can be easily avoided. The mappings can be optimized by understanding and analyzing such behavior. Temporal behavior of a task is vital not only to test the correctness of the mapping algorithms but also for their optimization. We also observe the spatial behavior of tasks from Table 5.3. For example, when MJPEG2 arrives, VideoIn1(VIn1) from MJPEG1 is moved to the GPP (indicated by SW in the table) and DCT from MJPEG2 is mapped onto the RP. This is again due to the area limitation on the RP.

5.5.1.3 Number of hardware or software tasks

The number of hardware or software tasks provides information about the total number of tasks being executed on HW and SW at a particular point during the application run. Table 5.4 shows this information at various checkpoints of the execution time-line, while performing runtime task mapping of the considered application model onto the Molen architecture. As it can be inferred from the table, at the first checkpoint, only MJPEG1 is running, and tasks such as, DCT1, DCT2, DCT3, DCT4, Q2, Q3, Q4 and Q5 are mapped onto the RP. Note that, not all tasks are mapped onto the reconfigurable hardware at once, they are executed after reconfiguration, whenever it is required. At the second checkpoint (see bi-direction arrow in the table), when MJPEG2 arrives, DCT and Q from MJPEG2 are also mapped onto the RP. Note that, Table 5.4 accumulates all the tasks that are mapped onto the RP in each interval. The mapping of tasks within an interval may be different in each snapshot. The detailed representation of a quarter of a period in Table 5.4 is given as a snapshot in Figure 5.8. The information provided by the rSesame framework is indispensable in order to evaluate the correctness of the mapping.

5.5.2 Execution time

The execution time is recorded in terms of simulated clock cycles. The SW execution time is noted as the total number of cycles when all the tasks are mapped onto the GPP only and HW execution time is recorded when the tasks are mapped onto the RP. The speedup is calculated as a ratio of these two values.

Tasked Mapped onto the RP								
DCT1, DCT2, DCT3, DCT4, Q2, Q3, Q4, Q5	DCT1, DCT2, DCT3, DCT4, Q2, Q3, Q4, Q5, DCT, Q	DCT1, DCT2, DCT3, DCT4, Q2, Q3, Q4, Q5, DCT, Q	DCT1, DCT2, DCT3, DCT4, Q2, Q3, Q4, Q5, DCT, Q	DCT1, DCT2, DCT3, DCT4, Q2, Q3, Q4, Q5, DCT, Q	DCT1, DCT2, DCT3, DCT4, Q2, Q3, Q4, Q5, DCT, Q	DCT1, DCT2, DCT3, DCT4, Q2, Q3, Q4, Q5, DCT, Q	DCT1, DCT2, DCT3, DCT4, Q2, Q3, Q4, Q5, DCT, Q	DCT
MJPEG1	↔							
MJPEG2								

Time: 0, 21023003, 50233590, 79237914, 112004986, 142307136, 175019325, 202942891, 236278868, 298006350

Table 5.4: Task mapping breakdown with respect to time for runtime task mapping of the given application model with MJPEG1 and MJPEG2 onto the Molen architecture. Each interval lists all the tasks that are mapped on the RP within that interval.

5.5.3 Percentage of HW/SW execution time

The percentage of HW execution and SW execution are computed as the total percentage of the execution time contributed by the RP for HW execution and the total percentage of the execution time contributed by the GPP for SW execution of an application respectively. Similarly, the percentage of reconfiguration time calculates the percentage of the total execution time spent in reconfiguration. This provides an indication on how much of the total time is spent in the computation and how much is just spent in reconfigurations. These values are calculated as follows.

The percentage of SW execution time is given by:

$$\text{SW Exec}(\%) = \frac{\sum_{i=1}^{N} \#\text{SWEx}(T_i) \cdot T_{\text{SW(i)}}}{\text{TotalExecTime}} \cdot 100 \tag{5.2}$$

where #SWEx(T_i) is the total number of SW executions counted by the model for task T_i, $T_{SW(i)}$ is the software execution latency for task T_i and TotalExecTime is the total simulated execution time.

The percentage of HW execution time is given as:

$$\text{HW Exec.}(\%) \leq \frac{\sum_{i=1}^{N} \#\text{HWEx}(T_i) \cdot T_{\text{HW(i)}}}{\text{TotalExecTime}} \cdot 100 \tag{5.3}$$

where #HWEx(T_i) is the total number of HW executions counted for task T_i by the model, $T_{HW(i)}$ is the hardware execution latency for task T_i and TotalExecTime is the total execution cycles incurred while running an application onto the given reconfigurable architecture.

Note that, the HW execution percentage can only be given here as an upper bound, since the execution of tasks on the RP can be performed in parallel. The metric calculated here is an accumulated value. The simulator, however, can give the actual snapshots. A similar equation holds for the time spent reconfiguring, which is given as percentage of the total execution time as follows:

$$\text{Recon}(\%) \leq \frac{\sum_{i=1}^{N} \#\text{Recon}(T_i) \cdot T_{\text{Recon(i)}}}{\text{TotalExecTime}} \cdot 100 \tag{5.4}$$

where #Recon(T_i) is the number of times T_i is configured and $T_{Recon(i)}$ is the reconfiguration delay of T_i and TotalExecTime is the total execution cycles incurred while running an application onto the given reconfigurable architecture.

5.5.4 Number of reconfigurations

The number of reconfigurations is recorded as the total number of reconfigurations incurred during the execution of an application onto the given architecture. This provides an indication on how efficiently the reconfiguration delay was avoided, while mapping tasks onto the RP. For example, mapping task A, task B, and then task A again on the RP requires 3 reconfigurations, while changing this sequence of mapping to task A, task A and then task B requires only 2 reconfigurations.

5.5.5 Time-weighted area usage

The weighted area usage factor is a metric that computes how much area is used throughout the entire execution of an application on a particular architecture. This provides an indication on how efficiently the RP area is utilized. This is calculated as follows:

$$\text{Area Usage}(\%) = \frac{\sum_{i=1}^{N} \text{Area}(T_i) \cdot T_{\text{HW(i)}} \cdot \#\text{HWEx}(T_i)}{\text{TotalExecTime} \cdot \text{Area(RP)}} \cdot 100 \tag{5.5}$$

where Area(T_i) is the area occupied by task T_i on the RP, $T_{HW(i)}$ is the hardware execution latency of T_i, #HWEx(T_i) is the total number of HW executions counted by the model for task T_i, Area(RP) is the total area available on the RP and TotalExecTime is the total execution time of the application.

5.5.6 Reusability Efficiency

A CCU execution onto the RP has two phases: the *configuration phase*, where its configuration data, which represents a task is loaded onto the RP, and the *running phase*, where the CCU is actually processing data. In an ideal case, a CCU can be configured onto the RP only once and it is executed in all other cases. Nonetheless, this is not always possible as the RP has limited area. The Reusability Efficiency (RE) is the ratio of the reconfiguration time that is saved due to the hardware configuration reuse to the total execution time of any task. The RE of a CCU can be defined as follows:

$$\mathrm{RE}_{\mathrm{task}} = \frac{(\#\mathrm{HWEx} - \#\mathrm{Recon}) \cdot T_{\mathrm{Recon}}}{\#\mathrm{HWEx} \cdot T_{\mathrm{HW}} + \#\mathrm{SWEx} \cdot T_{\mathrm{SW}} + \#\mathrm{Recon} \cdot T_{\mathrm{Recon}}} \tag{5.6}$$

where #HWEx, #SWEx and #Recon are the number of HW executions, SW executions and reconfigurations of a CCU respectively. Similarly, T_HW, T_SW and T_Recon is the corresponding hardware, software and reconfigurable latencies.

The RE of a task indicates the percentage of the total time saved by a CCU when multiple reconfigurations are avoided or, in other words, a CCU is reused. In Equation 5.6, the numerator represents the time that is saved, when a mapping of a CCU is reused and the denominator represents the total execution time. The total RE for an application can be calculated as the summation of the numerator in Equation 5.6 for all N tasks divided by the total execution time for the whole application as follows:

$$\mathrm{RE}_{\mathrm{App}} \leq \frac{\sum_{i=1}^{N} (\#\mathrm{HWEx(i)} - \#\mathrm{Recon(i)}) \cdot T_{\mathrm{Recon\ (i)}}}{\mathrm{TotalExecTime}} \tag{5.7}$$

Note that the RE calculated in this way for the whole application can only be given here as an upper bound, since the execution of tasks on the reconfigurable hardware can be performed in parallel. A higher RE can obtain a higher speedup. To study this relation, we use the RE as an evaluation parameter to study the behavior of each CCU.

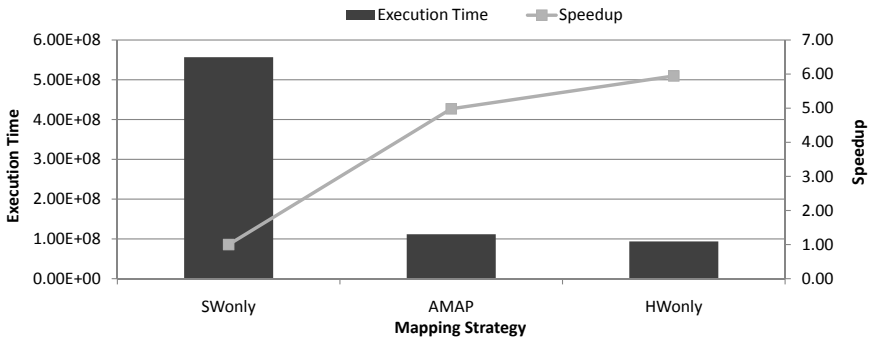


Figure 5.10: Application execution time and the corresponding application speedup of mapping the given MJPEG application onto the Molen architecture with the HWonly, the SWonly and the runtime task mapping strategies.

5.6 Results Analysis

In this section, we provide a detailed analysis of the experimental results and their implications for the presented case study. We conducted a variety of experiments on the instantiated model by performing mapping of the application model in Figure 5.5 for the Molen architecture with different static and runtime mapping strategies. At first, we performed the SWonly mapping, where all the tasks are mapped onto the GPP only. Secondly, we performed the HWonly mapping, where all tasks are mapped onto CCUs only. In both cases, the tasks are mapped statically onto the Molen architecture. The task mapping is fixed, and it cannot be changed at runtime. Furthermore, all tasks are either considered HW or SW tasks, i.e. there are no pageable tasks. Lastly, we performed runtime mapping of tasks onto the GPP and the RP. To perform this experiment, all tasks are considered as pageable, such that the full impact of runtime task mapping can be evaluated. As discussed before, we employed a simple strategy, called AMAP, for performing runtime task mapping. AMAP tries to map tasks based on area availability. We compared the results of the runtime task mapping using the AMAP heuristic with the static task mapping, when all tasks are either mapped onto the GPP only or to the RP only. We evaluated and compared the task mapping based on the aforementioned design parameters, which are execution time, speedup, percentage of HW/SW execution, number of reconfigurations, time-weighted area usage and RE. In the rest of this section, we discuss the results by using these parameters in more detail.

Figure 5.10 depicts the results of mapping the given MJPEG application

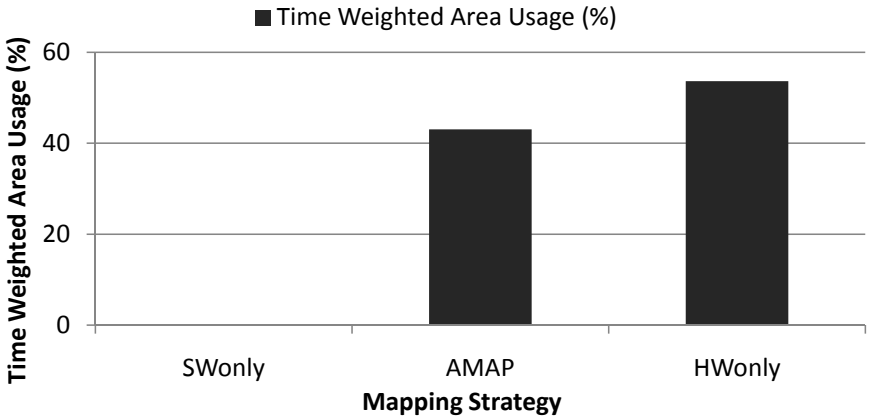


Figure 5.11: Time-weighted area usage and the corresponding application speedup of mapping the given MJPEG application onto the Molen architecture with the HWonly, the SWonly and the runtime task mapping strategies.

model onto the Molen architecture with both static and runtime mapping strategies. The HWonly (SWonly) execution is measured when all the tasks are mapped onto CCUs (the GPP). The primary y-axis (left) in the figure represents the measured application execution time, and the secondary y-axis (right) represents the application speedup compared to the SWonly execution. It can be inferred from the figure that the application execution time of the runtime mapping is five times better than the SWonly execution, and the application execution time of the HWonly mapping is almost six times better than that of SWonly execution. Similarly, the runtime application mapping is 1.19 times slower than the HWonly mapping. When all tasks are mapped onto CCUs, the performance is higher than when they are all mapped onto the GPP. With the runtime mapping, the performance has ranged between these two values. Mapping all the tasks onto CCUs as in the case of the HWonly gives better performance, but it consumes more hardware resources. This can be observed from Figure 5.11.

Figure 5.11 depicts the time-weighted area usage for mapping the given application onto the Molen architecture with both static and runtime mapping. Comparing the time-weighted area usage of the HWonly mapping and the AMAP heuristic in the figure, we can observe that the cost of using reconfigurable area in case of the HWonly mapping is higher than the performing runtime mapping with the AMAP heuristic. With the HWonly mapping all tasks are mapped onto the CCUs, and as a result, the hardware cost is high.

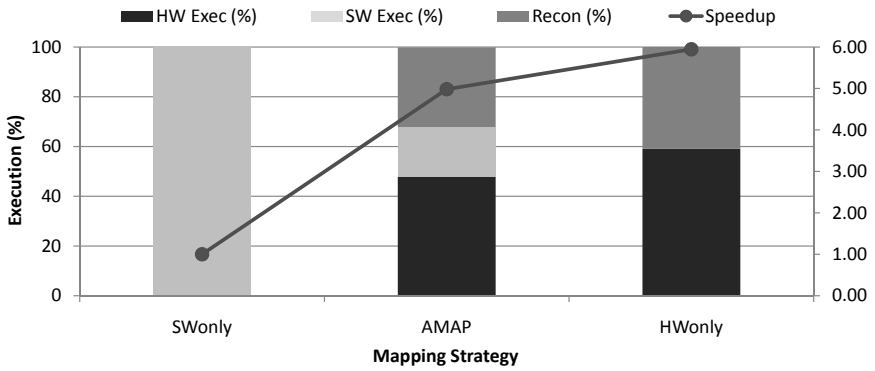
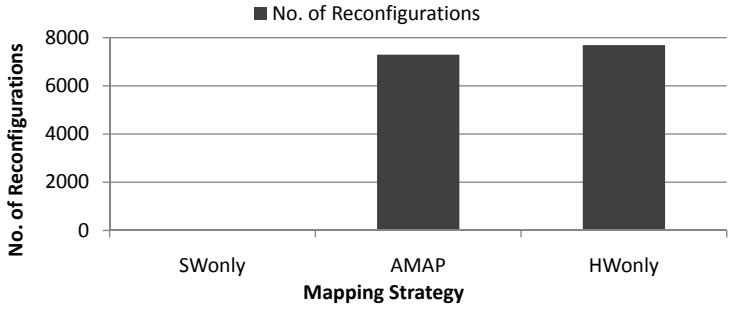


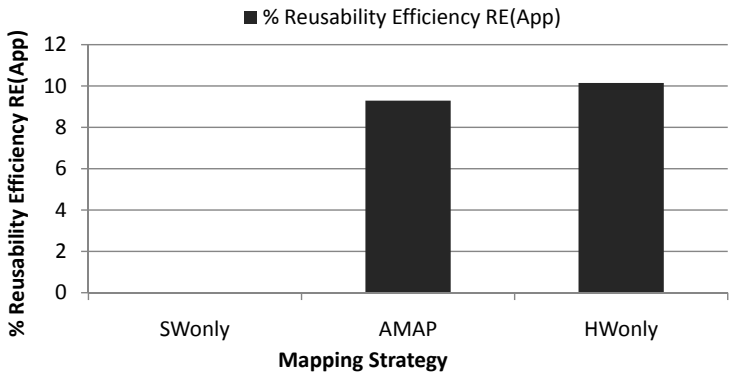
Figure 5.12: Percentage of hardware execution, software execution and reconfiguration of mapping the given MJPEG application onto the Molen architecture with the HWonly, the SWonly and the runtime task mapping strategies.

However, mapping all tasks onto the GPP, has no hardware cost, but results with slow performance. A tradeoff in terms of performance and resources can be obtained with the runtime mapping, which performs selective task mapping onto the RP at runtime. The rSesame framework assists in exploring such tradeoffs.

Figure 5.12 depicts the percentage of hardware execution, software execution and reconfiguration for mapping the given MJPEG application onto the Molen architecture. With the SWonly mapping, all tasks are mapped onto the GPP, and as a result, there are no hardware execution and reconfigurations. With the HWonly mapping, more than 40% of the application execution time is spent while reconfiguring CCUs, and with the runtime mapping, approximately 32% of the time is spent with the reconfiguration. This slight decrease in the reconfiguration time, in AMAP heuristic is due to the result of SW mapping. In the case of AMAP, tasks are also mapped onto the GPP. While mapping tasks onto the GPP, the reconfiguration overhead is avoided, and as a result, the reconfiguration percentage is lowered. The reconfiguration overhead can be avoided by moving a task onto the GPP. In some cases, this may achieve a considerable performance gain, especially in the system, where reconfiguration overhead is significantly high. Nevertheless, changing a task mapping has a migration and a context switching delay. Moving a task onto the GPP is beneficial, only if the software latency of a task is lower than executing it on the hardware together with the task migration delay. Therefore, the decision, weather to change the mapping for a particular task, depends on a particular system and the constraints imposed on it. With the rSesame



(a) **Number of Reconfigurations**



(b) **Reusability Efficiency (RE_{App})**

Figure 5.13: Number of reconfigurations and the application Reusability Efficiency (RE_{app}) of mapping MJPEG application onto the Molen architecture with the HWonly, the SWonly and the runtime task mapping strategies.

framework, such possibilities can be explored.

Figure 5.13 depicts the number of reconfigurations and the RE_{app} of mapping the given MJPEG application onto the Molen architecture. With the SWonly mapping, none of the tasks is mapped onto the RP, and as a result, there is no value for reconfiguration and the RE_{app} for the former in Figure 5.13(a) and Figure 5.13(b) respectively. While comparing the HWonly mapping and the AMAP in terms of number of reconfigurations, it can be inferred from the figure that the AMAP has less reconfigurations than the HWonly mapping. As explained before, when mapping tasks onto the GPP at runtime with the AMAP heuristic, the reconfiguration overhead is avoided, and as a result the number of reconfigurations is lowered. Similarly, it can be inferred from Figure 5.13(b) that there is no significant difference in the RE_{app} of the runtime

mapping and the HWonly mapping, while we may expect a better configuration re-use for the runtime mapping. This is due the fact that the task mapping policy implemented by the RMM performs task mapping based on area, and it does not take the configuration re-use in account. While implementing other policies, the different value for RE_{app} can be obtained. Such detailed analysis of architectural parameters is provided in the next chapter.

For this case study, we note that all the above system-level simulations of running MJPEG1 and MJPEG1 application with inputs of 8 and 4 picture frames of 128 x 128 pixel respectively, can be executed in less than 10 seconds, thus allowing *fast performance* evaluation of the model and extensive design space exploration.

5.7 Conclusions

In this chapter, we described a case study to show an application of the rSesame framework on a real reconfigurable architecture. In order to perform mapping exploration based on various design attributes, we instantiated a model from the rSesame framework for the Molen architecture. A mapping strategy based on the area availability is implemented in the model in order to perform exploration of different application-to-architecture mappings at runtime. We demonstrated that the instantiated model can be employed to perform both static and runtime mapping exploration of the Molen reconfigurable architecture. We also showed that the instantiated model can be efficiently used to perform exploration of various design parameters, such as execution time, area usage, number of reconfigurations and percentage of hardware and software execution. Based on these design parameters, we evaluated and compared the runtime mapping against static mapping. The obtained results show that mapping all tasks onto the RP gives better performance, however, it consumes more hardware resources. With the runtime mapping, a tradeoff can be obtained in terms of performance and resources. The rSesame framework assists to explore such tradeoffs.

In the next chapter, we evaluate the rSesame framework's characteristics by showing that it can easily and quickly model, simulate and compare a wide range of runtime mapping heuristics from diverse domains. We evaluate these heuristics with the rSesame framework by considering, for the same reconfigurable architecture model, different numbers of resources, under a more

complex application setup.

Note. The content of this chapter is based on the the following articles:

K. Sigdel, M. Thompson, A.D. Pimentel, T. P. Stefanov, K.L.M. Bertels, **System-Level Design Space Exploration of Dynamic Reconfigurable Architectures**, *Proceedings of the International Symposium on Systems, Architectures, MOdeling and Simulation (SAMOS'08)*, Samos, Greece, July 2008, pp. 279–288.

K. Sigdel, M. Thompson, A.D. Pimentel, C. Galuzzi, K.L.M. Bertels, **System-Level Runtime Mapping Exploration of Reconfigurable Architectures**, *Proceedings of the Reconfigurable Architectures Workshop (RAW'09)*, Rome, Italy, May 2009, pp. 1-8.

6

Task Mapping Heuristics Evaluation at Runtime

As described in Chapter 4, the rSesame framework is a generic modeling and simulation framework which can be used to explore and evaluate reconfigurable systems at early design stages. The framework can be used to perform rapid exploration of several parameters, such as architectural characteristics, hardware-software partitionings, application-to-architecture mappings, and scheduling strategies. In Chapter 5, we showed that the rSesame framework can be efficiently used to perform quick exploration of several application-to-architecture mappings reconfigurable architectures, both statically and at runtime. In this chapter, we elaborate the use of the rSesame framework to perform runtime mapping exploration of reconfigurable architectures in more detail. In particular, we show that the framework can easily and quickly model, simulate and compare a wide variety of task mappings strategies at runtime, under different resource conditions.

The main features of the rSesame framework include flexibility, ease of use, fast performance, and its applicability to a wide range of reconfigurable systems. We describe a case study where we test these characteristics of the framework on the Molen architecture, by evaluating and comparing different task mapping heuristics at runtime, under different resource conditions. These heuristics are evaluated and compared based on various design attributes, such as execution time, number of reconfigurations, time-weighted area usage, number of hardware and software tasks, percentage of hardware/software execution, percentage of reconfiguration and hardware reusability efficiency. With this case study, we show that the rSesame framework can be efficiently deployed not only to perform rapid exploration of different application-to-

architecture mappings, but also to evaluate different task mapping strategies and different architectural conditions at runtime.

This chapter is organized as follows. Section 6.1 presents an overview of the mapping heuristics taken from various domains, which are considered in the proposed case study. Section 6.2 discusses the required experimental setup to perform this case study. In particular, it discusses the application model and the architecture details of the reconfigurable architecture. In Section 6.3, we present the detailed analysis and the comparison of the results obtained from the case study. Finally, Section 6.4 summarizes the major contributions and concludes the chapter.

6.1 Task Mapping Heuristics

The rSesame framework allows easy modification and adjustment of individual components in the model, while keeping other parts intact. We illustrate this feature of the framework by allowing designers to experiment with different kinds of runtime application mapping heuristics. The considered heuristics have variable complexity with respect to their implementation, and the nature of their execution. In the original context, they were used at different system stages, ranging from the lower architecture level to operating system (OS), and the higher application levels. This illustrates an example of the rSesame framework's flexibility in incorporating different kinds of algorithms from various domains. These heuristics are taken from literature, and have been adapted to fit in the framework. In the following, we discuss these heuristics in more detail.

6.1.1 AMAP : As Much As Possible Heuristic

AMAP tries to maximize the use of FPGA resources (such as area) as much as possible, and it performs task mapping based on resource availability. In this case, tasks are executed to the FPGA if the latter has enough resource to accommodate them; otherwise, they are executed on the GPP. This straightforward heuristic can be used as a simple resource management strategy in various domains. This is also used as a default mapping strategy for the rSesame framework. Algorithm 6.1 presents the pseudo-code that describes the

Algorithm 6.1 Pseudo-code for the AMAP task mapping heuristic.

```
1: HW ← set of tasks mapped onto FPGA
2: SW ← set of tasks mapped onto the GPP
3: if T_i.area ≤ area then
4:     {T_i is mapped onto FPGA}
5:     HW = HW ∪ T_i
6:     area = area - T_i.area
7: else
8:     {Map T_i onto the GPP}
9:     SW = SW ∪ T_i
10: end if
```

functionality of AMAP heuristic for performing runtime mapping of a task T_i. The heuristic chooses to execute task T_i to the FPGA if there is sufficient resources (e.g. area in the algorithm) for T_i (line 3 to 6 in Algorithm 6.1). On all other conditions, tasks are executed on the GPP (line 7 to 9 in Algorithm 6.1).

6.1.2 CBH : Cumulative Benefit Heuristic

CBH maintains a Cumulative Benefit (CB) value for each task that represents the amount of time that would have been saved up to that point if the task had always been executed to the FPGA. In this case, mapping decisions are made based on these values and the available resources. For example, if the available FPGA resources are not sufficient to load the current task, tasks can be swapped if the CB of the current task is higher than that of the to-be-swapped-out set. In [103], this heuristic is used for dynamic coprocessor management of reconfigurable architectures at architecture level.

Algorithm 6.2 presents the pseudo-code that describes the functionality of the CBH heuristic for performing runtime mapping of a task T_i. If resources, such as area slices, are available in the FPGA, then T_i is executed to the FPGA, if the CB of T_i is larger than its loading time, defined by ($T_{SW(i)}$ - $T_{HW(i)}$), where $T_{SW(i)}$ and $T_{SW(i)}$ are software and hardware latencies of task T_i respectively (line 3 to 8 in Algorithm 6.2). If the FPGA lacks current capacity for executing task, the heuristic searches for a subset of FPGA-resident tasks, such that removing the subset yields sufficient resources in the FPGA to

Algorithm 6.2 Pseudo-code for the CBH task mapping heuristic.

```
1:  HW ← set of tasks mapped onto FPGA
2:  SW ← set of tasks mapped onto the GPP
3:  if T_i.area ≤ area then
4:     if CB(T_i) > (T_SW(i) - T_HW(i)) then
5:        {T_i is mapped onto FPGA}
6:        HW = HW ∪ T_i
7:        area = area - T_i.area
8:     end if
9:  else
10:    {Not enough area, swap the mapped tasks.}
11:    while area ≤ T_j.area and j ∈ HW do
12:       if CB(T_i) - (T_SW(i) - T_HW(i)) > CB(T_j) then
13:          area = area + T_j.area
14:       end if
15:    end while
16:    if T_i.area ≤ area then
17:       {T_i is mapped onto FPGA}
18:       HW = HW ∪ T_i
19:       area = area - T_i.area
20:    else
21:       {Map T_i onto the GPP}
22:       SW = SW ∪ T_i
23:    end if
24: end if
```

execute the current task. The condition, however, is that, all the tasks in the subset must have smaller CB value than the current task (line 9 to 18 in Algorithm 6.2). If such a subset is not attained, then the current task is executed to the GPP (line 19 to 22 in Algorithm 6.2).

6.1.3 IBH : Interval Based Heuristic

In IBH, the execution is divided into a sequence of time slices (intervals) for mapping and scheduling. At the beginning of each interval, a task is examined for its execution. In each interval, the execution frequency of each task is counted, and the mapping decisions are made based on the frequency count

Algorithm 6.3 Pseudo-code for the IBH task mapping heuristic.

```
1:  T ← set of all tasks.
2:  while T != empty and area ≤ Total_area do
3:     Select T_i with maximum frequency count
4:     if area + T_i.area ≤ Total_area then
5:        map T_i onto FPGA
6:        area = area + T_i.area
7:     else
8:        map T_i onto GPP
9:     end if
10:    Remove T_i from T
11: end while
12: Map rest of the tasks from T onto the GPP
```

of the previous intervals, such that tasks with the highest frequency count are mapped onto the FPGA.

Algorithm 6.3 presents the pseudo-code that describes the functionality of the IBH heuristic for performing runtime mapping in each interval for a T set of tasks. Working from the highest to the lowest frequency count, each task $T_i \in$T that satisfies the current resource conditions is selected for FPGA execution. The area constraint is updated accordingly before considering the next task. This process continues until the FPGA is full, or there is no task left in T (line 2 to 6 in Algorithm 6.2). If the FPGA current capacity is not enough for executing any task from T, then they are executed with the GPP (line 8 to 12 in Algorithm 6.2). As it can be seen in Algorithm 6.3, tasks are executed to the FPGA based on frequency count, but other mapping criteria, such as speedup, can also be used. The main idea of this heuristic is to divide the execution into different intervals and perform mapping in each interval. In [104], this heuristic is used for resource management in a multi-threaded environment at OS level.

The implementation of this heuristic with the Molen model is straight forward. The only added concern with this heuristic is intervals marking for the task mapping. To implement this heuristic, marking of the execution intervals is done by inserting a special event in the application model. Whenever this special event is encountered in the mapping layer, the frequency count is revisited, and the task mapping is updated.

6.1.4 RBH: Reusability Based Heuristic

Reconfiguration overhead has always been a serious concern for reconfigurable architectures, as it can drastically limit the performance of such architectures. In an ideal case, a task can be configured on the reconfigurable hardware only once and then be reused to accelerate the application in all other cases. The reuse of the hardware configuration avoids multiple configurations, and as a result, reconfiguration overhead can be significantly reduced. Especially in case of application domains such as streaming and networking, where certain tasks are executed in a periodic manner e.g. on the basis of pixel blocks or entire frames, hardware configuration reuse can easily be exploited. To take advantage of such characteristics of streaming applications, we propose a new heuristic called Reusability Based Heuristic (RBH).

RBH is based on the hardware configuration reuse concept, which tries to avoid the reconfiguration overhead by reusing the configurations, which are already available on the FPGA. The basic idea of the heuristic is to avoid reconfiguration as much as possible, in order to reduce the total execution time. For certain tasks that are mapped onto the FPGA, the heuristic preserves them in the FPGA after their execution. These tasks are not removed from the hardware, so that their hardware configurations can be reused when the task is re-executed. Reusing hardware configurations multiple times can significantly avoid reconfiguration overhead; thus, performance can be considerably improved. Unfortunately, preserving hardware configurations is not possible for all tasks. For this reason, the heuristic tries to preserve hardware configurations for selected tasks. For example, tasks that have higher reconfiguration delay, and occur more frequently in the system have priority on being preserved in the FPGA.

As mentioned in Chapter 5, we define three states for a task: a waiting state, a mapped state, and a running state. A task is in the waiting state if it waits to be mapped. A task is in the mapped state if it is already configured on the FPGA, but it is not being executed; however, it may be re-executed later. A task is in the running state when the task is actually processing data. It should be noted that when a task is in the mapped state, its hardware configuration is saved in the FPGA. Thus, when the task needs to be re-executed it can immediately start processing without reconfiguration.

Algorithm 6.4 presents the pseudo-code that describes the functionality of the RBH heuristic for performing runtime mapping of a task T_i. If T_i is

Algorithm 6.4 Pseudo-code for the RBH task mapping heuristic based on hardware configuration reuse.

```
1: {Task already mapped on FPGA, do not configure.}
2: if T_i == MAPPED then
3:    T_i.state ← RUNNING;
4: else
5:    if area ≥ T_i.area then
6:       if SpeedUp(T_i) > 1 then
7:          {Task not mapped on FPGA, configure it.}
8:          configure(T_i);
9:          T_i.state ← RUNNING;
10:      end if
11:   else
12:      for All tasks T_j on the FPGA do
13:         if SpeedUp(T_j) < SpeedUp(T_i) then
14:            candidateSet = candidateSet ∪ T_j
15:         end if
16:      end for
17:      while area ≤ T_i.area do
18:         Select T_k ∈ candidateSet with lowest RER
19:         removeSet = removeSet ∪ T_k
20:         area = area + T_k.area;
21:      end while
22:      if T_i.area ≤ area then
23:         for All task T_m ∈ removeSet do
24:            T_m.state = WAITING;
25:         end for
26:         {Task not mapped on FPGA, configure it.}
27:         configure(T_i);
28:         T_i.state ← RUNNING;
29:      end if
30:   end if
31: end if
```

already configured, then it starts directly processing data (line 1 to 4 in Algorithm 6.4). However, if T_i is not currently available in the FPGA, then the task is evaluated for its speedup. If resources are available, T_i is executed to the FPGA only if there is a performance gain (line 5 to 10 in Algorithm 6.4). The

performance gain in this case is measured in terms of speedup. The speedup for each task is measured at runtime by using the following equation:

$$\text{Speedup} = \begin{cases} \frac{T_{SW}}{T_{HW}} & t = 0 \\ \frac{T_{SW} \cdot (\#HWEx + \#SWEx)}{\#SWEx \cdot T_{SW} + \#HWEx \cdot T_{HW} + \#Recon \cdot T_{Recon}} & t > 0 \end{cases} \tag{6.1}$$

where #HWEx, #SWEx and #Recon are the number of HW executions, SW executions and reconfigurations of a task respectively. Similarly, T_{HW}, T_{SW} and T_{Recon} are the corresponding hardware, software and reconfigurable latencies, and t is the execution time-line. When the application execution starts, t = 0. The heuristic maintains a profiling count of HW executions, SW executions and reconfigurations for all tasks. Each time a task is executed, these counters for that task are updated. For instance, if a task is executed with the GPP, its SW count is incremented, and if the task is executed in the FPGA, its HW count is incremented. Similarly, the reconfiguration count of a task is incremented when a task is (re)configured. These count values for each task are accumulated from all the previous executions. As a result, they reflect the execution history of a task. The speedup calculated with these count values indicates the precise speedup of a task up to that point of execution.

If the available resources are not enough in the FPGA, a set of tasks from the FPGA is swapped to accommodate T_i in the FPGA. The task swapping, in this case, is done based on two factors: a) Speedup and b) Reconfiguration-to-Execution Ratio (RER). In the first step, a candidate set of tasks from the FPGA is selected, in such a way that these tasks are less beneficial than the current task in terms of speedup (line 12 to 16 in Algorithm 6.4). The speedup in this case is also calculated by using the Equation 6.1. In the second step, the candidate set is examined for its RER ratio, such that tasks with the lowest RER values are swapped first (line 17 to 21 in Algorithm 6.4). The RER value for each task is computed as follows:

$$\text{RER} = \frac{T_{Recon}}{T_{HW}} \cdot \text{Exec\_Freq} \tag{6.2}$$

where Exec_Freq is the average execution frequency of the task in its past history. The execution frequency of a task can be simply computed from the execution profile of each task with respect to the total execution count of that

application as follows.

$$\text{Exec\_Freq} = \frac{\#\text{HWEx}}{\sum_{i=1}^{N} \text{HW}_i\text{Ex}} \tag{6.3}$$

where the numerator represents the number of times a task is executed on a hardware. The denominator represents the total hardware execution count of the entire application, and N represents the total number of tasks in an application.

The task with a high RER value indicates that it has high reconfiguration-per-execution delay, and it has executed frequently, in its history in the system, making it a probable candidate for future execution. The heuristic makes a careful selection while removing tasks from the FPGA. By preserving tasks with high RER values as long as possible in the FPGA, we try to avoid the reconfiguration of the frequently executed tasks. We would like to stress the fact that the speedup value computed using Equation 6.1 is not a constant factor. This value is continuously updated based on the execution profile of the task at runtime. Hence, mapping tasks onto the FPGA based on such value, represents the precise system behavior at that instance of time. Note that the RBH is a generic heuristic, and it is not restricted to one type of resources or one type of architecture. To perform runtime mapping decisions considering multiple resources (such as memory, DSP slices) for different architectural components, the parameters defining the heuristic can be easily customized, hence making it a flexible approach.

6.2 Experimental Setup

For this case study, we consider a model instantiated from the rSesame framework for the Molen reconfigurable architecture. The detailed descriptions of the instantiated model and the Molen architecture have been provided in Section 5.2 and 5.1 respectively. In this case study, the previously described Molen model is used to evaluate the different task mapping heuristics described in Section 6.1. With the Molen model, an extended MJPEG application as shown in Figure 6.1 is mapped onto the Molen reconfigurable architecture. We incor-

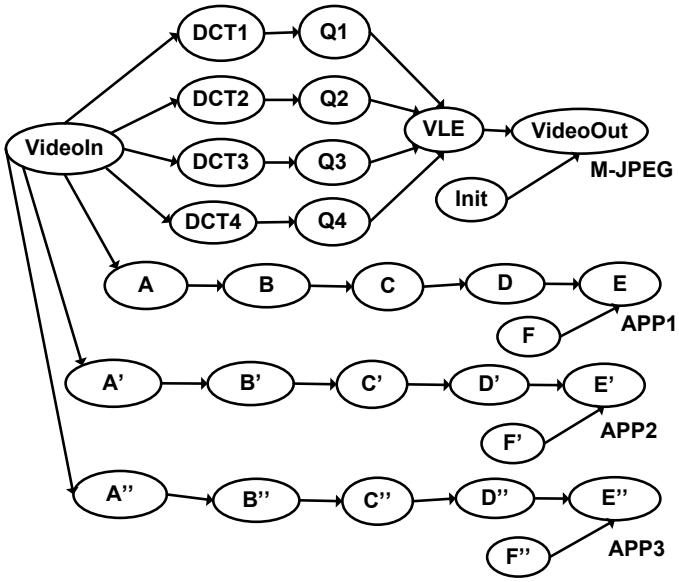


Figure 6.1: The Motion-JPEG (MJPEG) application Model considered for the case study. The MJPEG application is extended by injecting sporadic applications in each frame.

porate the aforementioned task mapping heuristics as strategies for the Molen model to perform runtime task mapping of the extended MJPEG application onto the Molen architecture. We conduct an evaluation of these task mapping heuristics based on various system attributes recorded from the model. In the rest part of this chapter, we discuss the case study in more detail.

We extend a Motion-JPEG (MJPEG) encoder application to use it as an application model for this case study. The corresponding KPN is shown in Figure 6.1. The frames are divided into blocks, and each task performs a different function on each block as it is passed from task to task. MJPEG operates on these blocks (partially) in parallel. A random number (0 to 3) of applications (APP1 to APP3) are injected in each frame of the MJPEG application in order to create a dynamic application behavior. These applications are considered as sporadic ones, which randomly appear in the system, and compete with MJPEG for the resources.

The model instantiated for this case study consists of 30 CCUs. Thus, each task is mapped onto one CCU. The mapping of the tasks onto CCUs is given in Table 6.1, where the odd rows list all the tasks and the even row list

Task	DCT1	DCT3	DCT2	F'	DCT4	Q1
CCU	CCU0	CCU1	CCU2	CCU3	CCU4	CCU5
Task	Q2	Q3	Q4	C	B	F
CCU	CCU6	CCU7	CCU8	CCU9	CCU10	CCU11
Task	Init	VideoIn	A	VLE	D	Vout
CCU	CCU12	CCU13	CCU14	CCU15	CCU16	CCU17
Task	E	E'	D'	B'	A'	C'
CCU	CCU18	CCU19	CCU20	CCU21	CCU22	CCU23
Task	E"	D"	C"	B"	A"	F"
CCU	CCU24	CCU25	CCU26	CCU27	CCU28	CCU29

Table 6.1: Mapping of tasks onto CCUs. Odd rows in the table list all the tasks, and even rows list their corresponding CCUs.

their corresponding CCUs. Note that the number of CCUs is a parameter that can be defined based on the number of pageable and HW tasks. For this case study, we consider all tasks as pageable to fully exploit the runtime mapping by deciding *where* and *when* to map them at runtime depending on the system condition. Therefore, for all the tasks, mapping decisions are performed at runtime. The model allows dynamic partial reconfiguration, and therefore, if the FPGA cannot accommodate all tasks at once, the latter can be executed after runtime reconfiguration.

We study and evaluate different task mapping heuristics from various domains by considering, for the same architecture model, different FPGA sizes. We consider six FPGAs from the Xilinx Virtex-4 FX family [36], namely XC4VFX12, XC4VFX20, XC4VFX40, XC4VFX60, XC4VFX100 and XC4VFX140. These FPGAs have different available area (slices) as shown in Table 6.2. As a result, they are used to evaluate the runtime task mapping under different resource conditions. We assume the Processor Local Bus (PLB) of these FPGAs is 4 bytes wide, and the Internal Configuration Access Port (ICAP) functions at 100 MHz, thus, its configuration speed is considered at 400 MB/sec [128].

The main purpose of this case study is to demonstrate that the rSesame framework can be used to evaluate different task mapping heuristics at runtime, under different resource conditions. As a result, we use estimated values of the computational latency, the area occupancy (on the FPGA) and the reconfiguration delay for each CCU. The computational latency values for the GPP model are initialized using estimates obtained from literature [2, 126] (but non-Molen specific). As mentioned before, we estimate the area occu-

Hardware	Area (Slices)
XC4VFX12	5472
XC4VFX20	8544
XC4VFX40	18624
XC4VFX60	25280
XC4VFX100	42176
XC4VFX140	63168

Table 6.2: Available area (in slices) for different FPGAs for the Xilinx Virtex4 FX family [36].

pancy for each of the Kahn processes mapped onto the CCU using the Quipu model [120]. Quipu establishes a relation between hardware and software, and it predicts FPGA resources from software code, typically C-level description of an application, using partial regression. Kahn processes contain functional C-code together with annotations that generate events such as R, X and W. Therefore, the Quipu model can be employed to estimate the area occupancy of each process.

Based on the reconfiguration delay of each FPGA, and estimated area of each Kahn process, we computed the reconfiguration delay of each CCU using the following equation:

$$T_{Recon} = \frac{\text{CCU slices}}{\text{FPGA slices}} \cdot \frac{\text{FPGA bitstream}}{\text{ICAP bandwidth}} \tag{6.4}$$

where CCU slices is the total number of area slices a CCU requires, FPGA slices is the total number of slices available on a particular FPGA, FPGA bitstream is the bitstream size in MBs of the FPGA and ICAP bandwidth is the ICAP configuration speed. As a final remark, we assume that there is no delay associated with the runtime mapping, such as task migration and context switching.

6.3 Heuristics Evaluation

In this section, we provide a detailed analysis of results and their implications for the aforementioned case study. We conducted a wide variety of experi-

ments on the above-mentioned task mapping heuristics with the Molen architecture by considering various FPGAs of different sizes. We evaluated and compared these heuristics based on the following parameters: execution time, number of reconfigurations, percentage of hardware/software executions and reusability efficiency. The detailed description of these parameters has been provided in Section 5.5. In the rest of this section, we discuss the evaluation results by using these parameters in more detail.

6.3.1 Execution Time

Figure 6.2 depicts the results of running different task mapping heuristics for mapping an extended MJPEG application onto the Molen architecture with various FPGAs of different sizes. The primary y-axis (left) in the graph represents the application execution time measured for each heuristic. The software only (SWonly) execution is measured when all the tasks are mapped onto the GPP. Similarly, the hardware only (HWonly) execution is measured when all the tasks are mapped onto the FPGA[1]. The static execution (STonly) is measured when only static exploration is performed. In STonly execution, a fixed set of hardware tasks are considered for the FPGA mapping and this set does not change during the application runtime. For this experiment, tasks considered as fixed hardware are DCT1-DCT4 and Q1-Q4. This implies the corresponding CCUs (i.e. CCU0 to CCU8) of these tasks are mapped onto the FPGA, and the rest are mapped onto the GPP. The secondary y-axis (right) in Figure 6.2 represents the application speedup for each heuristic compared to the SWonly execution. The x-axis lists different types of FPGAs which are ranked (from left to right) based on their sizes, such that XC4VFX12 has the smallest number of area slices and XC4VFX140 has the largest number of area slices (see Table 6.2). Several observations can be made from Figure 6.2 in terms of FPGA resources and speedup for different heuristics.

A first observation that can be noticed from Figure 6.2 is that the application performance is proportional to the FPGA size: the bigger the available area in the FPGA the higher the application performance. In the case of XC4VFX12, there is no significant performance gain by using any heuristic compared to the software execution. Since there is a limited area, only few tasks can be mapped onto the FPGA, thus, performance is constrained. Never-

[1] In HWonly, tasks are forced to be executed on the FPGA. However, if the task does not fit on entire FPGA, the task is executed with the GPP.

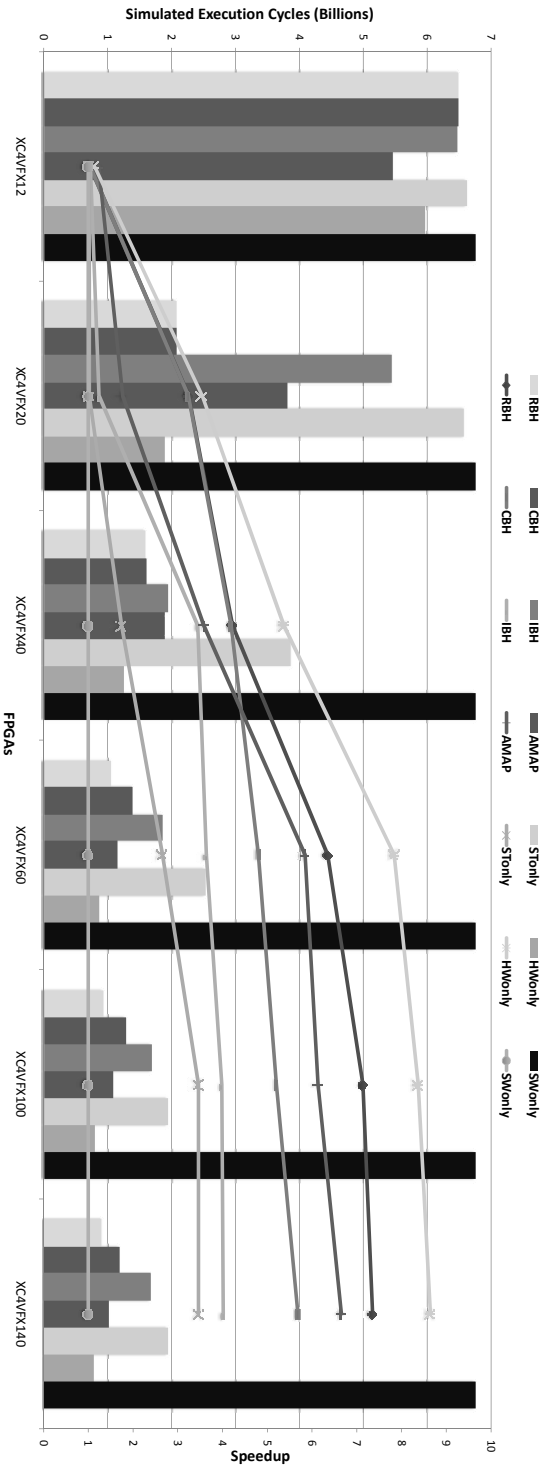


Figure 6.2: Comparison of the different heuristics tested in the proposed case study under different FPGAs conditions in terms of simulated execution time with corresponding application speedup. The application performance is proportional to the FPGA size. HWonly mapping has the best performance followed by RBH, AMAP, CBH and IBH. STonly has the worst performance.

Heuristics	Performance Increase (%)	
	XC4VFX12 ⇒ XC4VFX20 [has 54% slice increase]	XC4VFX100 ⇒ XC4VFX140 [has 33% slice increase]
HWonly	67.9	3.14
STonly	0.69	0.007
AMAP	30	7.7
IBH	15	0.87
CBH	67	8.5
RBH	70	2.8

Table 6.3: The performance increase in different heuristics with corresponding area increase in the FPGA. There is no linear increase in the performance with the increase in area.

theless, there is a notable performance improvement with other FPGAs.

Secondly, comparing the results of different heuristics for different FPGAs in the Figure 6.2, we observe that there is no linear increase in the performance with the increase in the FPGA area. For instance, although XC4VFX20 has 54% more slices than XC4VFX12, the corresponding increase in the application performance is 67.9%, in the case of HWonly, as shown in Table 6.3. Similarly, there is 33% increase in area slices while comparing XC4VFX140 with XC4VFX100 in Table 6.2. Nevertheless, there is considerably lower increase in the performance in this case, as compared to the former case. The performance increase with corresponding area increase in XC4VFX12 and XC4VFX20 as compared to XC4VFX100 and XC4VFX140 respectively, in case of different heuristics, is reported in Table 6.3. The table depicts that there is indeed no linear increase in the performance with area increase. This implies that the performance increase in an application is bounded by the degree of parallelism in that application. The use of more resources does not always guarantee a better application performance.

Another observation that can be made from Figure 6.2 is in terms of application performance of each heuristic. As it can be seen from the figure, STonly has the worst application performance, and HWonly has the best application performance. HWonly executes *all* tasks on the FPGA, and as a result, has approximately upto 9 times better performance than SWonly. STonly executes only a set of tasks on the FPGA, and mapping optimizations cannot be performed at runtime, and as a result has only upto 3 times better performance than SWonly. On the other hand, with runtime heuristics such as AMAP, IBH,

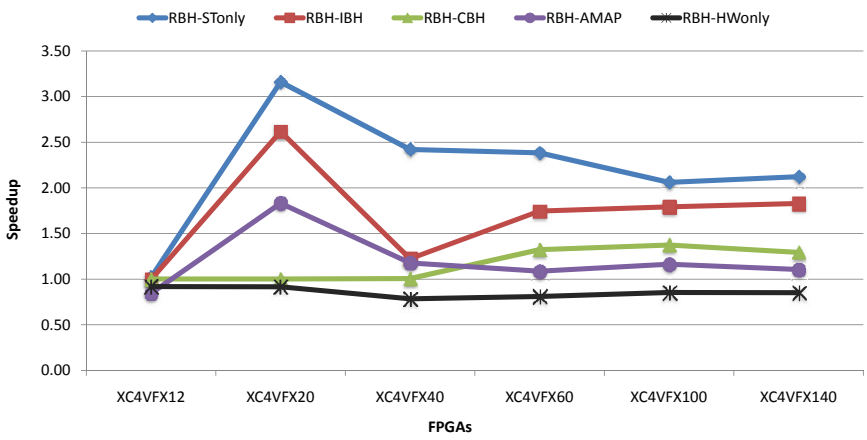


Figure 6.3: The performance increase of the RBH as compared to HWonly, STonly, IBH, CBH and AMAP. RBH performs better than AMAP under all resource conditions expect XC4VFX12. RBH performs better than STonly, IBH and CBH under all resource conditions.

CBH and RBH, the task mapping is performed at runtime. When the application behavior changes due to the arrival of a sporadic application, task mapping is optimized, and better performance can be obtained in latter cases. This can be clearly seen in the figure, where the performance of the other heuristics, such as RBH, CBH, IBH and AMAP, has ranged between HWonly and STonly.

While comparing the application performance of RBH against other heuristics, such as AMAP, IBH and CBH, we observe that, RBH provides the best performance. RBH outperforms IBH under all resource conditions. RBH performs similar to CBH in the case of XC4VFX12, XC4VFX20 and XC4VFX40, while it performs better than CBH for the rest of the FPGAs. Task mapping is highly influenced by the task selection criteria and the FPGA size. CBH chooses a task with the highest SW/HW latency difference and executes it to the FPGA. RBH also maps tasks based on the speedup factor, but the major difference is in the way this value is calculated. RBH calculates the speedup value at runtime taking into account the past execution history, while with CBH the SW/HW value is calculated statically. This difference significantly influences the performance of these heuristics. The performance increase of the RBH as compared to HWonly, STonly, IBH, CBH and AMAP is reported in Figure 6.3. As it can be inferred from the figure, the performance improvement of the RBH compared to AMAP shows an irregular behavior. The RBH performs 10% worst than AMAP for XC4VFX12. However, the improvement

significantly increases for XC4VFX20. For XC4VFX40, the improvement suddenly decreases to 10%. The improvement is regained for XC4VFX60, and stays identical for XC4VFX100 and XC4VFX140. AMAP performs task mapping based on the area availability in an ad-hoc manner, in the sense that it tries to map as many tasks as possible at once. However, the RBH performs a selective task mapping based on the task speedup and the hardware configuration reuse. When area is limited, as in the case of XC4VFX12, not many hardware configurations can be preserved in the FPGA, thus, configuration reuse cannot be exploited with the RBH. As a result, AMAP performs better than RBH. With the increase in area, many hardware configurations can be preserved in the FPGA. Consequently, the RBH performs better than AMAP.

6.3.2 Number of Reconfigurations

Figure 6.4 depicts an overview of the number of reconfigurations for different heuristics, by considering different FPGAs. Several observations can be made from Figure 6.4 in terms of FPGA resources and the number of reconfigurations for the different heuristics. Only few tasks can be executed on the FPGA with limited area slices, contributing to the small reconfiguration counts. When area slices increases, more tasks executed in the FPGA, and hence reconfiguration counts increases. Nevertheless, the reconfiguration count is greatly influenced by the mapping strategies used. As it can be inferred from Figure 6.4, HWonly has relatively higher reconfigurations as compared to other heuristics. With HWonly, *all* tasks are executed to the FPGA, and hence they are configured frequently. In large FPGAs, there is a possibility for CCUs to save and re-use their configurations, and hence avoid reconfiguration. Therefore, reconfigurations saturates with large FPGAs. Similarly, STonly has a relatively low number of reconfigurations with small FPGAs, such as XC4VFX12 and XC4VFX20. The reconfiguration count increases in case of XC4VFX40 and XC4VFX60, and then it stays constant in all other cases. STonly executes a fixed set of HW tasks in all cases, since the number of HW task is constant, the reconfiguration also saturates.

We can observe from Figure 6.4 that AMAP has significantly higher reconfiguration counts unlike the other heuristics. AMAP performs task mapping based on the area availability in an ad-hoc manner, in the sense that any task can be mapped onto the FPGA. This leads to a significant increase in reconfiguration counts. It is worth noticing that the application performance in

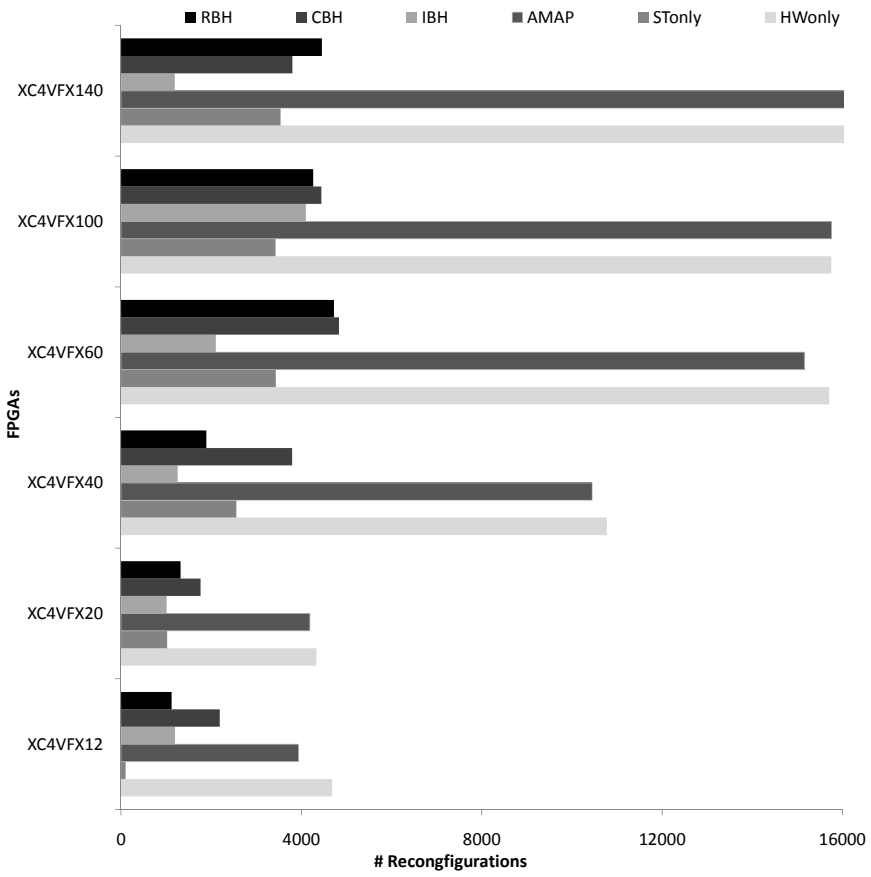


Figure 6.4: Heuristics comparison under different FPGAs conditions in terms of number of reconfigurations. There is a direct relation between the number of reconfigurations and the FPGA area.

case of AMAP does not decrease drastically with the higher reconfiguration numbers. We may expect a significant performance decrease due to massive reconfigurations. The reconfiguration latency considered for a task is relatively small compared to the HW execution latency. Despite the larger number of reconfigurations, the performance can be considerably improved with the HW execution in such cases. Similarly, in the case of CBH, the reconfiguration counts are lower in smaller FPGAs due to lower hardware executions. This number increases with large FPGAs. There are no significant changes in the reconfiguration counts with the increase in area slices once sufficient area is available.

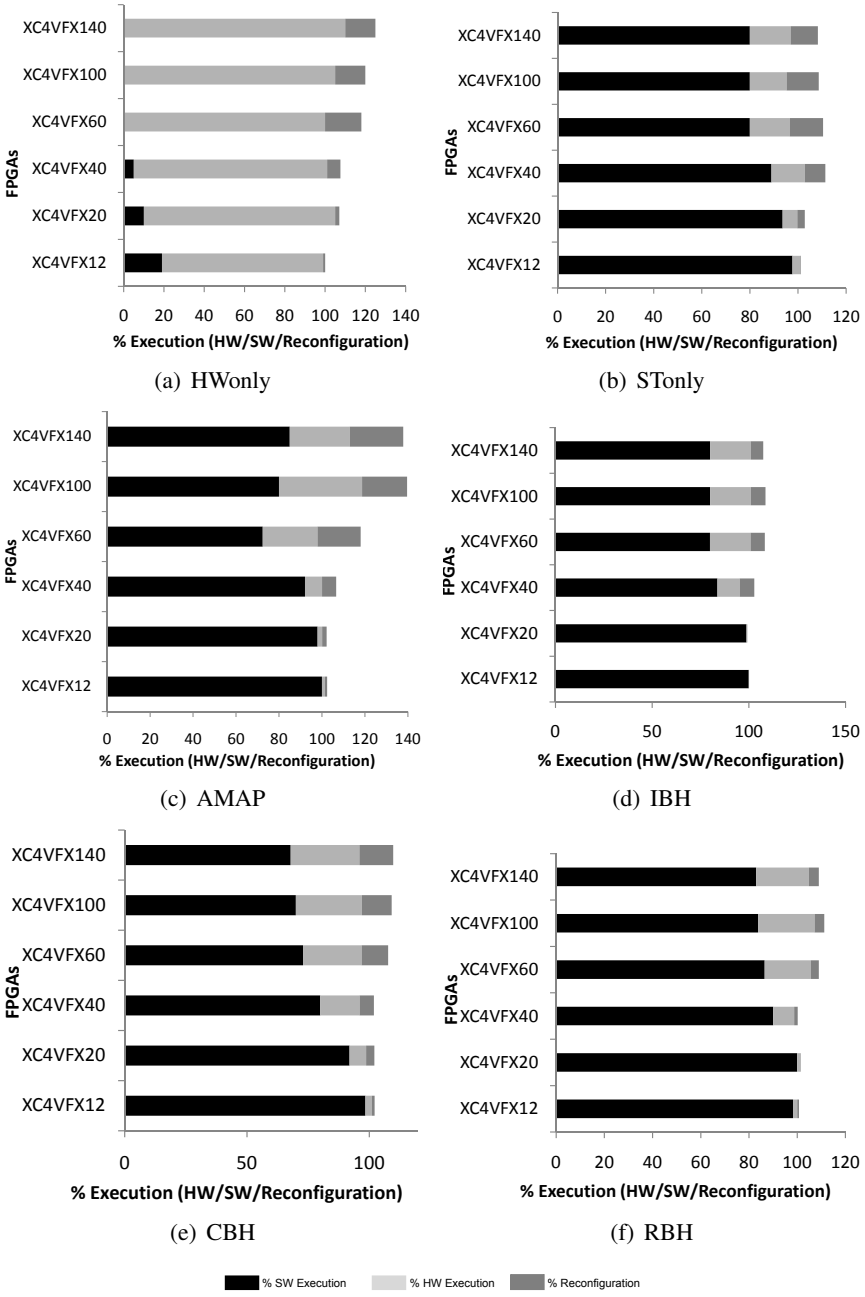
The number of reconfigurations for IBH is somewhat lower compared to the other heuristics, such as AMAP, CBH and RBH under all FPGA conditions. This is not due to an efficient algorithm which tries to optimize the reconfiguration delay, rather it is the effect of limited HW execution. In case of IBH, the mapping decision is changed only in the beginning of each interval, and the mapping behavior is fixed within an interval. Thus, a fixed set of tasks is mapped onto the FPGA during such an interval. This limits the hardware execution percentage, and hence the reconfigurations. On the other hand, RBH reuses the hardware configurations to reduce the total number of reconfigurations. As a result, we observe a lower number of reconfigurations in case of RBH compared to CBH and AMAP in Figure 6.4. Note that IBH and STonly have lower reconfigurations than RBH as a consequence of their lower hardware execution. Nonetheless, RBH has a better reconfiguration-to-HW-execution ratio as compared to IBH and STonly, making the former better in terms of performance.

6.3.3 Percentage of Hardware Execution, Software Execution and Reconfiguration

Figure 6.5 shows the comparison between different task mapping heuristics in terms of hardware execution, software execution and reconfiguration measured using the equations 5.2, 5.3 and 5.4 respectively. The x-axis in the graph is stacked as 100%, and it shows the contribution of hardware execution, software execution and reconfiguration to the total execution time. We observe that in few FPGAs the percentage of execution is greater than 100%. The hardware execution percentage measured in Equation 5.2 is provided as an upper bound to address the parallel execution possibility of the FPGA, as a result, its value can go beyond the 100% limitation.

A first observation that can be made from Figure 6.5 is in terms of execution percentage and FPGA area. The limited area slices in the FPGA confines the HW execution percentage in smaller FPGAs. The hardware execution percentage increases considerably with more area slices, but this increase is not linear. As it can be seen from the figure, hardware execution percentage somewhat saturates with large FPGAs, such as XC4FX100 and XC4FX140. This observation is valid for all the runtime mapping heuristics including STonly and HWonly.

With HWonly mapping, *all* tasks are forced to be executed to the FPGA.



(a) HWonly (b) STonly

(c) AMAP (d) IBH

(e) CBH (f) RBH

(g) Legend

Figure 6.5: The comparison of different heuristics based on percentage of hardware execution, software execution and reconfiguration. The hardware execution percentage is low in smaller FPGAs, and it increases considerably with more area slices.

However, if the task does not fit on the entire FPGA, then the task is executed with the GPP. Therefore, in Figure 6.5, we observe certain percentage of software execution with small FPGAs, but with larger FPGA, there is only HW execution and the corresponding reconfiguration. With smaller FPGAs, almost no tasks are executed in hardware, and as a result, STonly has very minimal hardware execution (if any) and the corresponding reconfiguration. With the larger FPGAs, STonly has a relatively good but constant hardware execution and reconfiguration percentage, since it executes a fixed set of tasks on the FPGA.

While comparing the runtime heuristics, such as AMAP, CBH, IBH and RBH, we can observe that AMAP has the best hardware execution percentage in larger FPGAs, followed by RBH and CBH. CBH and RBH somehow shows similar behavior in terms of hardware execution percentage. However, in case of reconfiguration percentage, they do not follow the same trend. The reconfiguration is somewhat linear to the hardware execution in case of CBH. However, RBH does not show any linear increase in reconfiguration with hardware execution. RBH performs task mapping based on configuration re-use, and as a result, tries to avoid reconfiguration with more hardware executions. This behavior of RBH heuristic is apparent in the figure, especially in the case of moderate to large FPGAs, such as XC4FX60, XC4FX100 and XC4FX140. IBH follows a behavior similar to STonly in terms of software and hardware execution, as it also executes a fixed set of tasks on the FPGA.

By mapping more tasks onto FPGA, the application can be accelerated, but it also has reconfiguration overhead. The efficiency of the mapping heuristics lies in finding the best mapping while minimizing the number of reconfigurations. Nevertheless, in Figure 6.5, we see almost a linear contribution of the reconfiguration overhead to the total execution time in all heuristics except in RBH. This phenomenon is highly influenced by the policy implemented for task mapping. Another observation that can be made from the figure is the contribution of the hardware execution, SW execution and reconfiguration to the total execution time. The figure shows that the GPP executes most of the application, and the FPGA computes only less than 40% of the total application. This is due to the architectural restrictions of the Molen architecture - the GPP and the RP run in a mutual exclusive way, due to the processor/co-processor nature of the architecture. This influences the mapping decision, which in turn contributes to the low hardware execution rates. This significantly increases the total execution time. Another reason for the lower percentage of hardware execution is due to the lower hardware latency for each task. The execution

percentage is calculated as the ratio of execution latency of all tasks to the total execution time of an application. The hardware latency is comparatively lower than the SW latency for each task. Therefore, the corresponding hardware execution contribution is always lower as compared to the percentage of SW execution.

6.3.4 Time-Weighted Area Usage

Figure 6.6 depicts an average time-weighted area usage measured using the equation provided in Equation 5.5 for different heuristics under different FPGA devices. The primary y-axis (left) in the graph represents the time-weighted area usage measured for each heuristic. The secondary y-axis (right) in the figure represents the application speedup for each heuristic compared to the SWonly execution. Several observations can be made from Figure 6.2 in terms of FPGA resources and time-weighted area usage of different heuristics. The first observation that can be made from the figure is in terms of time-weighted area usage and the hardware resource. As it can be seen from the figure, the time-weighted area usage is directly impacted by the number of area slices in the FPGA. With the limited area slices in small FPGAs, few tasks are executed in the FPGA, contributing to a smaller number of hardware executions. This, in turn, contribute to the lower area usage. With sufficient area slices, there is a considerable number of hardware executions, and hence the area usage is high. Nonetheless, there is no linear relation between the time-weighted area usage and the available FPGA area. In XC4VFX140, the area usage is relatively low compared to XC4FX100, despite the fact that area slices is greater in the former. The area usage measured is the time weighted factor, and it depends on the hardware execution, the total FPGA area and the total execution time, as shown in Equation 5.5. The increase in the area slices, with no significant increase in hardware executions, contribute to the lower area usage in the former case.

As it can be inferred from Figure 6.6, HWonly has the highest time-weighted area usage under all FPGA conditions. HWonly executes *all* tasks onto the FPGA, and as a result, the cost of using FPGA in this case is higher than all other heuristics. STonly, however, has the lowest area usage due to its lower number of hardware executions, and therefore, its corresponding performance is also very poor. Similarly, AMAP has higher area usage compared to other heuristics, such as CBH, IBH and RBH, under all FPGA conditions,

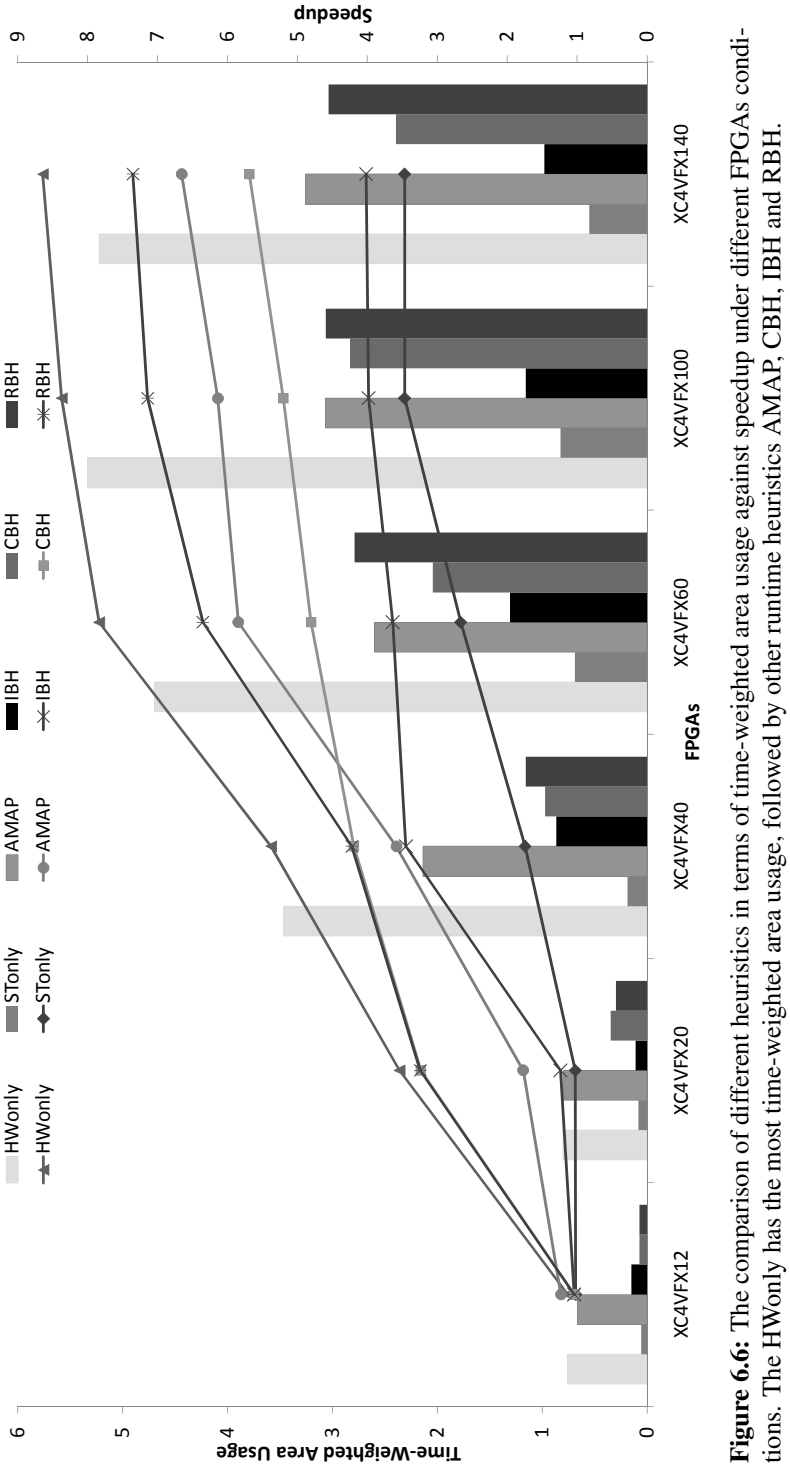


Figure 6.6: The comparison of different heuristics in terms of time-weighted area usage against speedup under different FPGAs conditions. The HWonly has the most time-weighted area usage, followed by other runtime heuristics AMAP, CBH, IBH and RBH.

except XC4VFX60. AMAP performs task mapping based on area availability. As a matter of fact, it has a relatively higher number of hardware executions compared to the other heuristics, and therefore it consumes additional area. RBH, on the other hand, has less time-weighted area usage. While comparing AMAP and RBH, we can observe that RBH performs somewhat better than AMAP in terms of performance. This implies that the RBH reuses the hardware configuration already present in the FPGA to avoid reconfiguration overhead, and as a result it can give better performance with the same amount of hardware resources as required by AMAP. Likewise, CBH has a comparable percentage of time-weighted area usage, but it lags behind in terms of speedup as compare to RBH. However, IBH has a considerably low percentage of area usage, as it also has lower hardware executions due to the constantly executed HW task set, and hence it also has lower performance. We can summarize that HWonly has the best performance, but consumes more hardware resources. STonly has the lowest area usage, but straggle behind in terms of performance. A tradeoff in terms of performance and resources can be obtained with task mapping at runtime, which performs selective task mapping onto the FPGA at runtime.

Another compelling observation that can be made from Figure 6.6 is the lower value of the time-weighted area usage. The Molen architecture is based on processor/co-processor paradigm, as a result, the GPP and the reconfigurable processor run in a mutual exclusive. This contributes to the lower number of hardware executions, which consequently increases the total execution time. Thus, these two factors significantly contribute to the low value of area usage. The area usage can be increased either by mapping more tasks onto the FPGA or by operating the reconfigurable processor and the GPP in parallel.

6.3.5 Reusability Efficiency

Figure 6.7 depicts the Reusability Efficiency (RE_{Task}) recorded for all CCUs using equation provided in Equation 5.6 for different heuristics under different FPGA conditions. Several observations can be made from the figure in terms of FPGA area and the RE_{Task} of each CCU. Firstly, we observe that the CCU reuse is significantly affected by the number of area slices in the FPGA. Small FPGAs, such as XC4VFX12 and XC4VFX20, have many CCUs with RE_{Task} value zero. A CCU has an RE_{Task} value of zero under the following conditions:

- when a CCU is always mapped onto the GPP or

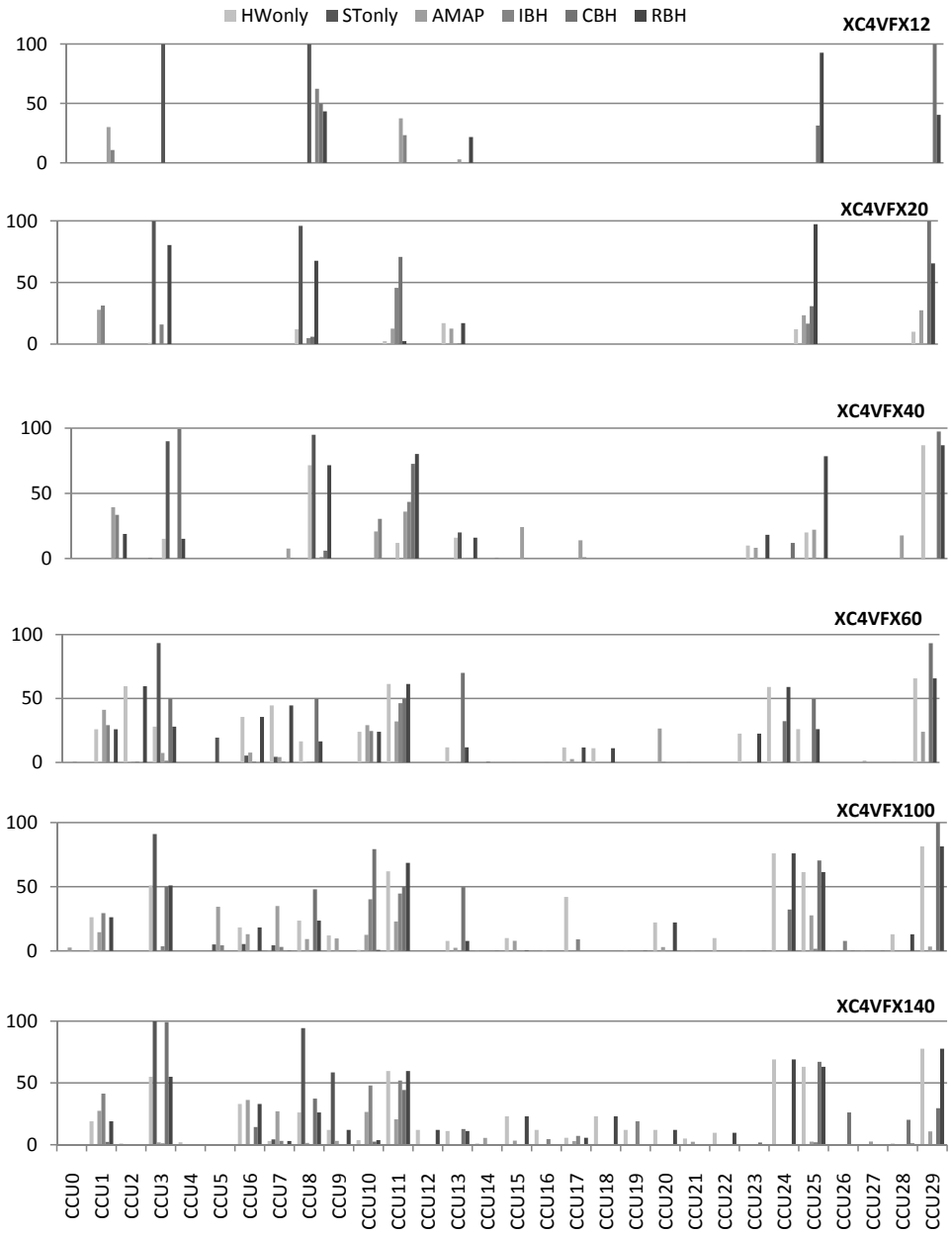


Figure 6.7: Reusability Efficiency (RE_{task}) of CCUs for different heuristics under different FPGA conditions. The CCU reuse is significantly affected by the number of area slices in the FPGA.

- when a CCU is configured every time it is executed to the FPGA.

With few resources in the FPGA, only a limited number of tasks can be executed to the FPGA. Additionally, in such cases, hardware configurations cannot be preserved for future reuse. As a result, CCUs have an RE_{Task} value of zero. Moreover, in this case, CCUs that are reused have a very small size in terms of area. With the increase in number of slices in the FPGA, more CCUs are reused. Medium sized FPGAs, such as XC4VFX40 and XC4VFX60, reuse more CCUs compared to smaller FPGAs, but in such cases, the reuse percentage is still low. With the larger FPGAs such as XC4VFX100 and XC4VFX140, more CCUs are reused with large RE_{Task} value.

Table 6.4 reports the mapping of CCUs onto the FPGA for different heuristics under different FPGA conditions. From the table, we observe that different heuristics map different CCUs onto the FPGA. As a result, different heuristics reuse different set of CCUs. Nevertheless, there are few CCUs such as CCU3 CCU8, CCU11 and CCU13 which are reused in many cases. These CCUs have a small area requirement. Consequently, their hardware configurations can be preserved even with smaller FPGAs.

As it can be inferred from Figure 6.7, HWonly has the best RE_{Task} for many CCUs in large FPGAs, such as XC4VFX100 and XC4VFX140. HWonly executes all the tasks on the FPGA, and as a result, it has high hardware execution count. However, with small FPGAs, all the tasks are configured, due to area restrictions, and there is no configuration re-use. On the other hand, with larger FPGAs, more configurations are saved and re-used, and as a consequence many CCUs have a considerably high RE_{Task} value. Similarly, STonly always maps a set of fixed tasks onto the FPGA. Out of these tasks, only a few number of small tasks can be reused. We notice that, these CCUs have a relatively higher RE value compared to the ones reused with the AMAP heuristic in the figure. AMAP has higher HW execution percentage as compared IBH and CBH. As a matter of fact, many tasks are reused in case of AMAP, but the reuse percentage of these CCUs is low. AMAP has no fixed pattern for task execution, and as a result, any task can be executed in FPGA. Therefore, the re-usability is rather distributed among many CCUs. CBH, on the other hand, follows a specific policy for task execution in FPGA, and hence, executes a fixed set of selected task. As a result, a set of selected tasks is reused. Similar behavior is observed in the case of CBH, as it also executes a set of specific task within an interval, same tasks are reused (if any).

	STonly	AMAP	IBH	CBH	RBH
XC4VFX12	CCU3,CCU8	CCU1,CCU3, CCU8,CCU11, CCU13,CCU25, CCU29	CCU1,CCU8, CCU11,CCU13, CCU25	CCU3,CCU8, CCU13, CCU25, CCU29	CCU8,CCU13, CCU24,CCU25, CCU29
XC4VFX20	CCU3,CCU6, CCU8	CCU1,CCU3, CCU6,CCU8, CCU11,CCU13, CCU17,CCU25, CCU29	CCU1, CCU3, CCU6, CCU8, CCU11, CCU13, CCU25	CCU3,CCU8, CCU11,CCU25, CCU29	CCU1,CCU3, CCU8,CCU11, CCU13,CCU25
XC4VFX40	CCU2-CCU3, CCU6-CCU8	CCU0-CCU11, CCU13-CCU15, CCU17, CCU22-CCU23, CCU25, CCU28-CCU29	CCU1-CCU11, CCU13-CCU17, CCU25	CCU3,CCU8, CCU11,CCU23, CCU29	CCU1,CCU3, CCU6,CCU10, CCU11,CCU13, CCU17,CCU23, CCU25,CCU29
XC4VFX60	CCU2-CCU9	All CCUs	CCU1-CCU11, CCU13-CCU17, CCU19-CCU20, CCU25-CCU26	CCU3,CCU8, CCU10,CCU13, CCU23,CCU25, CCU29	CCU1-CCU3, CCU8,CCU10, CCU11,CCU13, CCU23,CCU25, CCU29,
XC4VFX100	CCU2-CCU9	All CCUs	CCU1-CCU11, CCU13-CCU17, CCU19-CCU20, CCU25-CCU26	CCU2,CCU3, CCU6,CCU8, CCU10,CCU11, CCU13,CCU23, CCU25,CCU29	CCU1-CCU4, CCU6,CC8, CCU10,CCU11, CCU13,CCU14, CCU17,CU23, CCU28,CCU29
XC4VFX140	CCU2-CCU9	All CCUs	CCU1-CCU11, CCU13-CCU17, CCU19-CCU20, CCU25-CCU26	CCU1-CCU3, CCU5,CCU6, CCU10,CCU11, CCU13,CCU17, CCU23-CCU26, CCU28-CCU29	CCU1-CCU4, CCU6,CC8, CCU10,CCU11, CCU13,CCU14, CCU17,CCU23, CCU28,CCU29

Table 6.4: CCUs mapped onto the FPGA for different heuristics under different FPGA conditions. Different heuristics map different set of tasks onto the FPGA under different conditions.

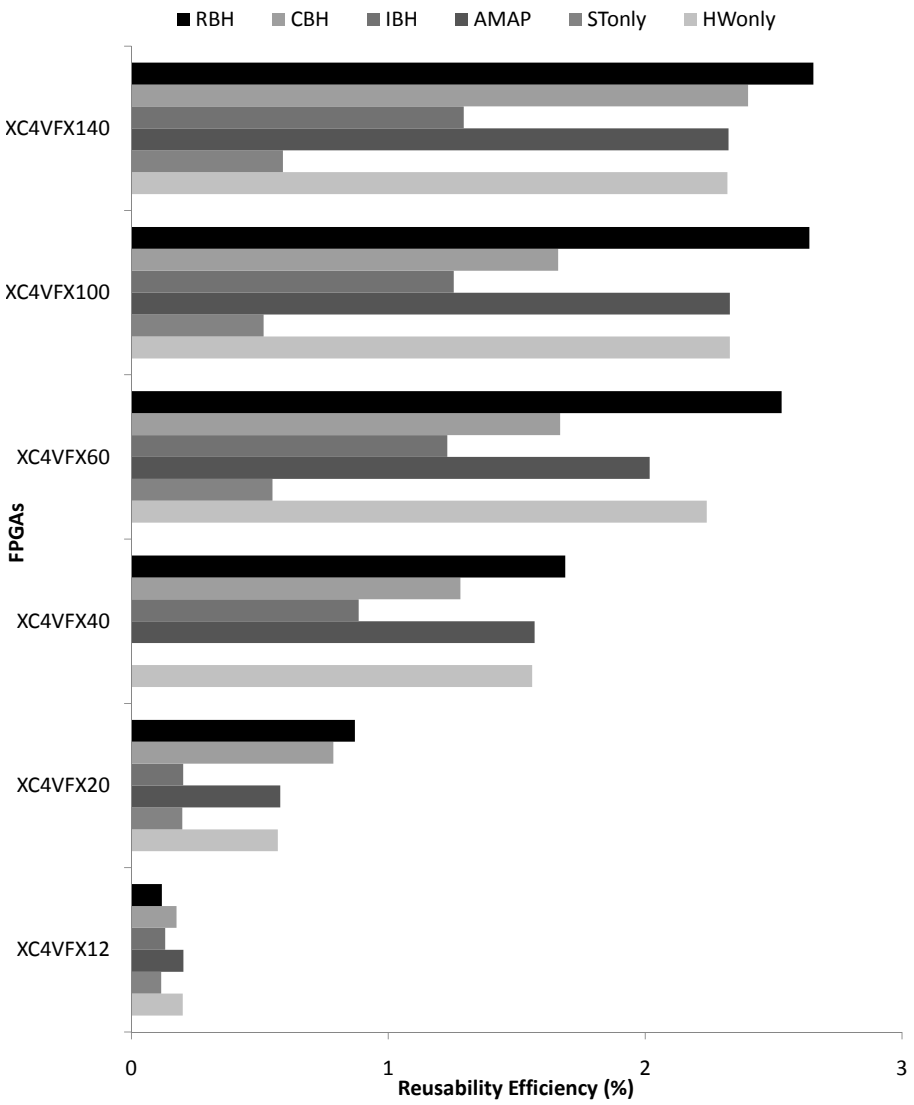


Figure 6.8: Heuristics comparison under different FPGAs conditions in terms of application Reusability Efficiency (RE_{app}). RBH has better RE_{app} compared to other heuristics.

Likewise, from Figure 6.7, we observe the RE_{Task} of RBH is better than that of other runtime heuristics for many CCUs. The impact of this hardware configuration reuse, in case of RBH, can be directly seen in terms of performance gain in Figure 6.2, where RBH has better speedup than the other heuris-

tics. From Figure 6.8, we also observe that, for few tasks, RE_{task} decreases when FPGA resources increase. With larger FPGAs, more tasks can fit onto the the FPGA. As a result, these tasks are also map onto the FPGA, thus over writing the saved configurations of other tasks. RE_{task} for few tasks decreases in the FPGAs with moderate size. With the abundant resources, the hardware configuration can be saved for more tasks, and RE_{task} increases again.

Note that, STonly, AMAP, CBH, IBH do not map the task based on the hardware configuration reuse. The reuse obtained in the case of STonly, AMAP, CBH and IBH is a default value determined based on the arrival of the application task. If a CCU is already configured on the FPGA, when its corresponding task arrives, the task can be executed without reconfiguration. However, the RBH reuses more hardware configurations than the other heuristics on top of the default value obtained.

Figure 6.8 depicts the total RE_{app} recorded using Equation 5.7 for different heuristics under different resource conditions. In the figure, we again observe that the reusability increases when using larger FPGAs. HWonly execute *all* the tasks in FPGA, and therefore there can be a possibility many of these tasks are re-used when sufficient area is available, resulting into higher RE_{app}. STonly has almost a constant RE_{app} in larger FPGA, since it executes a constant set of tasks in FPGA. While comparing runtime heuristics, such as AMAP, CBH, IBH and RBH, we can observe that since the RBH has more CCUs reused than other heuristics as shown in Figure 6.5(f), RBH has a better RE_{app} value than other heuristics in all resource conditions but XC4VFX12. Since XC4VFX12 has less area, all the heuristics have approximately the same value for RE_{app}. RE_{app} is the accumulation of the time saved due to hardware configuration reuse of each CCU. If all CCUs obtain the same value of the RE for a task mapping heuristic, then the application RE_{app} depends on the corresponding total execution time of that heuristics.

6.3.6 Observations and Recommendation

The case study shows that the rSesame framework is flexible and can efficiently assess various runtime mapping heuristics in terms of various design parameters. Based on the evaluation discussed in the previous section, we can summarize the following observations:

- Comparison of different FPGAs shows that with very limited resources

(in case of the small FPGAs), the number of tasks that can be mapped onto the FPGA is low. Consequently, tasks are mapped onto the GPP. This leads to the poor application performance.

- More resources (in case of moderate/higher FPGAs) imply more tasks mapped onto the FPGA. Consequently, we can obtain better application performance.

- Runtime mapping provides better performance compared to static mapping in case of dynamic application/architecture conditions. If the application behavior is well known in advance, static mapping can give equal performance.

- Mapping all the tasks onto the FPGA gives better performance, but it consumes more hardware resources. Runtime mapping performs task mapping based on the runtime system conditions, as a matter of fact, with the runtime mapping, a tradeoff can be obtained in terms of performance and resources.

- Comparing different heuristics, in case of limited resources conditions (small FPGAs), the ad-hoc task mapping of AMAP performs better as compared to CBH, IBH and RBH. The careful task selection with RBH, CHB and IBH cannot be fully exploited in such cases, due to limited resources.

- The reuse of hardware configurations is better in case of sufficient resource conditions (medium to large FPGAs). As a result, the configuration reuse can be well exploited. Additionally, the RBH provides better application performance than AMAP and CBH.

- In case of abundant resource conditions (very large FPGAs), the performance saturates due to application constraints. Under such scenarios, all the heuristics have similar performances.

6.4 Conclusions

In this chapter, we evaluated the rSesame framework's characteristics by showing that it can easily and quickly model, evaluate and explore a wide range of application mapping heuristics, under different architectural conditions. We

described a case study in which we tested four different runtime mapping heuristics from diverse domain by using the rSesame framework. The obtained results show that the rSesame framework can be used to efficiently perform rapid exploration and evaluation of various task mapping heuristics based on several architectural parameters, such as execution time, number of reconfigurations, time-weighted area usage, percentage of hardware/software execution, percentage of reconfiguration and hardware reusability efficiency from the framework. These explorations were performed under different architectural conditions, where different FPGAs were used to evaluate a number of application-to-architecture mappings under different resource conditions. This indicated that the presented framework can be efficiently used as a standard platform to facilitate easy comparison between various evaluations, such as application-to-architecture mappings, task mapping strategies, and architectural conditions.

We showed that the rSesame framework can be efficiently used to perform system-level DSE by performing quick exploration of several design parameters of reconfigurable architectures, both statically and at runtime. In the next chapter, we summarize the major conclusion of this dissertation. We list the main contributions and present open issues and future directions.

Note. The content of this chapter is based on the the following articles:

K. Sigdel, M. Thompson, C. Galuzzi, A.D. Pimentel, K.L.M. Bertels, **Runtime Task Mapping Based on Hardware Configuration Reuse**, *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig'10)*, Cancun, Mexico, December 2010, pp. 1-6.

K. Sigdel, M. Thompson, C. Galuzzi, A.D. Pimentel, K.L.M. Bertels, **Evaluation of Runtime Task Mapping Heuristics with *rSesame* - A Case Study**, *Proceedings of the Design, Automation and Test in Europe (DATE'10)*, Dresden, Germany, January 2010.

7

Conclusions

In this dissertation, we have developed a system-level framework which can assist designers in tackling various challenges while designing reconfigurable systems. The framework can be employed to perform early stage DSE by rapid exploration of several parameters, such as architectural characteristics, hardware-software partitionings, application-to-architecture mappings and scheduling strategies. The framework can model simulate and evaluate reconfigurable architectures combining static exploration together with runtime exploration in order to provide an efficient DSE. In the previous chapters of this dissertation, we discussed the methodology behind the framework and studied its key features. We employed the framework to perform quick exploration of several application-to-architecture mappings for reconfigurable architecture, both statically and at runtime. Furthermore, we also used the framework to evaluated various task mapping heuristics at runtime. In this chapter, we summarize the major conclusions of this dissertation and lists future directions.

This chapter unfolds in three sections. Section 7.1 summarizes the main contribution of this dissertations. In Section 7.2, we present the main conclusions of this dissertation. Finally, Section 7.3 lists future directions and open issues worthy of further research and investigation.

7.1 Summary of Contributions

This dissertation started in Chapter 2 by providing the necessary overview of the current state-of-the-art in the context of reconfigurable systems and the

system-level design methodology. More specifically, the chapter discussed the major challenges while designing heterogeneous reconfigurable systems. The use of heterogeneous and reconfigurable resources significantly enlarges the design space of modern systems, as a result, it is essential to perform DSE at the early design stages in order to efficiently investigate tradeoffs between various choices such as hardware-software partitioning, architecture-to-application mapping, task scheduling and task allocation. System-level modeling and design methodology allows fast and efficient traversal of a large design space. As a result, providing a system-level framework allows modeling and simulation of reconfigurable systems in an efficient manner.

In this dissertation, we presented a two-level mapping exploration approach for performing DSE of reconfigurable architectures. The presented approach combines a detailed static exploration together with a high quality runtime exploration. At the first level, static exploration identifies a set of mappings. At the second level, these mappings are optimized at runtime to address any changes in the application, in the architecture or in the environment. In Chapter 3, we provided an outline of two-level mapping exploration for reconfigurable architectures. The chapter also provided a detailed discussion of the static and the runtime mapping exploration. In Chapter 4 of this dissertation, we present a framework which can perform the two-level mapping exploration.

rSesame is a generic modeling and simulation framework for performing DSE of reconfigurable systems at the early design stages, which can model, simulate and evaluate reconfigurable systems both statically and at runtime. It employs the Sesame framework as a modeling and simulation platform for system-level DSE. The rSesame framework provides various architectural exploration parameters such as execution time, number of reconfigurations, area usage, percentage of HW/SW execution and reusability efficiency. The main features of the rSesame framework includes flexibility, ease of use, fast performance, and applicability. The rSesame framework together with its characteristics is discussed in Chapter 4 of this dissertation.

In this dissertation, we described two case studies to show the characteristics of the rSesame framework tested on the Molen reconfigurable architecture. First case study is discussed in Chapter 5 of this dissertation, in which we employed the rSesame framework to perform both static and runtime mapping exploration of the Molen reconfigurable architecture. In order to perform mapping exploration of the Molen reconfigurable architecture based on var-

ious design attributes, we instantiated a model for the Molen reconfigurable architecture using the rSesame framework. We showed that the instantiated model can be efficiently employed to perform exploration of various design parameters. Based on these design parameters, we evaluated and compared the runtime mapping exploration of the Molen architecture against its static exploration. The obtained results show that mapping all the tasks onto the reconfigurable hardware gives better performance, but consumes more hardware resources. With the runtime mapping, a tradeoff can be obtained in terms of performance and resources. The rSesame framework can explore such tradeoffs.

In another case study, discussed in Chapter 6 of this dissertation, we provided an evaluation of the rSesame framework by studying and comparing different runtime mapping heuristics based on various design attributes. We incorporated four different task mapping heuristics from various domains to perform runtime mapping of the given application onto the Molen reconfigurable architecture. In this case study, we studied and evaluated different heuristics by considering, different FPGAs for the same architecture model, in order to evaluate the task mapping under different resource conditions. The case study showed that the rSesame framework be efficiently deployed not only to evaluate different heuristics, but it can also be employed to evaluate different architectural conditions at runtime.

7.2 Main Conclusions

The main conclusions of the research presented in this dissertation can be summarized as follows.

- The use of heterogeneous and reconfigurable resources significantly enlarges the design space of modern systems. As a result, it is essential to perform DSE at the early design stages in order to efficiently investigate tradeoffs between various design choices. Such design choices include hardware-software partitioning, architecture-to-application mapping, task scheduling and task allocation. System-level modeling and design methodology allows fast and efficient traversal of a large design space in an efficient manner. Providing a system-level framework for DSE at early stages, facilitates design decisions to be made very quickly, which in turn, can significantly reduce overall design time.

- Reconfigurable systems can evolve under various runtime conditions due to changes imposed either by the architecture or by the application(s) or by the environment. In such systems, the design process becomes more sophisticated as all the design decisions have to be optimized in terms of runtime behaviors and changing system status. Performing only static exploration under such conditions often results with compromised accuracy. The two-level mapping exploration, presented in this dissertation, combines the benefit of fast static exploration with the detailed exploration at runtime. At the first level, static exploration leads to a set of mappings. After that, at second level, runtime exploration performs a high quality exploration at the runtime to optimize these mappings to address any changes in the system. Performing two-level mapping exploration provides an efficient DSE of reconfigurable systems.

- The rSesame framework is a generic system-level modeling and simulation framework which can explore and evaluate reconfigurable systems both statically and at runtime. The framework can be employed to perform two-level mapping exploration (static and runtime) for early stages DSE. Using the rSesame framework, we evaluated and compared the runtime application mapping, where task mapping can change at runtime, against the static application mapping, where task mapping is fixed. The obtained results showed that mapping all the tasks onto the reconfigurable hardware gives better performance, however, consumes more hardware resources. With the runtime mapping, a tradeoff can be obtained in terms of performance and resources. The rSesame framework can be efficiently used to investigate and appraise such tradeoffs.

- The main features of the rSesame framework includes flexibility, ease of use, fast performance, and applicability. We evaluated these features of the framework tested on a real reconfigurable architecture. We employed the rSesame framework to model, simulate and compare a wide range of runtime task mapping heuristics from various domain. Due to the fast execution times, the rSesame framework efficiently explored and/or evaluated these heuristics based on various design attributes recorded from the framework. This indicates that the presented framework is flexible, extendable and it can be efficiently used as a standard platform to facilitate easy comparison between various application-to-architecture mappings.

- The rSesame framework can also be used to evaluate different architectural conditions. In this dissertation, we also employed the rSesame framework to evaluate different architectural conditions. We explored different application-to-architecture mappings, by considering, for the same architecture model, different FPGA sizes, in order to evaluate different architectural conditions. The obtained result showed that the framework can also be used to efficiently evaluate different architecture conditions.

7.3 Open Issues and Future Recommendations

From the research presented in this dissertation, some open issues have been identified. In this section, we lists these issues, together with the possible future research directions.

- In the current status of the rSesame framework, we only consider one type of architectural resources i.e area while performing task mapping decisions for the reconfigurable architecture. A number of other detailed architectural resources, such as memory components, lookup tables, registers, DSP elements, wire segments and interconnects can be incorporated as different architectural parameters in the model. In that case, these resources can be considered as architectural parameters while performing mapping decisions. Furthermore, currently we consider simple structure for the reconfigurable hardware, and we do not consider fragmentation into consideration. It can be extended to model complex reconfigurable hardware structures, and resource fragmentation can also be measured as one of the non-functional design attributes.

- Although the rSesame framework envisions to evaluate/model various non-functional design attributes such as, performance, static or dynamic power consumption, energy estimation and memory requirements, the current version of the framework only evaluates performance. In the future, the framework can be extended to explore aforementioned attributes as well.

- The model can be extended to support the real-time behavior of applications. For this, application models in the application layer can be extended to support deadlines for different tasks. The architecture models

should then also be extended to record any deadline misses from the application model. This way the rSesame framework can model the real-time system behavior.

- The rSesame framework can be coupled with a hardware cost prediction model, such as QUIPU [120]. QUIPU is a hardware prediction model, which predicts different hardware attributes, such as hardware area, interconnect delays and hardware latency for a task when it is executed on a given reconfigurable hardware. Such cost prediction models predict different hardware attributes, such as hardware area, interconnect delays and hardware latency for a task. Linking such cost prediction models with the modeling and simulation framework, enables the framework to be more accurate in terms of architectural attributes. Furthermore, other tools, such as profilers, KPN generators and a visualization tool can also be integrated together with the rSesame framework. Such an integration will provide a complete DSE tool-flow, which can automatically perform DSE of the reconfigurable architectures at runtime starting from an input application to the output results.

- Another extension is possible in the direction of modeling and simulation of runtime behavior. The dynamic behavior of various architectural components such as, processor, memory, bus and buffer can be modeled at runtime. Such architectural components can be added or removed at runtime in order to model the behavior of the adaptive the reconfigurable systems.

- Similarly, in the current state of runtime modeling with the rSesame framework, only tasks can migrate from one processor to another. The communication channels between these tasks can also migrate from one memory resources to another depending on the architectural condition. This behavior can also be modeled with the framework. The mapping layer, for instance, can be extended to incorporate such sophisticated functionalities.

- One of the major challenges in the system-level modeling and the design is the model calibration. In the current version of the rSesame framework, the model created uses predicted values, and the calibration is left as a future work. Calibrating rSesame models requires comparing them against the real implementation data of the reconfigurable architectures. Models can be tweaked, if there is a deviation in the results.

Furthermore, the error percentage can also be reported. In this way, the correctness of the models can be guaranteed.

Bibliography

[1] J. Ou and V. K. Prasanna, "Design space exploration using arithmetic-level hardware–software cosimulation for configurable multiprocessor platforms," *ACM Transactions on Embedded Computing Systems TECS*, vol. 5, no. 2, pp. 355–382, May 2006.

[2] A. D. Pimentel, M. Thompson, S. Polstra, and C. Erbas, "Calibration of abstract performance models for system-level design space exploration," *Journal of Signal Processing Systems for Signal, Image, and Video Technology*, vol. 50, no. 2, pp. 99–114, 2008.

[3] G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria, "A design space exploration methodology supporting runtime resource management for multi-processor systems-on-chip," in *Symposium on Application-Specific Processors SASP'09*, 2009, pp. 21–28.

[4] G. Palermo, C. Silvano, and V. Zaccaria, "ReSPIR: a response surface-based pareto iterative refinement for application-specific design space exploration." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 12, pp. 1816–1829, 2009.

[5] A. Wieferink, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and A. Nohl, "A system-level processor/communication co-exploration methodology for multi-processor system-on-chip platforms," in *Proceedings of the conference on Design, Automation and Test in Europe DATE'04*, 2004, pp. 1256–1261.

[6] C. Erbas, A. D. Pimentel, M. Thompson, and S. Polstra, "A framework for system-level modeling and simulation of embedded systems architectures," *EURASIP Journal on Embedded Systems*, vol. 2007, no. 1, pp. 1–11.

[7] D. Knorreck, L. Apvrille, and R. Pacalet, "Fast simulation techniques for design space exploration," in *Proceedings of the International Conference on Objects, Models, Components and Patterns TOOLS'09*, 2009, pp. 308–327.

[8] M. Holzer, B. Knerr, and M. Rupp, "Design space exploration with evolutionary multi-objective optimisation," in *Proceedings of the Sympo-*

sium on Industrial Embedded Systems, July 2007, pp. 126–133.

[9] T. Blickle, J. Teich, and L. Thiele, "System-level synthesis using evolutionary algorithms," *Design Automation for Embedded Systems*, vol. 3, no. 1, pp. 23–58, 1998.

[10] M. Kaul and R. Vemuri, "Design-space exploration for block-processing based temporal partitioning of runtime reconfigurable systems," *Journal of Very Large Scale Integration (VLSI) Signal Processing Systems*, vol. 24, no. 2/3, pp. 181–209, 2000.

[11] F. Berthelot, F. Nouvel, and D. Houzet, "A flexible system-level design methodology targeting runtime reconfigurable fpgas," *EURASIP Journal on Embedded Systems*, vol. 2008, pp. 1–18, 2008.

[12] A. D. Pimentel, L. O. Hertzberger, P. Lieverse, P. van der Wolf, and E. F. Deprettere, "Exploring embedded-systems architectures with Artemis," *IEEE Computer*, vol. 34, no. 11, pp. 57–63, 2001.

[13] P. Lieverse, P. van der Wolf, K. Vissers, and E. de Prettere, "A methodology for architecture exploration of heterogeneous signal processing systems," *Journal of Very Large Scale Integration (VLSI) Signal Processing Systems*, vol. 29, no. 3, pp. 197–207, 2001.

[14] S. Mohanty, V. Prasanna, S. Neema, and J. Davis, "Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation," in *Proceedings of the joint conference on Languages, compilers and tools for embedded systems LCTES/SCOPES'02*, 2002, pp. 18–27.

[15] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," vol. 34, no. 2, pp. 171–210, June 2002.

[16] S. Vassiliadis and D. Soudris, *Fine- and Coarse-Grain Reconfigurable Computing*. Springer, 2007, vol. XVI.

[17] T. Todman, G. Constantinides, S. Wilton, O. Mencer, W. Luk, and P. Cheung, "Reconfigurable computing: Architectures and design methods," vol. 152, no. 2, pp. 193–207, June 2005.

[18] "www.xilinx.com."

[19] "www.altera.com."

[20] S. L. Coumeri and D. Thomas, "A simulation environment for hardware-software codesign," in *Proceedings of the International Conference on Computer Design ICCD'95*, 1995, pp. 58–63.

[21] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transaction on Computing*, vol. 55, no. 2, pp. 99–112, 2006.

[22] S. Vassiliadis, G. N. Gaydadjiev, K. Bertels, and E. M. Panainte, "The molen programming paradigm," in *Proceeding of the International workshop on Systems, Architectures, MOdeling and Simulation SAMOS'03*, July 2003, pp. 1–10.

[23] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The Molen polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, November 2004.

[24] P. Hsiung, C. Lin, and C. Liao, "Perfecto: A SystemC-based design-space exploration framework for dynamically reconfigurable architectures," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 1, no. 3, pp. 1–30, 2008.

[25] T. Rissa, M. Vasilko, and J. Niittylahti, "System-level modelling and implementation technique for runtime reconfigurable systems," in *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines FCCM'02*, 2002, pp. 295–296.

[26] C. Amicucci, F. Ferrandi, M. D. Santambrogio, and D. Sciuto, "SyCERS: a SystemC design exploration framework for soc reconfigurable architecture," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems & Algorithms ERSA'06*, 2006, pp. 63–69.

[27] "http://daedalus.liacs.nl."

[28] G. Kahn, "The semantics of a simple language for parallel programming," in *Proc. of the IFIP Congress 74*, 1974.

[29] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quanti-*

tative Approach, 2006.

[30] N. S. Voros and K. Masselos, *System-Level Design of Reconfigurable Systems-on-Chip*, 1st ed. Springer, November 2005.

[31] S. Hauck, "The roles of FPGAs in reprogrammable systems," *Proceedings of the IEEE*, vol. 86, no. 4, pp. 615–638, 1998.

[32] R. J. F. Stephen D. Brown and J. Rose, *Field-Programmable Gate Arrays*, ser. The Springer International Series in Engineering and Computer Science. Kluwer Academic Publlishers, 1992, vol. 180.

[33] Xilinx Corporation, "Coolrunner-ii cpld family overview," September 2008.

[34] R. W. Hartenstein and R. Kress, "A datapath synthesis system for the reconfigurable datapath architecture," in *Proceedings of the Asia and South Pacific Design Automation Conference ASP-DAC'95*, September 1995, pp. 479–484.

[35] R. W. Hartenstein, R. Kress, and H. Reinig, "A new FPGA architecture for Word-Oriented datapaths," in *Proceedings of the International Workshop on Field-Programmable Logic and Applications FPL'94*, 1994, pp. 144–155.

[36] Xilinx Corporation, "Virtex-4 family overview (V3.0)."

[37] "http://www.atmel.com/products/fpga/default.asp."

[38] "An iterative algorithm for hardware-software partitioning, hardware design space exploration and scheduling," *Design Automation for Embedded Systems*, vol. 5, no. 4, pp. 281–293, October 2000.

[39] K. B. Chehida and M. Auguin, "HW/SW partitioning approach for reconfigurable system design," in *Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems CASES'02*, 2002, pp. 247–251.

[40] Y.-C. Jiang and J.-F. Wang, "Temporal partitioning data flow graphs for dynamically reconfigurable computers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 12, pp. 1351–1361, 2007.

[41] K. M., G. Purna, and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," *IEEE Transactions on Computer Aided Design Integrated Circuits Systems*, vol. 48, no. 6, pp. 579–590, 1999.

[42] D. N. Rakhmatov and S. B. K. Vrudhula, "Hardware-software bipartitioning for dynamically reconfigurable systems," in *Proceedings of the International Symposium on Hardware/software codesign CODES'02*, 2002, pp. 145–150.

[43] Y.-C. Jiang and J.-F. Wang, "Temporal partitioning data flow graphs for dynamically reconfigurable computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 12, pp. 1351–1361, 2007.

[44] P. S. B. do Nascimento and M. E. de Lima, "Temporal partitioning for image processing based on time-space complexity in reconfigurable architectures," in *Proceedings of the conference on Design, Automation and Test in Europe DATE'06*, 2006, pp. 375–380.

[45] M. Koester, M. Porrmann, and H. Kalte, "Task placement for heterogeneous reconfigurable architectures," in *Proceedings of the International Conference on Field-Programmable Technology FPT'05*, 2005, pp. 43–50.

[46] C.-H. Lu, H.-W. Liao, and P.-A. Hsiung, "Multi-objective placement of reconfigurable hardware tasks in real-time system," *IEEE International Conference on Computational Science and Engineering*, vol. 2, pp. 921–925, 2009.

[47] F. Ghaffari, M. Auguin, M. Abid, and M. Ben-Jemaa, "Dynamic and online design space exploration for reconfigurable architectures," *Transactions on High-Performance Embedded Architectures and Compilers*, vol. 4050, pp. 179–193, 2007.

[48] Y. Lu, T. Marconi, G. Gaydadjiev, K. Bertels, and R. Meeuws, "A self-adaptive on-line task placement algorithm for partially reconfigurable systems," in *Proceedings of the International Parallel and Distributed Processing Symposium IPDPS'08*, 2008, pp. 1–8.

[49] A. Ahmadinia, C. Bobda, D. Koch, M. Majer, and J. Teich, "Task

scheduling for heterogeneous reconfigurable computers," in *Proceedings of the Symposium on Integrated Circuits and Systems Design SBCCI'04*, 2004, pp. 22–27.

[50] M. Holzer, B. Knerr, and M. Rupp, "Design space exploration for real-time reconfigurable computing," in *Proceedings of the Asilomar Conference on Signals, Systems, and Computers ACSSC'07*, November 2007, pp. 1981–1985.

[51] R. Guha, N. Bagherzadeh, and P. Chou, "Resource management and task partitioning and scheduling on a runtime reconfigurable embedded system," *Computers and Electrical Engineering*, vol. 35, no. 2, pp. 258–285, 2009.

[52] B. Miramond and J.-M. Delosme, "Design space exploration for dynamically reconfigurable architectures," in *Proceedings of the conference on Design, Automation and Test in Europe DATE'05*, 2005, pp. 366–371.

[53] J. Bhasker, *VHDL Primer*. Prentice Hall, 3rd edition, 1998.

[54] P. R. M. Donald E. Thomas, *The Verilog Hardware Description Language*. Springer, 5th edition, 2002.

[55] K. Keutzer, S. Malik, S. Member, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-vincentelli, "System-level design: Orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, pp. 1523–1543, 2000.

[56] B. Kienhuis, E. de Prettere, K. Vissers, and P. van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *Proceeding of the International Conference on Application-Specific Systems, Architectures and Processors ASAP'97*, 1997, pp. 338–349.

[57] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-software co-design of embedded systems: the POLIS approach*. Kluwer Academic Publishers, 1997.

[58] T. Grötker, G. Martin, and S. Swan, *System Design with SystemC*.

Kluwer Academic Publishers, 2002.

[59] T. Kogel, A. Wieferink, R. Leupers, G. Ascheid, and H. Meyr, "Virtual architecture mapping: A SystemC based methodology for architectural exploration of system-on-chip designs," in *Proceeding of the International workshop on Systems, Architectures, MOdeling and Simulation SAMOS'03*, 2003, pp. 138–148.

[60] T. Kogel, M. Doerper, A. Wieferink, R. Leupers, G. Ascheid, H. Meyr, and S. Goossens, "A modular simulation framework for architectural exploration of on-chip interconnection networks," in *Proceedings of the international conference on Hardware/software codesign and system synthesis CODES+ISSS'03*, 2003, pp. 7–12.

[61] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, and B. Vanthournout, "A modular simulation framework for spatial and temporal task mapping onto multi-processor SoC platforms," in *Proceedings of the conference on Design, Automation and Test in Europe DATE'05*, 2005, pp. 876–881.

[62] M. J. Rutten, J. T. van Eijndhoven, E. G. Jaspers, P. van der Wolf, E.-J. D. Pol, O. P. Gangwal, and A. Timmer, "A heterogeneous multiprocessor architecture for flexible media processing," *IEEE Design and Test of Computers*, vol. 19, pp. 39–50, 2002.

[63] P. Paulin, C. Pilkington, and E. Bensoudane, "StepNP: A system-level exploration platform for network processors," *IEEE Design and Test of Computers*, vol. 19, pp. 17–26, 2002.

[64] D. Quinn, B. Lavigueur, G. Bois, and M. Aboulhamid, "A system-level exploration platform and methodology for network applications based on configurable processors," in *Proceedings of the conference on Design, Automation and Test in Europe DATE'04*, 2004, pp. 1–6.

[65] "Design and simulation of heterogeneous systems using Ptolemy," in *Proceedings of ARPA RASSP Conference*, 1994, pp. 97–105.

[66] C. Hylands, E. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao, and H. Zheng, "Overview of the ptolemy project," Department of Electrical Engineering and Computer Science, University of California, Tech. Rep. UCB/ERL M03/25, July 2003.

[67] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *Computer*, vol. 36, pp. 45–52, 2003.

[68] G. Yang, X. Chen, F. Balarin, H. Hsieh, and A. Sangiovanni-Vincentelli, "Communication and co-simulation infrastructure for heterogeneous system integration," in *Proceedings of the conference on Design, Automation and Test in Europe DATE'06*, 2006, pp. 462–467.

[69] A. Sangiovanni-vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Design and Test of Computers*, vol. 18, pp. 23–33, 2001.

[70] L. Apvrille, W. Muhammad, R. Ameur-boulifa, S. Coudert, and R. Pacalet, "A UML-based environment for system design space exploration," in *Proceedings of 13th IEEE International Conference on Electronics, Circuits and Systems ICECS'06*, 2006.

[71] "Object Management Group. UML 2.0 Superstructure Specification, Geneva," 2003, www.omg.org/docs/ptc/03-08-02.pdf.

[72] C. Jaber, A. Kanstein, L. Apvrille, A. Baghdadi, P. L. Moenner, and R. Pacalet, "High-level system modeling for rapid HW/SW architecture exploration," in *Proceedings of the International Symposium on Rapid System Prototyping RSP'09*, 2009, pp. 88–94.

[73] R. Uhlig and T. N. Mudge, "Trace-driven memory simulation: A survey," *ACM Computing Survey*, vol. 29, no. 2, pp. 128–170, 1997.

[74] B. Ristau, T. Limberg, and G. Fettweis, "A mapping framework for guided design space exploration of heterogeneous mp-socs," in *Proceedings of the conference on Design, Automation and Test in Europe DATE'08*, 2008, pp. 780–783.

[75] M. Gries, "Methods for evaluating and covering the design space during early design development," *Integration Very Large Scale Integration (VLSI) Journal*, vol. 38, no. 2, pp. 131–183, December 2004.

[76] P. V. Knudsen and J. Madsen, "PACE: A dynamic programming algorithm for hardware/software partitioning," in *Proceedings of the International Workshop on Hardware/Software Co-Design*

Codes/CASHE'96, 1996, pp. 85–92.

[77] K. S. Chatha and R. Vemuri, "A tool for partitioning and pipelined scheduling of hardware-software systems," in *Proceedings of the International Symposium on System Synthesis ISSS'98*, 1998, pp. 145–151.

[78] B. Miramond and J.-M. Delosme, "Decision guide environment for design space exploration," in *Proceedings of the International Conference on Emerging Technologies and Factory Automation, ETFA'05*, 2005, pp. 888–896.

[79] L. Lanying and S. Min, "Software-hardware partitioning strategy using hybrid genetic and tabu search," in *Proceedings of the 2008 International Conference on Computer Science and Software Engineering CSSE'08*, 2008, pp. 83–86.

[80] B. Mei, P. Schaumont, and S. Vernalde, "A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems," in *Proceedings of the Annual Workshop on Circuits, Systems and Signal Processing ProRISC'00*, November 2000, pp. 1–8.

[81] G. Wang, W. Gong, and R. Kastner, "Application partitioning on programmable platforms using the ant colony optimization," *Embedded Computing*, vol. 2, no. 1, pp. 119–136, 2006.

[82] R. Maestre, F. Kurdahi, M. Fernandez, R. Hermida, N. Bagherzadeh, and H. Singh, "Kernel scheduling techniques for efficient solution space exploration in reconfigurable computing," *Systems Architectures*, vol. 47, pp. 277–292, 2001.

[83] K. S. Chatha and R. Vemuri, "An iterative algorithm for partitioning and scheduling of area constrained HW-SW systems," in *Proceedings of the International Workshop on Rapid System Prototyping RSP'99*, July 1999, pp. 134–139.

[84] P. Y. C. Theerayod Wiangtong and W. Luk, "Hardware/software codesign: a systematic approach targeting data-intensive applications," *IEEE Signal Processing*, vol. 22, no. 3, pp. 14–22, May 2005.

[85] "System-level performance evaluation of reconfigurable processors," *Microprocessors and Microsystems, Special Issue on FPGA Tools and*

Techniques, vol. 29, no. 2-3, pp. 63–73, 2005.

[86] R. Enzler, C. Plessl, and M. Platzner, "Co-simulation of a hybrid multi-context architecture," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms ERSA'03*, 2003, pp. 174–180.

[87] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, pp. 59–67, 2002.

[88] J. Noguera and R. M. Badia, "System-level power-performance trade-offs in task scheduling for dynamically reconfigurable architectures," in *Proceedings of the international conference on Compilers, Architecture and Synthesis for Embedded Systems CASES'03*, 2003, pp. 73–83.

[89] C. Haubelt, S. Otto, C. Grabbe, and J. Teich, "A system-level approach to hardware reconfigurable systems," in *Proceedings of the Asia and South Pacific Design Automation Conference ASP-DAC'05*, 2005, pp. 298–301.

[90] A. Pelkonen, K. Masselos, and M. Cupák, "System-level modeling of dynamically reconfigurable hardware with SystemC," in *Proceedings of the International Parallel and Distributed Processing Symposium IPDPS'03*, 2003, pp. 1–8.

[91] Y. Qu and J.-P. Soininen, "SystemC-based design methodology for reconfigurable system-on-chip," in *Proceedings of the Euromicro Conference on Digital System Design DSD'05*, 2005, pp. 364–371.

[92] Y. Qu, K. Tiensyrjä, and K. Masselos, "System-level modeling of dynamically reconfigurable co-processors," in *Proceedings of the International Conference on Field Programmable Logic and Applications FPL'04*, 2004, pp. 881–885.

[93] T. Rissa, A. Donlin, and W. Luk, "Evaluation of systemC modelling of reconfigurable embedded systems," in *Proceedings of the conference on Design, Automation and Test in Europe DATE'05*, 2005, pp. 253–258.

[94] K. Tiensyrjä, M. Cupák, K. Masselos, and M. Pettissalo, "SystemC and ocapi-xl based system-level design for reconfigurable systems-on-chip," in *Forum on specification and Design Languages FDL'04*, 2004, pp.

428–440.

[95] A. Ahmadinia, B. Ahmad, A. T. Erdogan, and T. Arslan, "System-level modelling and analysis of embedded reconfigurable cores for wireless systems," in *Proceedings of International Conference on Field Programmable Logic and Applications FPL'07*, 2007, pp. 757–760.

[96] P. Hsiung, S. Lin, Y. Chen, and C. Huang, "Perfecto: A SystemC-based performance evaluation framework for dynamically partially reconfigurable systems," in *Proceedings of the International Conference on Field Programmable Logic and Applications FPL'06*, August 2006, pp. 1–6.

[97] B. Miramond, E. Huck, F. Verdier, M. E. A. Benkhelifa, B. Granado, M. Aichouch, J.-C. Prvotet, D. Chillet, S. Pillement, T. Lefebvre, and Y. Oliva, "OveRSoC : A framework for the exploration of RTOS for RSoC platforms," *International Journal on Reconfigurable Computing*, vol. 2009, no. 450607, pp. 1–18, December 2009.

[98] C.-H. Tseng and P.-A. Hsiung, "UML-based design flow and partitioning methodology for dynamically reconfigurable computing systems," in *International Conference on Embedded and Ubiquitous Computing EUC'05*, 2005, pp. 479–488.

[99] C.-H. Huang, P.-A. Hsiung, and J.-S. Shen, "UML-based hardware/software co-design platform for dynamically partially reconfigurable network security systems," *Journal of Systems Architecture*, vol. 56, no. 2-3, pp. 88–102, 2010.

[100] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, J. Riihimäki, and K. Kuusilinna, "UML-based multiprocessor SoC design framework," *Transactions on Embedded Computing Systems*, vol. 5, no. 2, pp. 281–320, 2006.

[101] G. Still, R. Lysecky, and F. Vahid, "Dynamic hardware/software partitioning: A first approach," in *Proceedings of the Design Automation Conference DAC'03*, 2003.

[102] V. Nollet, P. Avasare, H. Eeckhaut, D. Verkest, and H. Corporaal, "Runtime management of a MPSoC containing fpga fabric tiles," *IEEE Transaction on Very Large Scale Integration (VLSI) System*, vol. 16,

no. 1, pp. 24–33, 2008.

[103] C. Huang and F. Vahid, "Dynamic coprocessor management for FPGA-enhanced compute platforms," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems CASES'08*, 2008, pp. 71–78.

[104] W. Fu and K. Compton, "An execution environment for reconfigurable computing," in *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines FCCM'05*, 2005, pp. 149–158.

[105] V. M. Sima and K. Bertels, "Runtime decision of hardware or software execution on a heterogeneous reconfigurable platform," in *Proceedings of the International Parallel and Distributed Processing Symposium IPDPS'09*, 2009, pp. 1–6.

[106] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and H. Yajun, "Resource manager for non-preemptive heterogeneous multiprocessor system-on-chip," in *Proceedings of the International Workshop on Embedded Systems for Real Time Multimedia ESTIMedia'06*, 2006, pp. 33–38.

[107] O. Moreira, J. J. Mol, and M. Bekooij, "Online resource management in a multiprocessor with a network-on-chip," in *Proceedings of the International Symposium on Applied computing SAC'07*, 2007, pp. 1557–1564.

[108] L. T. Smit, J. L. Hurink, and G. J. M. Smit, "Runtime mapping of applications to a heterogeneous SoC," in *International Symposium on System-on-Chip SoC'05*, 2005, pp. 78–81.

[109] P. K. F. Hölzenspies, G. J. M. Smit, and J. Kuper, "Mapping streaming applications on a reconfigurable MPSoC platform at runtime," in *Proceedings of the International Symposium on System-on-Chip SoC'07*, 2007, pp. 74–77.

[110] M. A. A. Faruque, R. Krist, and J. Henkel, "ADAM: runtime agent-based distributed application mapping for on-chip communication," in *Proceedings of the Design Automation Conference DAC'08*, 2008, pp. 760–765.

[111] L. Bossuet, G. Gogniat, and J.-L. Philippe, "Communication-oriented design space exploration for reconfigurable architectures," *EURASIP Journal on Embedded Systems*, vol. 2007, no. 1, 2007.

[112] ——, "Fast design space exploration method for reconfigurable architectures," in *Proceeding of International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2003, pp. 23–26.

[113] C. Ykman-Couvreur, E. Brockmeyer, V. Nollet, T. Marescaux, F. Catthoor, and H. Corporaal, "Design-time application exploration for MPSoC customized runtime management," in *Proceesings of International Symposium on System-on-Chip SOC'05*, 2005, pp. 66–69.

[114] C. Ykman-Couvreur, V. Nollet, T. Marescaux, E. Brockmeyer, F. Catthoor, and H. Corporaal, "Pareto-based application specification for MPSoC customized runtime management," in *Proceedings of the International Conference on Systems, Architectures, MOdeling and Simulation IC-SAMOS'06*, 2006, pp. 78–84.

[115] G. Mariani, P. Avasare, G. Vanmeerbeeck, C. Ykman-Couvreur, G. Palermo, C. Silvano, and V. Zaccaria, "An industrial design space exploration framework for supporting runtime resource management on multi-core systems," in *Proceedings of the conference on Design, Automation and Test in Europe DATE'10*, 2010, pp. 196–201.

[116] M. Gries, "Algorithm-architecture tradeoffs in network processor design," Ph.D. dissertation, Swiss Federal Institute of Technology Zurich (ETH), Zurich, Switzerland, July 2001.

[117] S. Verdoolaege, H. Nikolov, and T. Stefanov, "PN: a tool for improved derivation of process networks," *EURASIP Journal on Embedded Systems*, vol. 2007, no. 1, p. 13, 2007.

[118] B. K. Dwivedi, H. Dh, M. Balakrishnan, and A. Kumar, "RPNG: a tool for random process network generation," in *Proceedings of Asia and South Pacific International Conference in Embedded SoCs ASPICES'05*, 2004.

[119] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *ACM SIGPLAN Notices*, vol. 17, no. 6, 1982.

[120] R. J. Meeuws, Y. D. Yankova, K. Bertels, G. N. Gaydadjiev, and S. Vassiliadis, "A quantitative prediction model for hardware/software partitioning," in *Proceedings of International Conference on Field Programmable Logic and Applications FPL'07*, August 2007, pp. 735–739.

[121] V. Nollet, P. Avasare, J.-Y. Mignolet, and D. Verkest, "Low cost task migration initiation in a heterogeneous MPSoC," in *Proceedings of the conference on Design, Automation and Test in Europe DATE'05*, 2005, pp. 252–253.

[122] X. Zhou, L. Liang, Y. Wang, and C. Peng, "Online task scheduling for heterogeneous reconfigurable systems," *Computer Supported Cooperative Work in Design*, pp. 596–607, 2008.

[123] L. Gantel, S. Layouni, M. E. A. Benkhelifa, F. Verdier, and S. Chauvet, "Multiprocessor task migration implementation in a reconfigurable platform," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs ReConFig'09*, 2009, pp. 362–367.

[124] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting task migration in multi-processor systems-on-chip: a feasibility study," in *Proceedings of the conference on Design, Automation and Test in Europe DATE'06*, 2006, pp. 15–20.

[125] M. Thompson, H. Nikolov, T. Stefanov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere, "A framework for rapid system-level exploration, synthesis, and programming of multimedia MPSoCs," in *Proceedings of the 5th international conference on Hardware/software codesign and system synthesis CODES+ISSS'07*, 2007, pp. 9–14.

[126] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere, "Daedalus: Toward composable multimedia MPSoC design," in *Proceedings of the 45th annual Design Automation Conference DAC'08*, 2008, pp. 574–579.

[127] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast template placement for reconfigurable computing systems," *Design Test of Computers*, vol. 17, no. 1, pp. 68–83, 2000.

[128] Xilinx Corporation, "LogiCORE IP XPS HWICAP (v5.00a)," July 2010.

List of Publications

International Conferences

1. **K. Sigdel**, M. Thompson, C. Galuzzi, A.D. Pimentel, K.L.M. Bertels, **Runtime Task Mapping Based on Hardware Configuration Reuse**, *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig'10)*, Cancun, Mexico, December 2010, pp. 1-6.

2. **K. Sigdel**, M. Thompson, C. Galuzzi, A.D. Pimentel, K.L.M. Bertels, **Evaluation of Runtime Task Mapping Heuristics with *rSesame* - A Case Study**, *Proceedings of the Design, Automation and Test in Europe (DATE'10)*, Dresden, Germany, March 2010, pp. 831-836.

3. **K. Sigdel**, M. Thompson, C. Galuzzi, A.D. Pimentel, K.L.M. Bertels, **A Generic System-Level Runtime Simulation Framework for Reconfigurable Architectures**, *Proceedings of the International Conference on Field-Programmable Technology (FPT'09)*, Sydney, Australia, December 2009, pp. 460-464.

4. **K. Sigdel**, M. Thompson, A.D. Pimentel, C. Galuzzi, K.L.M. Bertels, **System-Level Runtime Mapping Exploration of Reconfigurable Architectures**, *Proceedings of the Reconfigurable Architectures Workshop (RAW'09)*, Rome, Italy, May 2009, pp. 1-8.

5. S.A. Ostadzadeh, R.J. Meeuws, **K. Sigdel**, K.L.M. Bertels, **A Multipurpose Clustering Algorithm for Task Partitioning in Multicore Reconfigurable Systems**, *Proceedings of the International Workshop on Multi-Core Computing Systems (MuCoCoS'09)*, Fukuoka, Japan, March 2009, pp. 663-668.

6. S.A. Ostadzadeh, R.J. Meeuws, **K. Sigdel**, K.L.M. Bertels, **A Clustering Framework for Task Partitioning Based on Function-level Data Usage Analysis**, *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA'09)*, Monterey, California, USA, February 2009, pp. 279-279.

7. R.J. Meeuws, **K. Sigdel**, Y.D. Yankova, K.L.M. Bertels, **High Level**

Quantitative Interconnect Estimation for Early Design Space Exploration, *Proceedings of the International Conference on Field-Programmable Technology (FPT'08)*, Taiwan, December 2008, pp. 317-320.

8. **K. Sigdel**, M. Thompson, A.D. Pimentel, T. P. Stefanov, K.L.M. Bertels, **System-Level Design Space Exploration of Dynamic Reconfigurable Architectures**, *Proceedings of the International Symposium on Systems, Architectures, MOdeling and Simulation (SAMOS'08)*, Samos, Greece, July 2008, pp. 279–288.

9. K.L.M. Bertels, G.K. Kuzmanov, E. Moscu Panainte, G.N. Gaydadjiev, Y. D. Yankova, V.M. Sima, **K. Sigdel**, R.J. Meeuws, S. Vassiliadis, **Hartes Toolchain Early Evaluation: Profiling, Compilation and HDL Generation**, *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'07)*, Amsterdam, The Netherlands, August 2007, pp. 402-408.

10. K.L.M. Bertels, G.K. Kuzmanov, E. Moscu Panainte, G.N. Gaydadjiev, Y.D. Yankova, V.M. Sima, **K. Sigdel**, R.J. Meeuws, S. Vassiliadis, **Profiling, Compilation, and HDL Generation within the hArtes Project**, *Proceedings of the Design, Automation and Test in Europe (DATE'07)*, Nice, France, April 2007.

International Workshops

1. **K. Sigdel**, M. Thompson, A.D. Pimentel, C. Galuzzi, K.L.M. Bertels, **rSesame: System-Level Design Space Exploration Framework for Reconfigurable Architectures**, *Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*, Terrassa, Spain, July 2010.

2. **K. Sigdel**, M. Thompson, A.D. Pimentel, K.L.M. Bertels, **System-Level Design Space Exploration and Runtime Mapping of Reconfigurable architecture**, *Architectures and Compilers for Embedded Systems (ACES)*, Edegem, Belgium, September 2008.

3. **K. Sigdel**, M. Thompson, A.D. Pimentel, K.L.M. Bertels, **Towards System Level Runtime Design Space Exploration of Reconfigurable Ar-**

chitecture, *Annual Workshop on Circuits, Systems and Signal Processing (ProRISC)*, Veldhoven, The Netherlands, November 2008.

4. **K. Sigdel**, R.J. Meeuws, K.L.M. Bertels, **A Profiling Framework for Design Space Exploration in Heterogeneous System Context**, *Annual Workshop on Circuits, Systems and Signal Processing (ProRISC)*, Veldhoven, The Netherlands, November 2007.

5. R.J. Meeuws, **K. Sigdel**, Y.D. Yankova, K.L.M. Bertels, **Quantitative Prediction for Early Design Space Exploration in Delft Workbench: An Outlook**, *Annual Workshop on Circuits, Systems and Signal Processing (ProRISC)*, Veldhoven, The Netherlands, November 2007.

6. **K. Sigdel**, L. Shuai, B. Pourebrahimi, K.L.M. Bertels, S. Vassiliadis, **Centralized Matchmaking - An Empirical Study**, *Annual Workshop on Circuits, Systems and Signal Processing (ProRISC)*, Veldhoven, The Netherlands, November 2006.

7. **K. Sigdel**, K.L.M. Bertels, B. Pourebrahimi, S. Vassiliadis, L.S Shuai, **A framework for Adaptive Matchmaking in Distributed Computing**, *Crakow GRID Workshop (CWG)*, Krakow, Poland, December 2004.

8. B. Pourebrahimi, K.L.M. Bertels, S. Vassiliadis, **K. Sigdel**, **Matchmaking within Multi-Agent Systems**, *Annual Workshop on Circuits, Systems and Signal Processing (ProRISC)*, Veldhoven, The Netherlands, November 2004.

Reports

1. **K. Sigdel**, M. Thompson, A.D. Pimentel, T. P. Stefanov, K.L.M. Bertels, **System-Level Dynamic Space Exploration for Dynamic Reconfigurable System**, *Technical Report*, Computer Engineering, TUDelft, December 2008.

2. **K. Sigdel**, Y.D. Yankova, K.L.M. Bertels, **Survey on Application Profiling and Dynamic Trace Analysis**, *Technical Report*, Computer Engineering, TUDelft, January 2007.

About the Author

Kamana Sigdel received her bachelor of engineering in Computer Engineering (CE) from Kathmandu University, Nepal. After completing her bachelor, she started her Msc study in the CE-lab at Delft University of Technology (TUDelft), where she successfully completed her master thesis under the supervision of Dr. Koen Bertels and Prof. dr. Stamatis Vassiliadis. In September 2006, she started her PhD in the same group under the guidance of Dr. Koen Bertels (TUDelft) and Dr. Andy D. Pimentel (University of Amsterdam). She has assisted as a course instructor in various courses, such as system programming in C and computer architecture. She has also worked as a part time document classifier in the European Patent Office. Besides her research, she serves as a board member for the international council board at EEMCS faculty, TUDelft.

Her research interests include, embedded systems, multi-core platforms, heterogeneous architecture, reconfigurable architectures, modeling and simulation, system-level design, real time embedded system, grid computing, agent based software engineering, multi-agent systems.