Scenario-based Run-time Adaptive Multi-Processor System-on-Chip

Wei Quan

University of Amsterdam

Scenario-based Run-time Adaptive Multi-Processor System-on-Chip Wei Quan

Scenario-based Run-time Adaptive Multi-Processor System-on-Chip

Scenario-based Run-time Adaptive Multi-Processor System-on-Chip

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor promoties ingestelde
commissie, in het openbaar te verdedigen in de Agnietenkapel
op donderdag 17 september 2015, te 14:00 uur

door

Wei Quan

geboren te Hunan, China

Promotiecommissie:

Promotor:	Prof. dr. ir.	C.T.A.M. de Laat	UvA
Co-promotor:	Dr.	A.D. Pimentel	UvA
Overige leden:	Prof. dr.	P.W. Adriaans	UvA
	Prof. dr. ir.	H. Corporaal	TU/e
	Prof. dr.	C. Zhang	NUDT
	Dr.	C.U. Grelck	UvA
	Dr.	A.L. Varbanescu	UvA

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

This research received full funding from China Scholarship Council.

This work was carried out in the ASCI graduate school. ASCI dissertation series number 334.

Cover design by Lingxue Cao
Typeset by LaTeX
Printed and bound by Off Page Amsterdam
ISBN: 978-94-6182-592-6

Contents

CHAPTER 1

Introduction

Embedded systems, a kind of computing systems designed for special purposes, greatly improve the quality of our lives. They span all aspects of modern life and can be found nearly everywhere like consumer electronics, common household devices, medical equipments, vehicles and so on. Different with traditional embedded systems that can perform only limited functions, modern embedded systems are increasingly powerful and versatile especially in the products of consumer electronics. However, with more and more features integrated in these systems, the constraints concerning size, performance, power consumption and so on are also increasing. Also the run-time behaviour of the embedded systems are becoming more and more complex and dynamic. These system constraints and the complex run-time behaviour make the design of such embedded systems a challenging effort. The designers not only need to carefully trade off various design objectives by means of software and hardware co-design approaches at the early stage of system design, but also should provide a proper adaptivity support on the target system to handle the dynamic run-time behaviour. The design of the latter part is a totally new topic in the domain of embedded systems which has drawn quite a lot of attention from researchers in recent years.

1.1 Motivation

In the embedded computer system domain, two trends are clearly visible. First, for the implementation of modern embedded system, such as those for consumer electronics like smart-phones, digital televisions, set-top boxes and so on, there has been an important move towards Multi-Processor System-on-Chip (MPSoC) architectures [137] to satisfy the non-functional requirements of embedded applications. These MPSoCs are often heterogeneous systems, containing programmable processor cores for flexible application support as well as dedicated processing elements for achieving power and performance goals. Taking the mobile SoC products such

as Snapdragon from Qualcomm [1], Exynos from Samsung [2], Tegra from Nvidia [3] and so on as an example, these processors are often integrated with multiple CPU (Central Processing Unit) cores, a number of application-specific accelerators targeted at the mobile market such as a Graphics Processing Unit (GPU), Digital Signal Processor (DSP), video and audio encoder/decoder, image signal processor, etc.. The number of processing elements in these MPSoCs also steadily increases. Whereas current MPSoCs still contain a limited number of processing elements, future MPSoCs will feature tens up to hundreds of (heterogeneous) processing elements that are all integrated on a single chip to handle the next generation of embedded applications like real-time physics, artificial intelligence, 3D rendering effects and so on [68].

A second trend is the increasing need for system adaptivity [70]. Future embedded systems will need to continuously customize their underlying system at run time according to the application workload at hand and the state of the system itself. There are multiple driving forces for this technology shift towards adaptive systems. First, today's MPSoC systems often require supporting a growing number of applications and standards, where multiple applications can run simultaneously. For each single application, there may also be different execution modes (or program phases) with different computational and communication requirements. For example, in Software Defined Radio appliances a radio may change its behaviour according to resource availability, such as the Long Term Evolution (LTE) standard which uses adaptive modulation and coding to dynamically adjust modulation schemes and transport block sizes based on channel conditions. As a consequence, the behaviour of application workloads executing on embedded systems can change dramatically over time. Second, for a large variety of embedded systems, dynamic Quality of Service management, in which different system qualities like performance, precision and power consumption can be dynamically traded off, becomes more and more important. Take for example a video decoder that dynamically lowers its bit rate (and thus image quality) to reduce its computational demands to save battery power. Third, reliability also becomes an important driver for adaptive systems, as the advances in chip technology have reached a level where our circuits are no longer fully reliable, increasing the chances of transient and permanent faults [109].

The combination of the above trends in embedded systems motivates the research of this thesis where the main goal is to ***increase the adaptivity of MPSoC systems to efficiently deal with the complex and dynamic behaviour of embedded applications***.

1.2 Current Techniques for Adaptive Systems

In recent years, a number of techniques have emerged to increase the adaptivity of MPSoC systems. These techniques are mainly based on run-time system reconfiguration which can be simply divided into two categories: software reconfiguration and hardware reconfiguration.

[1] https://www.qualcomm.com/products/snapdragon
[2] https://www.samsung.com/exynos
[3] http://www.nvidia.com/object/tegra

For adaptive MPSoC systems with software reconfiguration, a commonly used technique is the dynamic remapping of application tasks (both computing and communicating tasks) onto the underlying hardware resources at run time [21, 14, 70]. In this kind of solutions, a middleware support layer like a run-time system manager (or multiple system managers in case of distributed resource management) which could be either integrated in the Operating System (OS) or implemented on top of the OS should be provided on the target system to achieve dynamic software reconfiguration. And the problem of how the application tasks should be remapped to improve the system performance and/or energy consumption is solved by a task mapping mechanism implemented in the above-mentioned middleware layer which is also the main research issue in these solutions. Besides the task mapping based approaches, another research direction for improving system adaptivity by software reconfiguration focuses on the programming model of MPSoCs. [118] gives a typical example for this kind of solutions where the idea of resource-aware programming is introduced into the programming model. Under such a novel programming model, a program gets the ability to explore and dynamically spread its computations to processors.

With regard to MPSoC systems where the adaptivity is achieved by hardware reconfiguration, the hardware components in these systems can be dynamically reconfigured according to the workload and the execution environment/state. These solutions are totally different from the solutions using software reconfiguration. For the purpose of dynamic hardware reconfiguration, at least two techniques can be found. One of them is the technique of dynamically changing system parameters such as done in dynamic frequency and voltage scaling. And another technique is dynamically reconfiguring processing components for accelerating application tasks (e.g., [32, 131, 3, 13]) or network components to customize the network to a specific application workload (e.g. [130, 114]).

Among the currently emerged techniques in adaptive MPSoC systems, the solutions using hardware reconfiguration often have a better performance compared with the ones using software reconfiguration. However, this benefit comes at the cost of more hardware resources. Considering the two kinds of solutions that use software reconfiguration, the ones that focus on the programming model involve higher design complexity not only for application designers but also for system designers to provide the support for a new programming model. However, the solutions using application task remapping provide a good trade off between the hardware overhead and design complexity. Consequently, in the research of this thesis, we focus on the techniques for ***dynamic application task remapping*** to achieve our research goal.

1.3 Task Mapping Methodologies for MPSoCs

1.3.1 Traditional Solutions

The problem of optimally mapping application tasks onto a given set of heterogeneous processors with the aim of optimising goal(s) like maximal throughput (performance) and/or minimal overall energy consumption has been known, in general, to be NP-complete. This problem is exacerbated when mapping

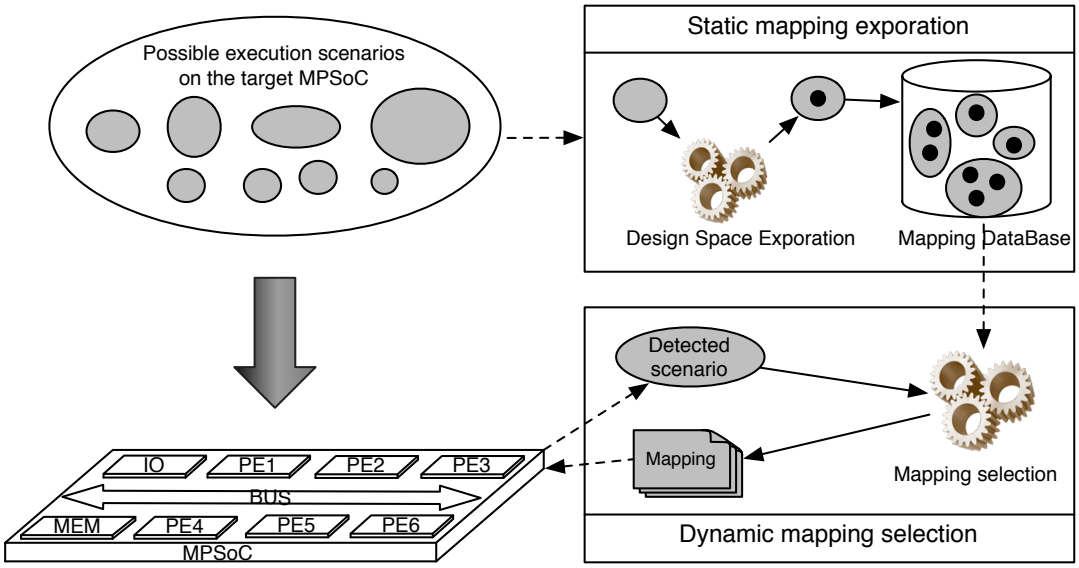


Figure 1.1: State-of-the-art task mapping methodologies.

multiple applications (i.e., bigger task sets) onto the target platform. Traditionally, the mapping of applications onto the underlying architectural components of MPSoC systems has always been done in a static fashion at design time [88, 63, 26, 69, 53, 102, 104, 15, 120, 25, 23]. These methods typically use computationally intensive search methods to find the optimal mapping or near optimal mapping for all applications that may run on the system. Evidently, the drawback of such static mapping techniques is that they cannot cope with dynamic application behaviour in which different combinations of applications are concurrently executing over time and contending for system resources. To overcome this drawback of static mapping techniques, dynamic (on-the-fly) task mapping techniques [46, 110, 79, 119, 132, 83, 34] for MPSoC systems have been widely studied by researchers. Different to the static task mapping techniques, the dynamic opponents cannot be computationally intensive as they have to efficiently make task mapping decisions at run time. Therefore, these techniques typically use heuristics to find good task mappings. However, the static task mapping techniques usually obtain mappings of higher quality compared to those derived from dynamic algorithms as the former allow for exploring a larger design space for the underlying architecture. This, of course, at the cost of consuming more time.

1.3.2 State-of-the-art Solutions

In the above mentioned two mapping methodologies, each has its own advantages but also drawbacks. To address the issues and incorporate the advantages of pure static and dynamic mapping strategies, hybrid (semi-static) mapping approaches [80, 4, 57, 64, 140, 7, 136, 91, 128, 60, 82, 61, 106] have been proposed recently.

These hybrid strategies combine the design-time static task mapping with the run-time management in order to select mapping configurations that are best suited to the current workload execution scenario on the target system [115]. A

simple workflow of hybrid task mapping techniques is illustrated in Figure 1.1. As can be seen from this figure, this type of methods can be divided into two stages. The first stage is the design-time preparation which determines one or multiple system configurations for each possible workload scenario that may appear on the target system. For example, these configurations could be different task-to-resource mappings (derived by static mapping techniques) optimizing the system for e.g. performance and/or energy consumption. The second stage is the run-time stage in which a light-weight run-time resource manager chooses the appropriate system configuration from the pre-optimized configurations based on the current system execution scenario and system status.

1.3.3 Advantages of State-of-the-art Solutions

In hybrid task mapping methodologies, the compute intensive analysis is performed at design-time which greatly reduces the computational overhead of run-time mapping optimisation and consequently facilitates efficient task remapping. They take advantage from both design-time static task mapping techniques (high mapping quality) and run-time dynamic task mapping heuristics (low computation overhead and support for application dynamism). However, there are some drawbacks for this kind of solutions as well, which will be explained in detail as our research issues in the following section.

1.4 Problem Statement

In our research, the overall research problem is *how to improve the adaptivity of MPSoC systems with dynamic and complex application behaviour using dynamic application task remapping*. As mentioned in Section 1.1, today's MPSoC systems often require to supporting an increasing number of applications where each application may also have different execution modes (or program phases) with different requirements. As a consequence, the behaviour of application workloads executing on MPSoCs can change dramatically over time. Here, one can distinguish two forms of dynamic application behaviour: inter-application dynamism and intra-application dynamism. These forms of dynamism are often captured using *scenarios* [93, 42, 43, 127]. This means that there are two different kinds of scenarios: inter-application scenarios to describe the simultaneously running applications in the system, and intra-application scenarios that define the different execution modes for each application. The combination of these inter- and intra-application scenarios are called ***workload scenarios*** [128, 125], and specify the application workload in terms of the different applications that are concurrently executing and the mode of each application as shown in Figure 1.2. For a target MPSoC system for which n target applications need to be supported where each application has m execution modes, the total number of possible workload scenarios on the system is $(m+1)^n - 1$. Evidently, when the parameters n and m are relatively small, the previously discussed hybrid task mapping techniques could perfectly improve the efficiency of the target MPSoC system. On the contrary, with the scaling of these two parameters which might happen in the near future of MPSoCs, several problems will occur in these state-of-the-art techniques, as will be elaborated on below.

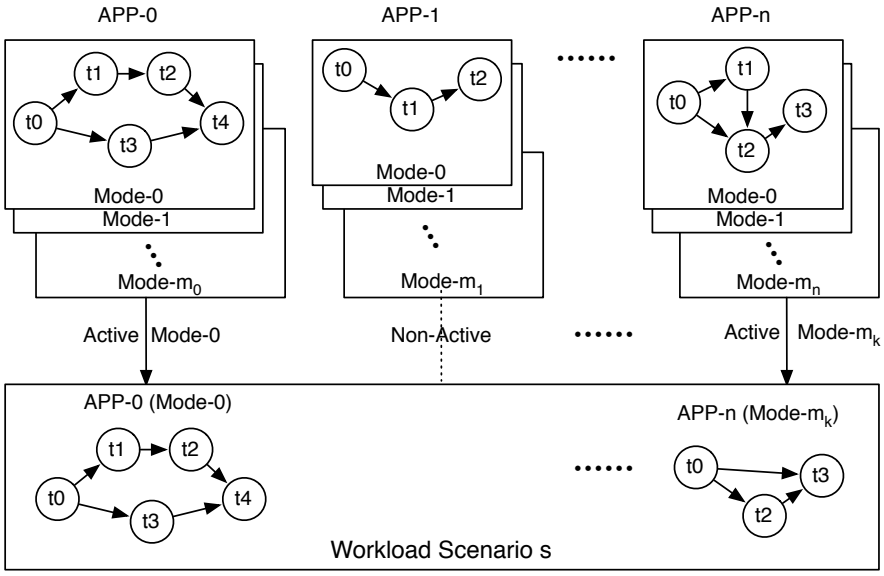


Figure 1.2: Definition of a workload scenario.

Following the process of hybrid task mapping methodologies shown in Figure 1.1, the first step is to explore one or multiple task mappings for each workload scenario. In this process, the static task mapping problem is usually solved by computational intensive algorithms like Simulated Annealing [87] or Genetic Algorithms [6] to generate optimal or near optimal mapping solutions under the target optimisation goal(s). For each single task mapping problem, its complexity[4] depends on the number of target application tasks and the number of heterogeneous processing elements in the target MPSoC system. With the number of the target application tasks increasing, the mapping solution space will explode exponentially. Consequently, in our research where the number of target application tasks for each separate design-time task mapping problem is usually very large, these computational intensive algorithms will suffer from the problem of unacceptable Design Space Exploration (DSE) time. This problem leads to the first concrete research question of this thesis:

- *How to improve the efficiency of static task mapping exploration?*

Even though the exploration time for each single task mapping problem can be reduced, the total time of mapping exploration for a large number of possible workload scenarios as mentioned in the first paragraph of this section is also a serious problem which will greatly influence the success of a modern embedded system with a stringent time-to-market requirement such as mobile phones. Considering, e.g., 10 applications with 5 execution modes for each application, there will be 60 million different workload scenarios. If each scenario takes only one second for task mapping exploration at design time, then one would need nearly two years to obtain all the mappings. Moreover, storing all these optimised mappings such that they can be used at run time by the system manager would also

[4]The number of possible mappings of a mapping problem where i application tasks need to be mapped onto j heterogeneous processing elements is j^i.

be unrealistic as this would take up too much memory storage. Therefore, the scalability of hybrid task mapping methodologies with regard to the number of workload scenarios is also a serious issue. Besides that, the flexibility is another issue for this type of methods at design time. This because all potential applications on the target platforms must be known at design time. Therefore, when the application set changes, design-time analysis needs to be redone entirely. These two issues of the above discussed hybrid task mapping methodologies bring out our second research question:

- *How to achieve scalability with regard to the number of workload scenarios as well as flexibility in hybrid task mapping techniques?*

Focusing on the run-time management of these hybrid task mapping techniques, the system manager typically always tries to reconfigure the system resources when a new workload scenario has been detected and, doing so, the reconfiguration costs are not explicitly taken into account. These reconfiguration costs may be substantial as they include the overhead of application tasks that may need to be migrated between different processors in the MPSoC. Especially in the case of fine-grained workload scenarios – which are workload scenarios that are only active for a short duration [5] – such overheads may easily eliminate the benefits of reconfiguring the system: the reconfiguration itself may take longer than the performance gain that is obtained after reconfiguration. Consequently, in the case of fine grained workload scenarios on the target MPSoC system, these hybrid task mapping solutions may actually degrade the system performance, especially on heterogeneous MPSoC systems. In our research, we refer to this problem as *blind adaptivity*. With regard to this problem, our third research question is:

- *How to deal with blind adaptivity at run time for an adaptive MPSoC system?*

In the second problem statement, we considered the mapping scalability problem related to the number of target application tasks. However, as mentioned in the second problem statement, it is also related to the target hardware platform (the number of heterogeneous processing elements in the system). With the technological advancement, the number of processing elements in a MPSoC system will increase to hundreds or even more. The future large-scale MPSoC systems impose a big challenge to manage their resources at run-time in a scalable manner. Therefore, we define a fourth research question:

- *Are hybrid task mapping techniques still applicable on future large-scale MPSoCs?*

1.5 Contribution and Thesis Overview

1.5.1 Main Contributions

The work presented in this thesis has been performed in the context of the Sesame system-level simulation framework [95, 124] which provides the ability for efficiently evaluating non-functional behaviour like performance and energy consumption of an embedded system at a high level of abstraction. Based on this simulation

[5] Different workload scenarios rapidly succeed one another at run time.

framework, we strive to solve the research questions discussed in Section 1.4 for our scenario-based adaptive MPSoC systems. The main contributions of this thesis are:

- A novel bias-elitist genetic algorithm (GA) that is guided by domain-specific heuristics to improve the efficiency of static task mapping exploration. For our task mapping problem, when the target mapping solution space is very large, it is impossible to explore each solution at design time. Consequently, in order to reduce the search time, a new GA-based mapping DSE algorithm is proposed to allow for effectively pruning the search space.
- An extension of the Sesame simulator with a run-time resource scheduling framework to support dynamic scenario execution and task remapping. As the original Sesame framework only supports the simulation of target applications and architectures under a static mapping, a run-time system management framework is provided. In order to be able to simulate the dynamic run-time application behaviour in Sesame.
- A scenario clustering based task mapping approach to solve the scalability problem of hybrid task mapping techniques with regard to the number of workload scenarios. In this proposed approach, by using the scenario clustering method, the number of task mappings and consequently the time cost for mapping exploration at design time and the memory storage for storing the pre-optimised mappings at run time can be greatly reduced. In addition, a run-time on-the-fly heuristic is presented for dynamic Quality-of-Service (QoS) management with regard to application performance requirements.
- A novel hybrid task mapping approach to solve both the scalability and flexibility problem of general hybrid task mapping techniques as mentioned in the second research question. In normal hybrid task mapping techniques as discussed in Section 1.3.2, the design time mapping exploration is done at workload scenario level, which is the reason why these two problems actually occur. To solve these problems, we use a divide-and-conquer method where the scenario-level task mapping problem is broken down into application-level task mapping problems at design time, and the application-level mapping solutions are then dynamically combined and further optimised to give a complete solution for a workload scenario at run time.
- A further extension of the Sesame simulator with run-time system reconfiguration cost evaluation. When considering the blind adaptivity problem discussed in the third problem statement, the system reconfiguration cost should be carefully considered for dynamic task remapping at run time. For this purpose, we have extended the system-level simulation framework based on the Sesame simulator with dynamic run-time resource management support as mentioned above. It has been extended to also support a flexible and efficient modeling, simulation and exploration of different system reconfiguration mechanisms and policies in MPSoCs.
- A run-time self-adaptive scheduler/manager for handling the blind adaptivity problem. The smart system scheduler tries to predict whether or not reconfiguration of the system actually is beneficial based on the active workload scenario and the status of the hardware platform. According to this prediction, the system will either be reconfigured or not. By using this

method, which is referred to *adaptivity throttling* in this thesis, unnecessary system reconfigurations can be avoided, and consequently the system efficiency can be improved.

- A hierarchical resource management mechanism which includes a hierarchical hybrid task mapping approach and a hierarchical adaptivity throttling method is implemented for a large-scale MPSoC system. Traditionally, run-time managers are either centralized or distributed. However, as a centralized approach comes with a performance bottleneck and a distributed approach leads to a high complexity, both approaches do not fulfill the requirements of embedded systems. To overcome this problem, a hierarchical resource management mechanism is presented for our target large-scale MPSoC system where the above mentioned novel hybrid task mapping approach is still applicable.

1.5.2 Thesis Overview

The content of this thesis is organised as follows. Firstly, the introduction chapter provides a general background of our research which mainly includes the motivation, the goal and the research questions. Secondly, the main part of this thesis from Chapter 2 to Chapter 5, is organised according to the research questions discussed in Section 1.4.

Chapter 2 gives the preliminary information and work that will be used in the subsequent chapters. It includes a description of the basic Sesame system-level modeling and simulation environment. Next, the formalisation of overall concepts is provided for our work. After that, the GA-based mapping DSE algorithm for effective design-time task mapping exploration is presented. It is foundational work for the research of this thesis which is applicable whenever a complex mapping performance optimisation DSE problem is presented.

Chapter 3 contains four parts. In the first part, the Sesame simulator extended with a run-time resource scheduling framework is presented. This extended simulator will be used for evaluating the techniques proposed in the remaining parts of this chapter. The second part introduces our scenario clustering based task mapping approach which is proposed to solve the scalability problem of hybrid task mapping techniques with regard to the number of workload scenarios. And the third part of this chapter gives the details of our novel hybrid task mapping approach targeting to handle both the scalability and flexibility problem of general hybrid task mapping techniques. After that, in the last part, the previously proposed two techniques are combined together to further improve the efficiency of MPSoC systems where the novel hybrid task mapping technique is applied for workload scenario mapping initialisation and the run-time on-the-fly heuristic of the scenario clustering based technique is used for dynamic QoS management during scenario execution.

The content of Chapter 4 focuses on how to solve the blind adaptivity problem as discussed in the third problem statement of Section 1.4. For this purpose, the system reconfiguration cost should be carefully considered for dynamic task remapping at run time. Consequently, this chapter firstly introduces our extended Sesame simulator that supports a flexible and efficient modeling, simulation and

exploration of different system reconfiguration mechanisms and policies in MPSoCs. After introducing this extended simulator, our proposed technique of adaptivity throttling on MPSoCs is presented in the second part of this chapter. In the last part, this technique combined with the novel hybrid task mapping approach from Chapter 3 is used to further improve the efficiency of MPSoCs.

Chapter 5 demonstrates our initial research on large-scale MPSoC systems. In this chapter, a tiled MPSoC architecture is firstly presented as our target large-scale heterogeneous MPSoC systems. After that, for this target MPSoC system, a hierarchical resource management mechanism based on our hybrid task mapping approach and adaptivity throttling technique are proposed to show that how our previously proposed techniques can be applied/scaled to a large-scale MPSoC system.

Finally, in Chapter 6, we first look back and summarize what we have achieved, and then look ahead to outline what can be accomplished next.

CHAPTER 2

Simulation-based Static Task Mapping Exploration

Static task mapping exploration plays an important role in our research for improving the adaptivity on a target MPSoC system. Such static task mapping exploration is normally solved by computational intensive algorithms. However, with the scaling of application tasks and architecture components, this static optimisation problem is becoming computational intractable by using these algorithms. To solve this problem, in this chapter, we propose a novel bias-elitist genetic algorithm that is guided by domain-specific heuristics to improve the efficiency of the static task mapping exploration. To evaluate the fitness of each mapping solution, a system-level MPSoC simulator, Sesame, has been adopted. This simulator is also a basic evaluation tool used in our work and will be introduced as a prerequisite of this thesis in the first section of this chapter. In the second section, the formalisation of overall concepts is provided for our work. After that, the GA-based mapping DSE algorithm for effective design-time task mapping exploration is presented as a foundational work of this thesis. Finally, a short summary is given for this chapter.

2.1 A System-level MPSoC Simulation Framework

Sesame [95, 124], an abbreviation for "Simulation of Embedded Systems Architectures for Multi-level Exploration", is a system-level modeling and simulation environment which aims at efficient design space exploration of embedded systems. Different with most related system simulation environments like Metropolis [11],

This chapter is based on:

- W. Quan and A. Pimentel, "Towards exploring vast mpsoc mapping design spaces using a bias-elitist evolutionary approach," in *Digital System Design (DSD), 2014 17th Euromicro Conference on,* Aug. 2014, pp. 655–658.
- W. Quan and A. D. Pimentel, "Exploring task mappings on heterogeneous mpsocs using a bias-elitist genetic algorithm," *CoRR, vol. abs/1406.7539,* 2014.

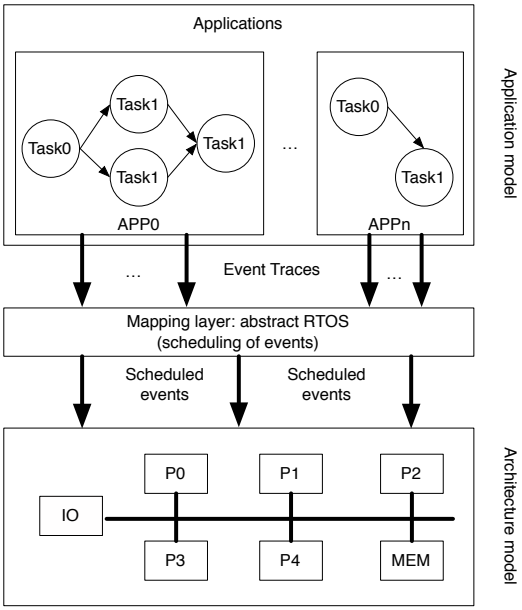


Figure 2.1: Sesame framework.

Milan [78], ARTS [66], etc., Sesame tries to push the separation of modeling application behaviour and modeling architectural constraints at the system level to even greater extents by individually considering the modeling of applications, architectures and application-to-architecture mappings. Sesame, as illustrated in Figure 2.1, recognizes separate application and architecture models, where an application model describes the functional behaviour (both computation and communication) of an application and the architecture model defines architecture resources and captures their performance/energy constraints. After explicitly mapping an application model onto an architecture model, they are co-simulated via trace-driven simulation. This allows for flexible evaluation of different applications, different underlying architectures, and different application-to-architecture mappings.

2.1.1 Application Modeling

Sesame uses Kahn Process Networks (KPNs) [56] as the Model of Computation (MoC) to specify the functional behaviour of an application. The choice of using KPNs is motivated by the target application domain of Sesame. Currently, it is mainly focused on the streaming application domain like multimedia and signal-processing applications. KPNs fit well to the streaming behaviour of this type of applications with a data-flow processing style [1]. A KPN of an application is defined as a network of concurrently running Kahn processes. The only way of communication between the processes is by blocking reading and non-blocking writing operations through Kahn channels, which are one-directional FIFO buffers of unbounded capacity between Kahn processes. According to the semantics of

[1] A stream of data passes though a series of actors and each actor processes the data and passes it on to the next actor.

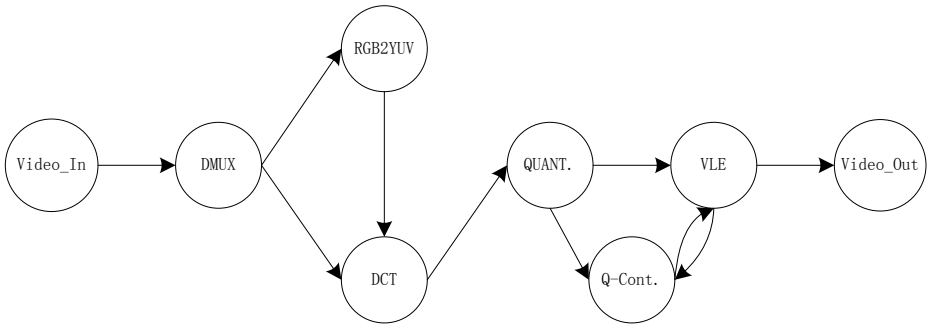


Figure 2.2: KPN for MJPEG decoder application.

KPNs, a process will not examine its input channel(s) for the presence of data and it suspends its execution whenever it tries to read from an empty channel. Unlike reads, writing to channels is always successful as the channels are defined to be infinite in size. The result of the above semantics is that the KPN MoC is deterministic: the order of the tokens sent over the channels does not depend on the order the Kahn processes are scheduled by the simulation. As a result, Sesame will produce the same output regardless of the scheduling of the processes or the architectural characteristics. This provides a lot of scheduling freedom when mapping KPN processes onto architecture models for quantitative analysis.

Figure 2.2 shows a KPN of a Motion-JPEG (MJPEG) decoder application. Internally, the Kahn processes may be implemented in any high level programming language, as long as the Kahn semantics are observed. The code of each Kahn processes is annotated with events that can be supported in the architecture model. By executing the Kahn model, these annotations cause the Kahn processes to generate traces of application events which subsequently drive the underlying architecture model. In Sesame, three basic events are supported: the communication events *READ* and *WRITE*, and the computational event *EXECUTE*. Each event has a set of arguments to describe what is done. The argument of *EXECUTE* describes which operation is performed. For instance, the execution of a Discrete Cosine Transform (DCT) in Figure 2.2 is expressed as *EXECUTE(DCT)*. For *READ* and *WRITE* events, the Kahn channel is specified and the amount of data that is communicated as, depending on the application, reading/writing may involve different communication units like a pixel or a complete video frame. The units defined for communication events could be used for controlling the granularity of events that will be simulated on the architecture model. Besides these basic events, Sesame can also be extended with other events to support a more flexible description of application behaviours. For example, the events of *STARTSCENARIO* and *ENDSCENARIO* are examples of extensions to Sesame for describing the scenario behaviour of applications when multiple applications are simulated simultaneously on the architecture model [124]. This extension is also used in our research.

2.1.2 Architecture Modeling

The architecture model of Sesame describes the hardware components of the system. It is responsible for modeling non-functional properties, like latencies and

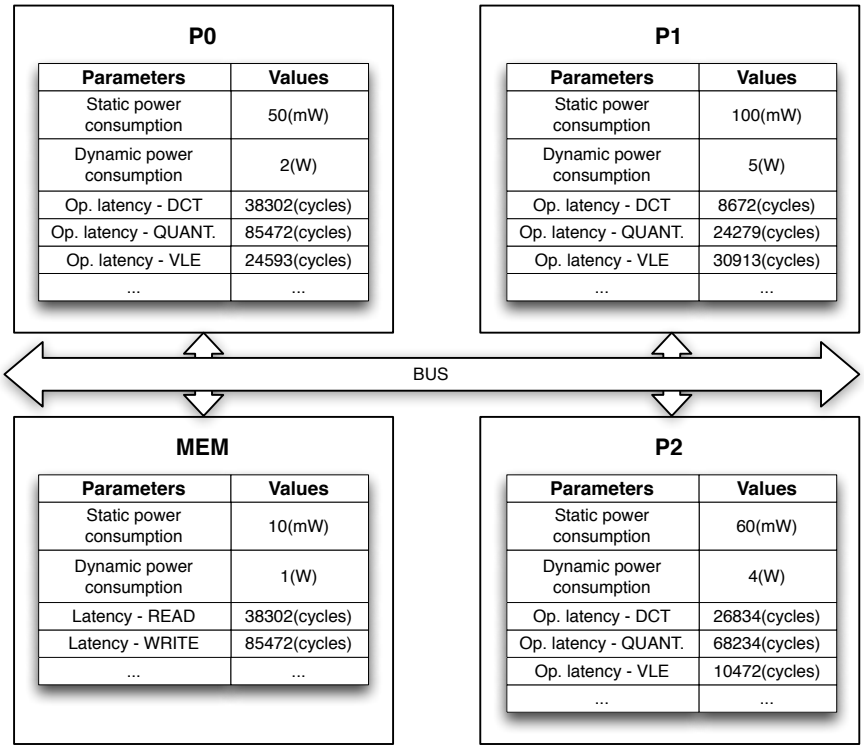


Figure 2.3: A simple parameterized architecture model of Sesame.

power consumption [124, 96], associated with the components of the system. This is possible because the functional behaviour is already captured in the application models, which subsequently drive the architecture simulation. Hardware components are described using the Pearl language [135] which is a C-based discrete event simulation language. Pearl provides easy construction of the models and fast simulation [94] by its two main features: 1) an object oriented approach for defining model components and 2) integrated primitives for communication and synchronization between model components.

The architecture models implemented in Pearl are highly parameterized black box models, which can simulate different characteristics of the components in the architecture model by changing the corresponding parameters. For example, the timing consequences of application events are simulated by parameterizing each architecture model component with an event table containing operation latencies. The table entries can include, for example, the latency of an execute event, or the latency of a memory access (*READ*/ *WRITE* event) in the case of a memory component. When simulating the architecture model, the timing characteristics of application events on each component and the whole system will be derived. According to the timing consequences and the power consumption parameters (static and dynamic) of each component, the energy characteristics of the system can also be derived [126, 99]. Figure 2.3 gives an example of a simple parameterized architecture model in Sesame.

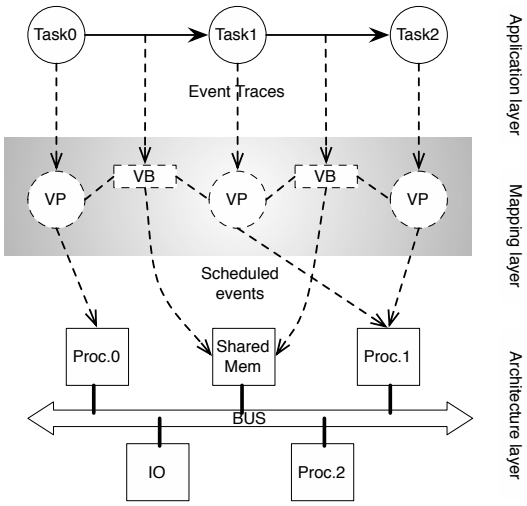


Figure 2.4: The different layers in Sesame and their connections.

2.1.3 Mapping Layer

Sesame provides an additional layer, the mapping layer, between the application model and architecture model layers to realize the trace-driven co-simulation of individual models. The main functions of this layer include: firstly, it controls the mapping of Kahn processes onto architecture model components by dispatching application events to the correct architecture model component, and secondly, it makes sure that no communication deadlocks occur when multiple Kahn processes are mapped onto a single architecture model component. The mapping layer comprises of virtual processors (VP) and virtual FIFO buffers (VB) for communication between the virtual processors. For this reason, the mapping layer is also called the virtual layer. Between the application layer and the mapping layer, there is a one-to-one relationship between the components in these two layers as shown in Figure 2.4. It means that each the Kahn process in the application model is connected to a virtual processor in the mapping layer. This is also true for the Kahn channels and the virtual FIFO buffers in the mapping layer, except for the fact that the latter are limited in size. Their size is parameterized and dependent on the modeled architecture. Connecting the application layer to the virtual layer can be done implicitly in Sesame as the structure of the mapping layer is equivalent to the structure of the application model under investigation. For connecting the mapping layer to the architecture layer, however, an explicit connection is required. There is a many-to-one relation between the mapping layer and the architectural layer. A virtual element may only be connected to one architectural component, but an architectural component may be connected to multiple virtual elements. In Figure 2.4, for example, two VPs are mapped onto the same processor *Proc.*1 and, similarly, the two VBs are mapped onto the same memory.

Under the connection definitions between the application model and the architecture model, a virtual processor in the mapping layer reads in an application trace from a Kahn process via a trace event queue or a trace file [124] generated from the application model and dispatches the events to a processing component in the architecture model. The mechanism used to dispatch application events from

a virtual processor to an architecture model component guarantees deadlock-free scheduling of the application events from different event traces [94]. In this mechanism, when a virtual processor receives a computation event, it is immediately dispatched by the virtual processor to the architecture component on which it is mapped as computation events do not cause any deadlocks. However, communication events cannot be directly dispatched to the underlying architecture model. When a virtual processor receives a communication event, it needs to consult with the corresponding FIFO buffer at the mapping layer to check whether or not the communication is safe (i.e., for read events the data should be available and for write events there should be room in the target buffer) to take place so that no deadlock can occur. The communication event can be dispatched only if it is found to be safe and otherwise the virtual processor blocks. In the architecture model, the architecture components schedule incoming events according to a given policy such as First-Come-First-Serve (FCFS) and subsequently models their timing and/or energy consequences.

2.1.4 Model Calibration

Under the above-described trace-driven simulation mechanism, Sesame is able to provide a flexible trade off between the simulation efficiency and accuracy. This trade off highly depends on the abstraction level of application events generated from the application model. This means that, if the application behaviour is abstracted by fine-grained application events (e.g. pixel block-level processing for the MJPEG application), Sesame will take longer in terms of simulation time but provides more accurate simulation results compared to the case where the application behaviour is abstracted by coarse-grained application events (e.g. frame-level processing for the MJPEG application). Sesame provides the ability of dynamically adjusting the simulation efficiency or accuracy by offering a mixed-level co-simulation technique [39, 122], called trace calibration, where parts of the system that are of particular interest may be defined at a more detailed level of abstraction than the rest of the model. By using this technique, coarse-grained application events can be refined as more detailed events and accurate values derived from low-level simulators (e.g. an Instruction Set Simulator) or even measurements on real systems can be provided to the parameter tables of each component in the architecture model. Consequently, the accuracy of the simulation could be greatly improved at the cost of consuming more simulation time. In the research of this thesis, we also use this model calibration technique to improve the simulation accuracy for our target applications where the values of events in the parameter tables of architecture components have been derived from FPGA prototyping [124, 96].

2.2 Formalisation of Basic Concepts

To facilitate the problem description for the remaining parts of this thesis, this section provides a general formalisation for some basic concepts.

2.2.1 Workload Scenario Formalisation

In the Sesame simulation framework, the applications are modeled by KPNs. Consequently, an application can be formalised as a directed graph $KPN = (P, F)$

where P is a set of processes p_i in the application and $f_{ij} \in F$ represents the FIFO channel between two processes p_i and p_j. As introduced in Section 1.4, to capture the application dynamism in a MPSoC system, we use the concept of workload scenarios which specifies the application workload in terms of the different applications that are concurrently executing and the mode of each application. We denote S as the set of all possible workload scenarios for the target applications. For a number of n target applications where each application has m execution modes, the total number of possible workload scenarios in S is $(m+1)^n - 1$. Each workload scenario $s_i \in S$ is described as a set of KPN graphs, $s_i = (..., KPN_j^k, ...)$ where KPN_j^k is the graph of app_j^k (application j, mode k) that is active in scenario s_i. Combining the KPN graphs in a workload scenario, the graph of a whole workload scenario can be expressed as $s_i = (T_i, C_i)$ where T_i is the set of tasks (i.e. processes in KPNs) in the scenario s_i and C_i represents the set of communication channels between communicating tasks. Each element in T_i and C_i, noted as t_i^{km} and c_i^{kn} respectively, represents the $m-th$ task and the $n-th$ communication channel in application app_k which is active in workload scenario s_i.

2.2.2 Hardware Architecture Formalisation

For the architecture formalisation, a MPSoC system can also be described as a graph $MPSoC = (PE, M)$, where PE is the set of processing elements used in the architecture and M is a multiset of pairs $m_{ij} = (pe_i, pe_j) \in PE \times PE$ representing a buffered communication medium, composed of a network channel (like a Bus, NoC, etc.) and a buffer located in system memory, between processors pe_i and pe_j. This general formalisation can be applied to different architectures of MPSoC systems. For example, by providing the processor type information to each element in PE, both homogeneous and heterogeneous MPSoC systems can be described under the above definition.

2.2.3 Task Mapping Formalisation

The task mapping [2] defines the allocating and binding of the underlying architecture resources to the components in a workload scenario (including the processes and the communication channels). Given a workload scenario and a target MPSoC, a correct mapping is a pair of unique assignments $(\mu : T \to PE, \eta : C \to M)$ such that it satisfies $\forall c \in C, src(\eta(c)) = \mu(src(c)) \wedge dst(\eta(c)) = \mu(dst(c))$. For each workload scenario $s_i \in S$, the possible task mappings are denoted as TM_i with each single mapping $tm_i^j \in TM_i$ complying with the mapping constraint. Under these definitions, the computation cost of task $t_i^{km} \in T_i$ and the communication cost of channel $c_i^{kn} \in C_i$ in workload scenario s_i under the task mapping of tm_i^j is represented as et_{ij}^{km} and ec_{ij}^{kn} respectively.

[2] Note that, in this thesis, we refer the term of "task" as both the computational process and the communication between processes for "task mapping".

2.3 A Novel Static Task Mapping Exploration Approach

Given a set of applications, system designers can determine a hardware platform[3] by applying different DSE approaches for architecture exploration [39, 122] under different target design constraints (performance, power consumption, cost of hardware resources and so on). On the derived hardware platform, the task mapping problem – consisting of assigning a set of application tasks to processors and binding communications between tasks to communication channels or memories in the system – plays a crucial role in achieving the application execution objective(s) such as maximising throughput and/or minimising energy consumption.

Traditionally, the mapping of applications onto the underlying architectural components of MPSoC systems has always been done in a static fashion at design time. These methods typically use computationally intensive search methods to find the optimal mapping or near optimal mapping for all applications that may run on the system. However, as workload scenarios can change dynamically at run time, such a static mapping optimised for all target applications (the most complex workload scenario) might not perform well for other workload scenarios because the hardware system is typically over dimensioned based on the worst case workload scenario. When the number of target workload scenarios is relatively small, one solution for this issue is exploring a task mapping under the target optimisation objective for each workload scenario, and then dynamically applying these pre-optimised mappings on the target system according to the change of workload scenarios. For solving each static task mapping problem, many heuristic algorithms have been proposed, which can roughly be divided into two categories: the ones that assign one task at a time like Minimum Execution Time (MET) or Minimum Completion Time (MCT) [16] and the algorithms that map all the tasks at once like Simulated Annealing [87] or Genetic Algorithms [6]. Comparing these two classes of algorithms, the former category of algorithms usually has lower algorithmic complexity, which means a shorter computing time, but they also produce poorer results. For the second category of task mapping algorithms, several investigations [16, 26, 90, 40] have shown that Genetic Algorithms (GA) can consistently generate efficient mapping solutions, also in comparison to alternative heuristic search methods like Simulated Annealing (SA), in a relatively short time period. However, for large problem sizes (i.e., search spaces) which are quite normal on modern MPSoCs where multiple applications should be supported, GAs will typically suffer from large computational costs as a significant number of solution evaluations are needed to find good solutions [115]. Therefore, it is essential to develop effective pruning techniques that can optimize the search process, allowing the design space exploration (DSE) algorithms to explore larger design spaces. To address this problem, this section presents a novel bias-elitist genetic algorithm that is guided by domain-specific heuristics to speed up the evolution process targeting maximising the mapping performance in terms of system throughput (or minimising the system execution time per unit of workload).

[3]The platform is usually optimised under the worst case execution of applications (i.e. all applications are active) to guarantee that all the target applications can be supported on the system.

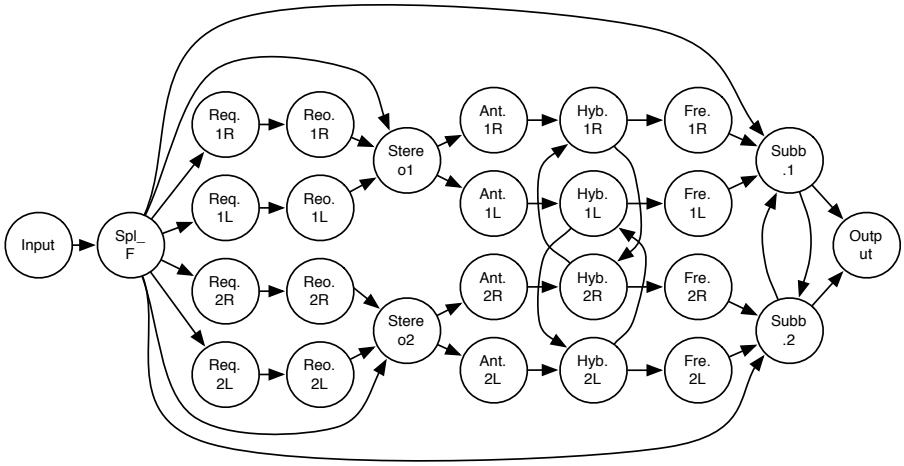
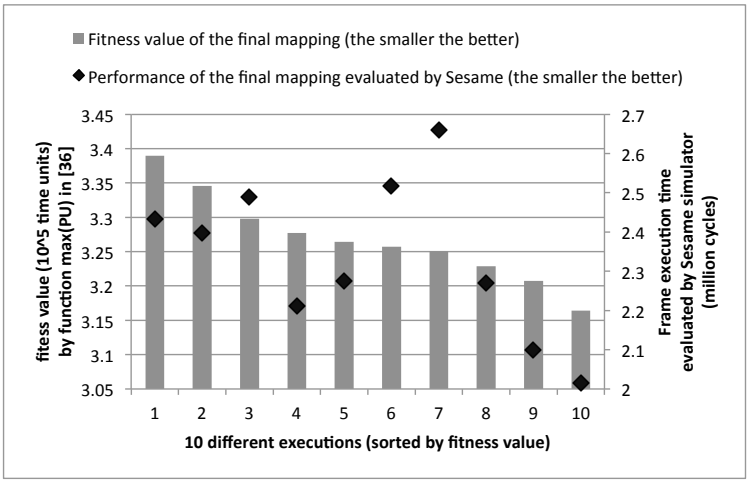


Figure 2.5: KPN for MP3 decoder application.

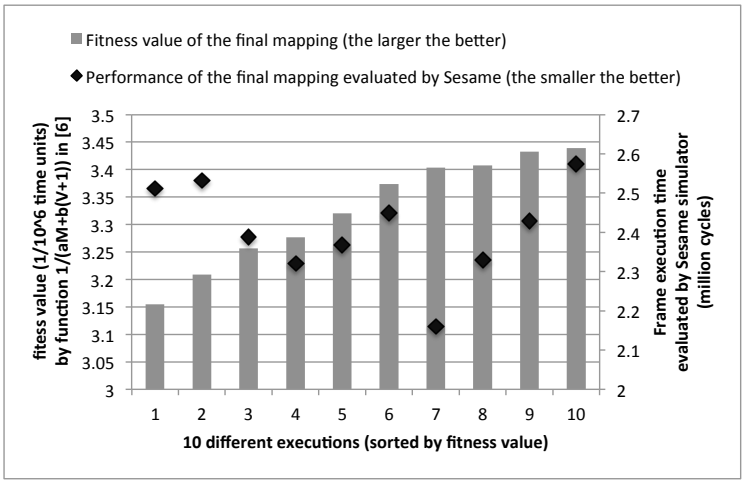
2.3.1 Motivation Examples

As a motivational example, we present a task mapping problem for which finding a good solution using a standard GA is difficult. Considering a MP3 decoder application with 27 tasks as shown in Figure 2.5, we want to find the task mapping with maximal throughput on a heterogeneous MPSoC system containing 5 processors with different computational characteristics. To solve this task mapping problem, the GA-based mapping algorithms from [36] and [6] are used first. The fitness function in [36] heuristically evaluates a mapping by analytically predicting the makespan considering both the computation and communication cost of tasks. In [6], on the other hand, the fitness function heuristically evaluates a mapping based on a makespan prediction and the processor workload difference considering only the computation cost of tasks.

Figure 2.6 shows the relationship between the fitness value (as evaluated by the fitness functions from [36, 6]) and the real mapping quality (derived from the Sesame simulator) of the final mappings as obtained by these two algorithms in 10 different executions. As we can see from Figure 2.6, the fitness functions used in these two GA-based mapping algorithms are not good enough to evaluate the fitness of individuals in our problem where both communication and computation cost of tasks on the heterogeneous MPSoC are considered. This means that a more accurate method is needed to evaluate the fitness of each mapping solution. To this end, a simulation-based approach can be considered to evaluate the fitness value of individuals. However, the simulation-based approaches usually suffer from the problem of longer evaluation times. To illustrate this, we have performed a second experiment with respect to the previous task mapping problem. In this experiment, we have used an elitist GA with random initial population, a simple one-point crossover operator and a random mutation operator to explore the possible mappings. For the fitness evaluation of the mapping solutions, the Sesame framework has now been used to obtain performance predictions of the evaluated mappings (the evaluation of a single mapping takes only in the order of a few seconds). For this particular problem, it took several tens of hours on a modern PC (2.93GHz Intel Core i7 CPU) to obtain a good mapping solution.



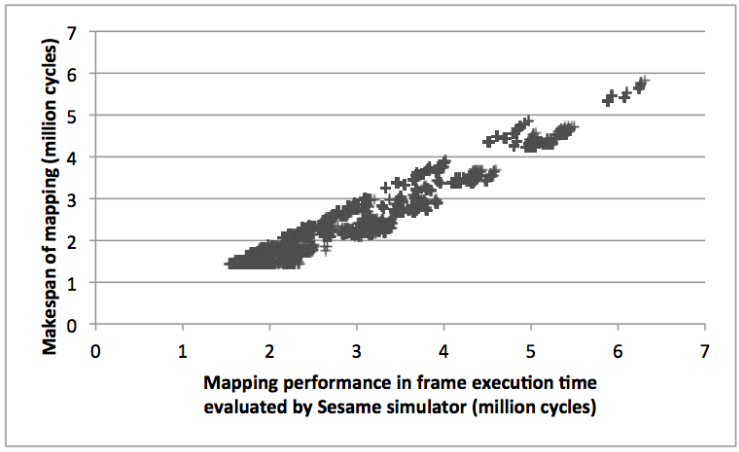
(a) The fitness value and real performance of mappings generated by the algorithm in [36]



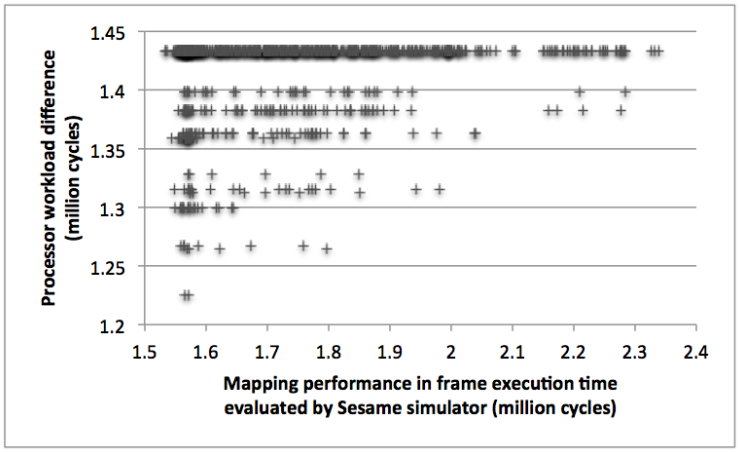
(b) The fitness value and real performance of mappings generated by the algorithm in [6]

Figure 2.6: The fitness value and real performance of final mappings.

But, as we will show in the later part of this section, the total evaluation time can be reduced by efficient pruning of the search space. To this end, our approach aims at optimizing the genetic operators in the GA that take care of deriving new individuals – representing design points – from the old individuals during search iterations. If the operators can be optimized such that they only generate a small set of chromosomes that has a high probability of containing the optimal or near optimal solutions, then the search time for a good result can be greatly reduced. In this work, we hypothesize that such an optimization of the genetic operators is possible through the exploitation of domain knowledge as captured by means of heuristics. To motivate this, please consider the following experiment in which we have exhaustively explored the mapping space of a Motion-JPEG decoder application (see Figure 2.2) for the same MPSoC system used in the previous experiments. Figure 2.7 shows the relationship between the mapping performance as evaluated by Sesame and two (analytical) performance heuristics of the same mapping solution, namely the makespan of the mapping



(a) Makespan versus real performance of the mappings in the mapping space of MJPEG



(b) Processor workload difference versus real performance of mappings with a small makespan value (y-value under 1.5 in the graph of (a)) of MJPEG

Figure 2.7: Relating real mapping performance to heuristic performance metrics for MJPEG.

and the processor workload imbalance. Although Figure 2.7a indicates that the makespan heuristic cannot predict the mapping performance with high accuracy (i.e., Figure 2.7a does not show a narrow linear line), it clearly shows a linear relationship, and thus a correlation, between mapping performance and makespan. Looking more deeply into the mappings with a smaller makespan, we can see from Figure 2.7b that the mappings with a smaller processor workload imbalance have a higher probability to be a good mapping solution. That means that good results for our mapping problem have some common properties such as a small makespan and a workload that is well balanced over processors.

Based on this observation, we propose a novel *bias-elitist genetic algorithm* in which the genetic operators have been optimized using application domain knowledge as captured by means of heuristics. We will show that this algorithm is able to find high-quality mapping solutions for applications that contain a large number of tasks, and it will do so in much shorter time frames as compared to a range of other well-known algorithms.

Algorithm 1 Bias-elitist genetic algorithm

Input:Application KPN
Output:Mapping solution
1: Normalize the target application;
2: Mapping encoding and the initial population generation;
3: Get the fitness of each initial chromosome using Sesame;
4: **repeat**:
5: Selection;
6: Crossover;
7: Mutation;
8: Evaluation;
9: **until** stopping conditions are met;
10: **return** the best solution;

2.3.2 Bias-elitist Genetic Algorithm

Our bias-elitist genetic algorithm, which is outlined in Algorithm 1, combines a form of elitism as found in classic elitist genetic algorithms with the concept of a domain knowledge guided genetic algorithm such as from [6]. It tries to find a task mapping for the target application(s) on a heterogeneous MPSoC system with the objective to maximize throughput. As the communication between tasks is considered in our mapping problem, the application KPN is normalized before evolution, as shown in line 1 of Algorithm 1. The application normalization process is used to merge the adjacent tasks that are involved in a communication cycle in the application KPN. When two adjacent tasks have a communication cycle, it means that these two tasks have a heavy communication dependence and consequently can not be explored for task parallelization. In this case, these two tasks should better be mapped onto the same processor to reduce the communication overhead. If such communication cycles exist in the target application, by applying the application normalisation, the mapping solution space can be greatly reduced. After the application has been normalized, our algorithm will be applied to explore the solution space. The details of our domain knowledge guided genetic algorithm will be explained in the following subsections.

2.3.2.1 Encoding and Initial Population

In this research, each mapping solution on a MPSoC system for the target application(s) is encoded as a string of integers. The tasks (including processes and FIFO channels) of the target application(s) are arranged in the chromosome according to the topological order in the application KPN. Each gene in the chromosome represents a unique identifier of the component in the MPSoC system (i.e., denoting the processor the task is mapped on). To simplify the chromosome of each mapping solution, in our approach, only the mappings of computational tasks (KPN processes) in the application(s) are explicitly encoded in the chromosome of a mapping solution. The mapping of FIFO channels in the application KPN(s) is implicitly encoded according to the mapping of the corresponding communicating tasks by our mapping encoder. The mapping encoder always maps

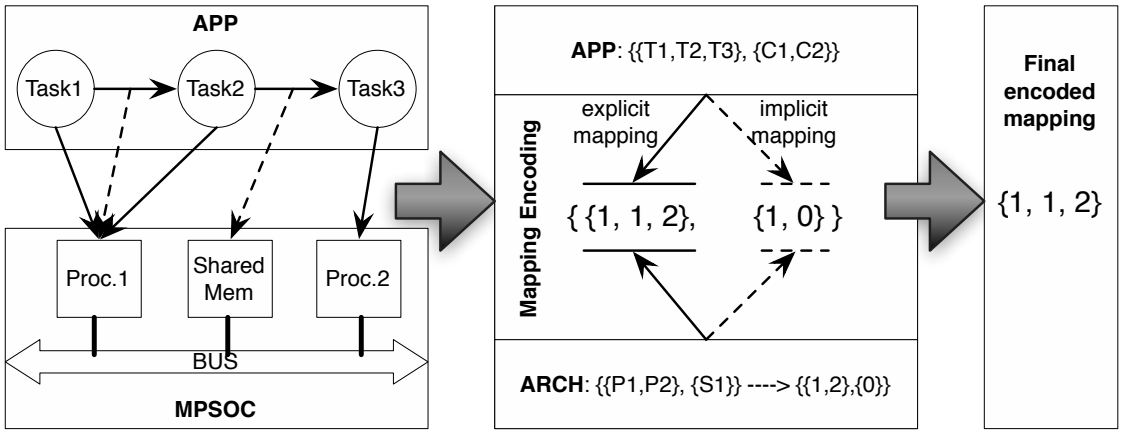


Figure 2.8: Mapping encoding.

a FIFO channel in a KPN onto the fastest communication media in the system that is available for communicating tasks. Figure 2.8 illustrates a simple example of mapping encoding. In this example, as $T1$ and $T2$ are mapped onto the same processor $P1$, the FIFO channel between these two tasks ($C1$) is mapped onto the internal local memory of $P1$ automatically to reduce the communication cost. However, $C2$ is automatically mapped onto the shared memory in the target system as $T2$ and $T3$ are found to different processors. Consequently, we do not need to explicitly encode the mapping of the FIFO channels in a chromosome which greatly simplifies the mapping problem but without loss of generality serves our purpose of showing the effectivity of our GA. However, we do want to stress that our GA can easily be extended to include explicit channel mappings, such as e.g. in [40].

In our GA, the chromosomes in the initial population are randomly generated. Moreover, we limit the size of the initial population as well as the size of the set of generated individuals during each evolution generation of the GA to reduce the simulation time.

2.3.2.2 Fitness Function

The fitness function is defined for measuring the quality of solutions. It comes from the evolutionary principle of 'survival of the fittest', where the organisms with the best characteristics for their environment have a better chance of surviving to the next generation than weaker organisms, which are less adapted to their environment [90]. The fitness function is always problem dependent. In some problems, it is hard or even impossible to define the fitness expression. In our task mapping problem, we not only need to optimize the makespan of application tasks like in a general task mapping problem (mapping independent tasks) but also the communication between tasks. Here, the resource contention and task communication should be carefully considered in the exploration. As analytical fitness evaluation approaches typically are not capable of accurately capturing such aspects (see Section 2.3.1), we deploy our Sesame system-level MPSoC simulator to accurately evaluate the fitness of each chromosome, i.e., mapping, in the population.

2.3.2.3 Selection

During each successive generation of the GA, a proportion of the existing population is selected to breed a new generation. Individual solutions are selected through a fitness-based selection process, where fitter solutions are typically more likely to be selected. Our algorithm uses a roulette wheel selection method [85] in which the best chromosomes are more likely to be selected but the poorer chromosomes also have a small chance to be picked. We should note that in this chapter we only consider a single-objective optimization problem. For multi-objective optimization problems (e.g., simultaneously optimizing performance and power consumption), the well-known selection approaches from the NSGA-II [35] or SPEA-II GAs [141] would be good options.

To control the population size in each generation, we use a strategy in which the best chromosome from the current population and $n-1$ chromosomes from the newly generated population are selected as the n survived individuals to breed the next new generation. The rationale behind this is that we aim at increasing the diversity of chromosomes in the mapping space that will be searched by keeping as few as possible old individuals in the new population. Therefore, in contrast to a general elitist GA, where the elitists in each generation will survive in the next generation, our GA only preserves the best individual in each generation. It is an extreme instance of an elitist GA. The fact that we refer to our GA as a *bias*-elitist GA will be explained in the next section.

2.3.2.4 Genetic Operators

To generate a new generation from the selected chromosomes, two genetic operators – crossover and mutation – are applied. In our algorithm, we have improved the mutation operator so that the algorithm can more quickly find better solutions. For the crossover operator, which produces a new pair of chromosomes from a selected pair of chromosomes, we apply a standard one-point crossover. We have chosen this operator because it is simple and produces similar results compared with other crossover operators like two-point crossover, uniform crossover, cycle crossover and so on [86].

The mutation operator is an essential part of our GA. It allows the GA to search new areas in the solution space. There are various methods of implementing the mutation operator, and the easiest one is the random mutation where a random task is re-assigned to a random processor. In our algorithm, we deploy a heuristic guided mutation operator that optimizes the mappings using domain knowledge. More specifically, the mutation operator considers the affinity of tasks with respect to processors, the communication cost between tasks, and the differences of processor workloads. The details of our mutation operator are outlined in Algorithm 2. By applying the mutation operation, a new chromosome will be derived through one of the following three approaches: task migration (lines 1-12), processor switching (lines 15-22) or a Minimum Completion Time (MCT) algorithm (lines 24-25).

At the beginning of the mutation, the task migration method will be used to find a new chromosome based on the input chromosome. In this process, the usage of each processor U_k under a given task mapping is calculated by Equation 2.1 in the function at line 1.

Algorithm 2 Heuristic guided mutation

Input:C (old chromosome)
Output:C^* (new chromosome)
1: $PU = usage(C)$;
2: $x =$ index of processor with $max(PU)$;
3: **for** task p_i mapped onto processor pe_x:
4: **for** processor pe_y different with pe_x:
5: $C' =$ migrate p_i from processor pe_x to pe_y;
6: $PU' = pusage(C')$;
7: **if** $max(PU') <= max(PU)$:
8: $MBF.append(B_i^{xy})$;
9: **if** array MBF is not empty:
10: $p_k, pe_k =$ task and target processor with maximal migration benefit $(max(MBF))$;
11: $C^* =$ migrate p_k from processor pe_x to pe_k;
12: **goto** step 1, start with the new mapping C^*;
13: **else**:
14: **if** no new mapping found in the previous steps:
15: **for** processor pe_y different than pe_x;
16: $C' =$ switch the tasks mapped onto pe_x and pe_y;
17: $PU' = pusage(C')$;
18: **if** $max(PU') <= max(PU)$:
19: $SBF.append(max(PU'))$;
20: **if** array SBF is not empty:
21: $pe_k =$ processor with $min(SBF)$;
22: $C^* =$ switch the tasks mapped onto pe_x and pe_k;
23: **else**:
24: shuffle the order of tasks in chromosome;
25: $C^* =$ generate new mapping using the MCT algorithm based on the shuffled task order;
26: **return** C^*;

$$U_k = \sum_{t_k^i \in T_k} et_k^i + \sum_{c_k^j \in C_k} ec_k^j \tag{2.1}$$

where T_k is the set of tasks mapped on processor pe_k, C_k is the set of task communication channels connected with the tasks in T_k. The symbols of et and ec represent the cost of the corresponding computation task and communication channel under the target task mapping as defined in Section 2.2.3.

$$B_i^{xy} = M_i^x - M_i^y \tag{2.2a}$$

$$M_i^r = et_i^r + \sum_{\substack{c_{ij}^{rs} \in C_i \\ t_i \mapsto pe_r, t_j \mapsto pe_s}} ec_{ij}^{rs} \tag{2.2b}$$

where C_i is the set of communication channels connected with task t_i. The symbol $a \mapsto b$ means a mapped onto b.

Lines 3-8 of Algorithm 2 try to find a task (among the tasks mapped onto the most heavily loaded processor) that has a maximal "migration benefit" under the condition of line 7. This task migration benefit, with regard to task t_i migrated from processor pe_x to pe_y, is labeled as B_i^{xy}. It is calculated by Equation 2.2 where M_i^x and M_i^y represent the cost of t_i on pe_x and pe_y respectively. Here, the cost not only considers the task computation time but also the accumulated communication costs of the task in question.

If a task can be found for migration after the steps in lines 3-8, lines 10-11 in Algorithm 2 will generate a new mapping by migrating this task to the corresponding target processor. Subsequently, the above process is repeated – using the new mapping as input – until no new mapping can be found anymore.

However, if the above task migration approach cannot find a new chromosome, then the processor switching method will be applied to the input chromosome. As shown in lines 15-22 in Algorithm 2, the new chromosome will be generated by exchanging the tasks mapped onto the heaviest loaded processor with the tasks mapped onto the processor which satisfies the conditions on line 18 and line 21 (the processor that will maximally reduce the value of $max(PU)$ by processor switching).

In the case that no new mapping can be found by using any of the two previous approaches, a heuristic-based random mutation operator will be applied. A totally new chromosome, which means that all the genes in the chromosome are different from the ones in the input chromosome, might be generated in this approach. The heuristic used for generating a new chromosome is the Minimum Completion Time (MCT) algorithm. The MCT algorithm assigns each task, in arbitrary order, to the processor with the minimum expected completion time for that task [8]. Different task assignment orders will produce different mapping results. Therefore, each time before generating a new chromosome using MCT, the task order in the chromosome is shuffled. Consequently, different well-balanced chromosomes will be added to the new population of our GA. This helps our GA to explore the mapping space with more gene diversity and prevents our GA from getting stuck in a local minimum.

A new chromosome generated by our mutation operator has a bias towards design points that are makespan optimized and/or workload balanced. This explains the name *bias-elitist GA*. The task migration approach can optimize both the makespan and the processor workload variation. However, the processor switching approach is supplementary to the task migration approach for optimizing the mapping makespan in situations such as illustrated in Figure 2.9. When a chromosome is selected for mutation, Algorithm 2 will first try to optimize the mapping makespan and the processor workload balance using the task migration method. However, if this does not succeed (e.g., when the input chromosome already represents a well-balanced system workload), then the processor switching method will be applied to the input chromosome to further improve the mapping makespan. If no improved mapping can be derived from the input chromosome using either method, then the MCT algorithm will be used to generate a new well-balanced mapping. By applying this domain knowledge guided mutation operator, the search space of our GA will be pruned to only those mappings with a high likelihood of being high-quality mappings. As we will show in the experimental section,

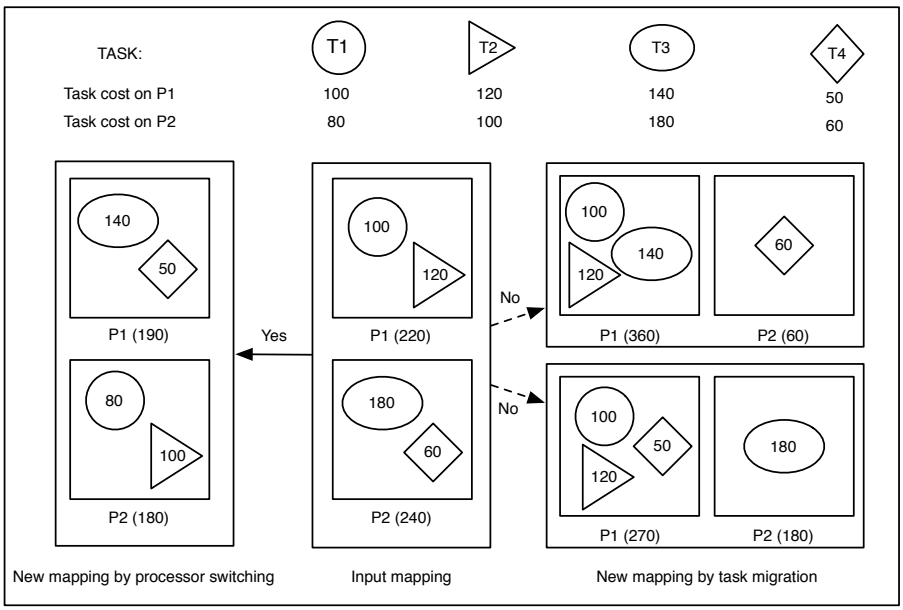


Figure 2.9: A simple example of improving the mapping makespan using processor switching.

this results in a much more efficient and effective search algorithm that allows for producing good mapping solutions in relatively short evolution times.

2.3.2.5 Termination

With respect to the stopping conditions for our GA, two conditions are used: (1) if the best solution has not changed after a pre-defined number of generations, then our GA will terminate automatically and (2) a maximum number of generations is adopted to guarantee that the evolution process will stop. Our bias-elitist GA aims at reducing the required (maximum) number of iterations as much as possible while still yielding good solutions. The above termination conditions are also applied to the other GAs that are studied in our experiments in the next section.

2.3.3 Experiments

2.3.3.1 Experimental Setup

For our experiments, we have selected a real multi-media application to investigate various aspects of our GA: a MP3 decoder consisting of 27 application processes (tasks). The target architecture considered in our experiments consists of 5 heterogeneous processors and 1 IO processor (for IO tasks). These processors are connected via a bus to a shared memory. In the MP3 task mapping problem, the total number of possible mapping solutions is $2.98 * 10^{17}$. Our Bias-Elitist Genetic algorithm (BEG) and several other algorithms will be used to explore this vast solution space to find the (near) optimal mapping with the objective to maximize the throughput. As our BEG algorithm is not limited to only solving the mapping problem for single applications, a multi-application mapping case – considering

Table 2.1: Parameters of genetic algorithms

Parameter	**Experiment 1**		**Experiment 2**	**Experiment 3**
	EG/GA3SM	BEG	all GAs	all GAs
Initial pop. size	128	8	8	128
Generation pop. size	128	8	8	128
Crossover prob.	0.7	0.7	0.7	0.7
Mutation prob.	0.8	0.8	0.8	0.8
Max. # of generations	2048	128	128	128

a Motion-JPEG encoder and Sobel filter for edge detection in addition to the MP3 decoder – is studied as well. There, we consider the maximization of system throughput when multiple applications are active simultaneously. The total number of possible mapping solutions in this multi-application mapping problem is $2.91 * 10^{24}$.

2.3.3.2 Single-application Task Mapping

For the purpose of comparison, three other mapping algorithms are studied as well: a general Elitist Genetic (EG) algorithm [36], a Genetic Algorithm with a 3-Step Mutation (GA3SM) [6] and Output-Rate Balancing (ORB) [22] which aims at balancing the computation and communication load of each processor. For the genetic algorithms (BEG, EG and GA3SM), the parameters in the experiments of single-application task mapping are listed in Table 2.1. The parameters of each GA are optimized for each experiment. Notice that the parameter of mutation probability used in our experiments is a chromosome-level concept[4]. It differs from the mutation probability used in typical GAs which is considered at the gene level[5] and is usually small (< 0.1). In our experiments, the gene-level mutation probability only exists in the EG algorithm and its value is 0.05. For the purpose of a fair comparison, the same randomly generated initial population is provided to the EG and BEG algorithms. For the GA3SM algorithm, the initial population is derived by replacing the worst individual in the randomly generated initial population with the result of the Min-Min heuristic. This is according to the original GA3SM algorithm. The results of all experiments have been averaged over 10 execution runs to deal with the stochastic behaviour of the GAs. For all experiments, we have used a PC with a 2.93GHz Intel Core i7 CPU.

In the first experiment, the original EG and GA3SM algorithms are compared with our BEG algorithm. This means that EG and GA3SM use their own analytical fitness functions to evaluate the fitness value of each chromosome. However, for our BEG algorithm, the fitness value of each chromosome is evaluated using the Sesame simulator. For our BEG algorithm, we have deliberately chosen a small population and generation size to keep the computational costs of the search as low

[4]Chromosome-level mutation probability: the likelihood of mutating a particular chromosome.

[5]Gene-level mutation probability: the likelihood of mutating each gene (bit) of a chromosome in mutation.

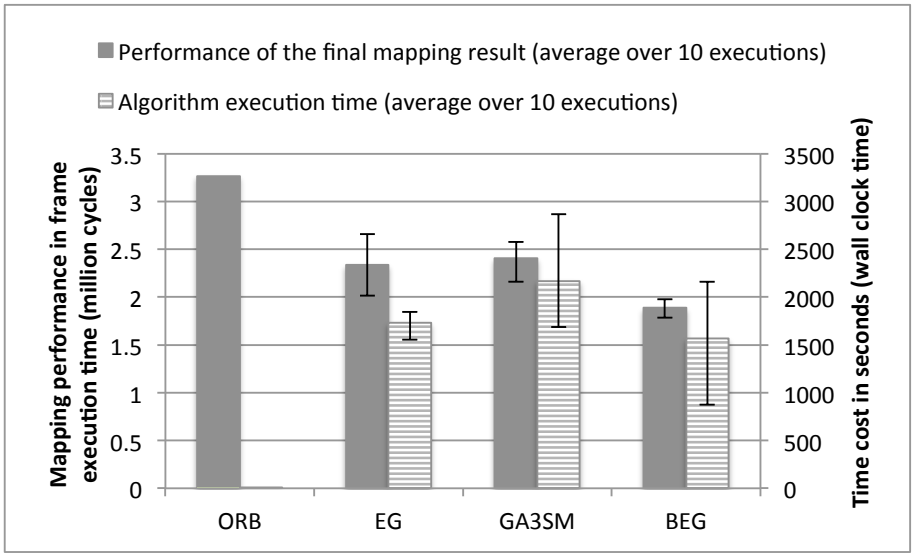


Figure 2.10: The quality of final mapping and search performance for different algorithms in the single-application task mapping problem of MP3.

as possible. Figure 2.10 shows the quality (frame execution time[6]) of the produced mapping solution and the average execution cost for each algorithm (averaged over 10 runs). Here, the ORB algorithm is only executed once as it does not have any stochastic behaviour like the GAs. From this figure, we can clearly see that the EG and GA3SM algorithms cannot generate final mappings as good as the BEG algorithm. Moreover, on average, the EG and GA3SM algorithms also spend more time on finding their final solution. Apparently, the heuristic ORB method takes the least time to get a final mapping solution which is negligible compared with the execution times of the other algorithms. However, the final mapping derived by ORB is much worse than the ones generated by the GAs.

As the analytic fitness functions used in the original EG and GA3SM algorithms are less accurate than the Sesame-based evaluations in our BEG algorithm, we have also adapted the EG and GA3SM algorithms to use Sesame to evaluate the fitness of chromosomes. These Sesame-based EG and GA3SM algorithms are used in the remainder of the experiments. In the second experiment, we compare our BEG algorithm to the (Sesame-based) EG and GA3SM algorithms on three aspects: (1) the quality (frame execution time) of the final mapping solution, (2) the algorithm execution time and (3) the convergence behaviour of an algorithm.

Table 2.2 shows the quality of the final mappings derived from the different algorithms as well as the algorithm execution cost of the search algorithms. From Table 2.2, we can see that our BEG algorithm can produce much better solutions than the other GAs. Our BEG algorithm also takes less time to find the final mapping solution as compared to the other two algorithms. The reason for this is that our BEG algorithm can converge much faster than the other two GAs, as shown in Figure 2.11. This graph shows the convergence behaviour of the execution run for each algorithm that produced the best final solution out of 10

[6]Here, as only a single application is considered, the frame execution time is defined as the execution time of the target application for processing a single frame/unit of workload.

Table 2.2: Comparison of final mapping quality in Frame Execution Time (cycles, the smaller the better) and algorithm execution cost (seconds) of GAs with small population size.

	EG	GA3SM	BEG
Max. FET	2342502	2218522	1979684
Min. FET	2022538	1911142	1784318
Average FET	2197809	2064753	1885810
Max. cost	4897	3074	2160
Min. cost	2145	1637	875
Average cost	3217	2245	1567

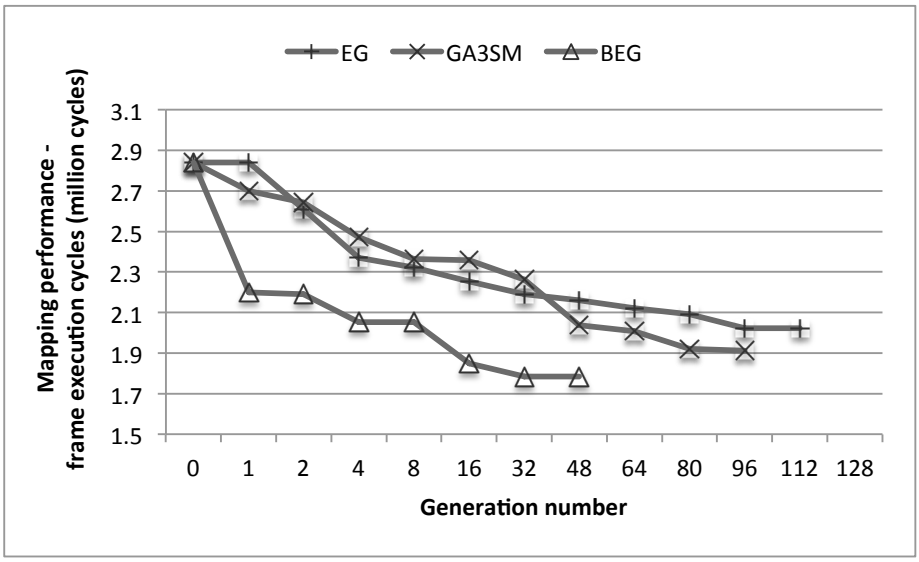


Figure 2.11: The convergence behaviour of each GA with a small population size.

runs. Considering all 10 runs, our BEG algorithm generated the final mapping solution between the 8th search generation (corresponding to minimal time cost in Table 2.2) and the 35th generation (corresponding to the maximal time cost in Table 2.2). For the EG and GA3SM algorithms, however, the final mapping solutions were found between the 9th – 95th and 12th – 82th search iterations respectively.

In the third experiment, we studied the behaviour of each GA using a larger search space by increasing the population size. The results are shown in Table 2.3 and Figure 2.12. As shown in Table 2.3, our BEG algorithm again outperforms the other algorithms with respect to the quality of the final mapping solution. Compared with the second experiment, each algorithm produces better mapping results. More specifically, the EG, GA3SM and BEG algorithms improve the mapping quality in frame execution time of the best mapping solution (Min. FET in Table 2.2 and Table 2.3) by 7.9%, 7.4% and 1.2%, respectively. However, these mapping solution improvements come at the expense of a much higher exploration

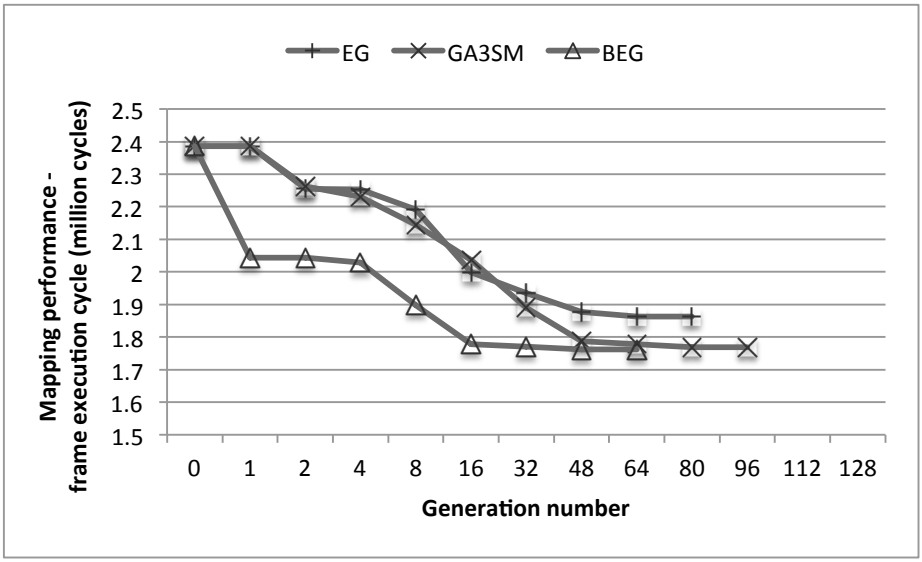


Figure 2.12: The convergence behaviour of each GA with a large population size.

time. The corresponding search times of EG, GA3SM and BEG increase by 7.1, 15.4 and 45.8 times, respectively. To provide more insight in the large increase of search time for our BEG algorithm, Figure 2.12 again shows the convergence behaviour. It shows that the BEG algorithm already produces a mapping solution after only 16 generations that is close to the final mapping solution in terms of quality. However, it takes nearly 40 more generations to derive a slightly better final result. Considering this and the previous experiment together, we can see that our BEG algorithm always yields the best solutions, and on top of this, it can already find a good mapping result in a relatively short time by reducing the population size. The EG and GA3SM algorithms, on the other hand, require algorithm execution times that are about an order of magnitude higher to find similar good mapping results as our algorithm.

Table 2.3: Comparison of final mapping quality in Frame Execution Time (cycles) and algorithm execution cost (seconds) of GAs with large population size.

	EG	GA3SM	BEG
Max. FET	2030686	1953746	1821842
Min. FET	1862988	1768808	1762048
Average FET	1943892	1847738	1779620
Max. cost	42729	55002	47824
Min. cost	21342	27814	29778
Average cost	33381	43274	39496

Table 2.4: Comparison of final mapping quality in Total Execution Time (cycles) and algorithm execution cost (seconds) of GAs for solving the multi-application mapping problem.

	EG	GA3SM	BEG
Max. TET	13200633	11857216	11288893
Min. TET	10831581	10705250	9193485
Average TET	11804280	11209070	10120488
Max. cost	7965	6174	3331
Min. cost	3844	2203	1499
Average cost	6314	4236	2499

Table 2.5: Parameters of genetic algorithms

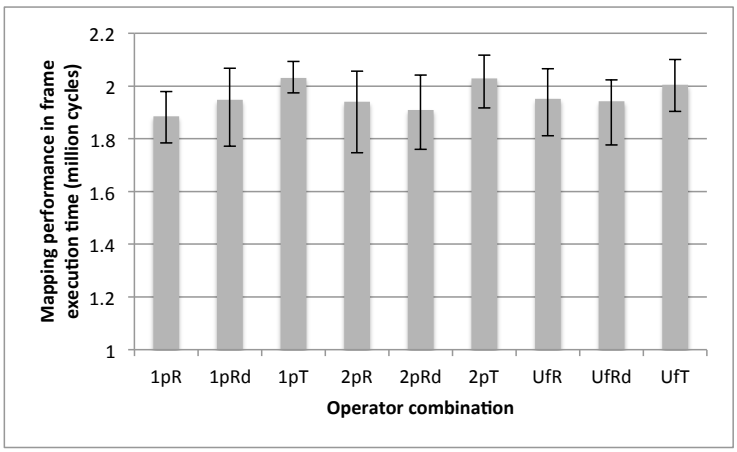
Parameter	**Experiment 5**	**Experiment 6**
	BEG	BEG
Initial pop. size	8	8
Generation pop. size	8	8
Crossover prob.	0.7	0.7
Mutation prob.	0.8	0.1–1.0
Max. # of generations	128	128

2.3.3.3 Multi-application Task Mapping

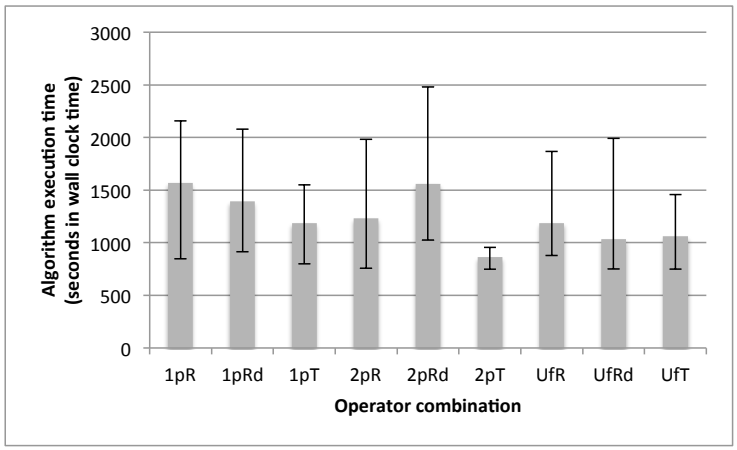
In this experiment (the fourth experiment), we investigate our BEG algorithm by solving a multi-application task mapping problem in which the MP3 decoder (27 application tasks), a Motion-JPEG encoder (8 application tasks) and a Sobel filter for edge detection (6 application tasks) will be mapped onto our previously described target platform. The quality of the final mapping (total execution time) and the algorithm execution cost are compared again for the BEG, EG and GA3SM algorithms. The parameters for each algorithm are the same as in the second experiment from the previous section (see Table 2.1). The experimental results are shown in Table 2.4. As can be seen from the results, our BEG algorithm again produces better solutions in a much shorter time frame than the other two GAs.

2.3.3.4 Sensitivity to BEG implementation choices

To study how the other operators (beside the mutation operator) like the crossover and selection method influence the behaviour of our algorithm, we have applied different operator combinations to BEG. For this experiment (experiment 5), we focus again on the MP3 decoder application, The GA parameters are shown in Table 2.5. For the crossover operator, the one-point (1p), two-point (2p) and uniform crossover (Uf) are used in this experiment. With regard to the selection approach, the roulette wheel (R), random (Rd) and tournament (T) selection



(a) Performance of final mappings (10 executions) generated by the BEG algorithm using different operators

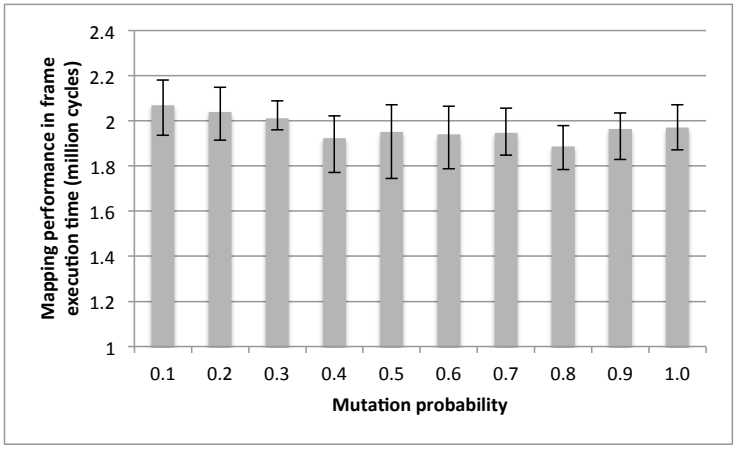


(b) Algorithm execution time cost (10 executions) of the BEG algorithm with different operators

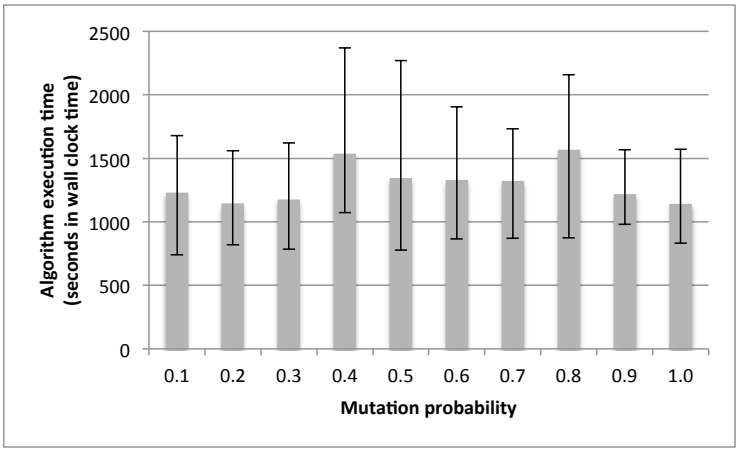
Figure 2.13: Comparison of final mapping performance and algorithm execution time cost of the BEG algorithm with different operators.

methods have been studied. Figure 2.13 shows the results of the final mapping quality and algorithm execution time cost for the different operator combinations. In this figure, the x-axis contains the operator combinations. For example, 1pR represents one-point crossover and roulette wheel selection. In Figure 2.13a, each bar represents the average mapping performance of the final mapping over 10 executions. From Figure 2.13, we can see that the two-point crossover besides the one-point crossover and the random selection besides the roulette wheel selection also work well for our BEG algorithm. However, the uniform crossover and tournament selection show poorer results for our BEG algorithm. For the algorithm execution cost shown in Figure 2.13b, a trend can be observed that shows that higher quality final mappings generally need more time for our algorithm, irrespective of what operators are used.

In the last experiment (experiment 6), the impact of the mutation probability in our BEG algorithm is investigated. In a general GA, the probability of crossover and mutation should be well-tuned to the problem at hand as it may greatly influence the convergence speed and the exploration time of the algorithm. However,



(a) Performance of final mappings (10 executions) generated by the BEG algorithm with different mutation probabilities



(b) Algorithm execution time cost (10 executions) of the BEG algorithm with different mutation probabilities

Figure 2.14: Comparison of final mapping performance and algorithm execution time cost of the BEG algorithm with different mutation probabilities.

as the crossover operator is not the main focus of this research, we will not change the crossover probability in this experiment. The mutation probability changes from 0.1 to 1.0. The other parameters for the BEG algorithm can again be found in Table 2.5. The results are shown in Figure 2.14 where Figure 2.14a shows the final mapping quality derived for a specific mutation probability and Figure 2.14b gives the corresponding algorithm execution time cost. In Figure 2.14a, we can notice that when the mutation probability is small (below 0.4), the BEG algorithm performs like a general genetic algorithm (EG) with respect to the final mapping quality. However, the algorithm execution cost is less than a general GA (EG takes 3,217 seconds on average) which can be seen in Figure 2.14b. On the other hand, if the mutation probability is very high (higher than 0.8), our BEG algorithm also yields less high-quality results even though the algorithm execution cost is reduced (because of a faster convergence) compared with BEG with a lower mutation probability. The reason might be that the algorithm will get stuck in the mapping solution space that only contains the mappings with the best makespan

and processor workload balance, and the algorithm may not have the chance to explore mappings which are slightly worse in terms of makespan and workload balance. For this particular MP3 task mapping problem, our algorithm can find good results when the mutation probability is between 0.4 and 0.8. Notice that, even though the mutation probability of our BEG algorithm is fixed in this research, some adaptive probability adjusting strategies [121, 107] can be applied to further optimize our algorithm.

2.3.4 Related Research

In recent years, much research has been performed in the area of task mapping for embedded systems. [115] gives a nice survey of the existing mapping methodologies.

In the context of static mapping performance optimization, some classic algorithms such as Simulated Annealing (SA) [88, 63], Genetic Algorithm (GA) [26, 6, 90], Tabu Search [69] and Integer Linear Programming (ILP) [53] have been proposed. Among these algorithms, the GA is considered to be a good mapping algorithm because it can obtain a good result in a relatively short time period [16]. There are different forms of GAs that can be used to obtain a better solution. For instance, [133] proposes a heuristic-based hybrid genetic-variable neighborhood search algorithm for guiding the search process and [90] uses eight heuristics to initialize the GA population for getting better solutions. Alexandrescu et al. [6] propose a GA with a 3-Step Mutation which aims at increasing the solution's convergence rate by using a combination of methods to mutate a chromosome. In contrast to these GAs, our domain-knowledge guided GA is proposed to solve the large scale task mapping problems on the heterogeneous MPSoC systems where the computation and communication cost of tasks and resource contention in the system are carefully considered in the evolution process.

In our approach, a simulator is used to evaluate the fitness of each chromosome which greatly increases the total evolution time. Consequently, design space pruning techniques need to be considered in our work. In this field, the most recently related work is from [123] where the system-level design space is implicitly pruned by exploiting domain knowledge in their GA-based DSE. In contrast to our work, however, the work of [123] only deals with homogeneous systems and enriched their GA with a "mapping distance" based crossover operator. Some other approaches, for example [10, 72, 92], perform design space pruning via meta-model assisted optimization, which combines simple and approximate models with more expensive simulation techniques. Another class of design space pruning is based on hierarchical DSE (e.g., [40, 58, 55, 54]). In these approaches, DSE is first performed using analytical or symbolic models to quickly find the interesting parts in the design space, after which simulation-based DSE is performed to more accurately search for the optimal design points.

2.3.5 Conclusion

The large scale task mapping problem is hard to solve especially when the communication between tasks also needs to be considered. Even though genetic algorithms have a proven track record in solving such problems, these algorithms

still need to be carefully designed in order to obtain high-quality solutions in an acceptable time. In this section, we have proposed a bias-elitist genetic (BEG) algorithm where the mutation operator has been optimized for our task mapping problem. More specifically, we have added domain-specific heuristics as well as a Minimum Completion Time heuristic to the mutation operator. In addition, the selection method in our genetic algorithm has also been tailored for the purpose of finding a good mapping in a short time period. In various experiments, different state-of-the-art algorithms have been compared to our BEG algorithm. These experimental results clearly confirm the effectiveness of our algorithm.

2.4 Summary

In this chapter, we firstly introduced the basic modeling and simulation environment of Sesame which is a system-level MPSoC simulator. It has separate models for applications, architectures and the mapping between them. That makes it able to provide a flexible evaluation for MPSoCs. In the research of thesis, this simulation framework is deployed and extended (in the following chapters) as an evaluation tool for different MPSoC systems with different target applications and application-to-architecture mappings. After that, a general formalisation of workload scenarios, hardware architecture and task mapping has been provided. The problem definitions in the remaining part of this thesis are based on this formalisation. Next to this section, the GA-based mapping DSE algorithm for effective design-time task mapping exploration has been presented. This algorithm has been proposed to solve the large-scale task mapping problem under a single optimisation objective like performance. It provides a foundation for the research of this thesis that is applicable whenever a complex mapping performance optimisation DSE problem is presented. In the state-of-the-art task mapping solutions as mentioned in Section 1.3.2, for a MPSoC system with a limited number of workload scenarios, after applying such a static mapping exploration at design time, these pre-optimised mappings could be applied at run-time according to the change of workload scenarios. By using this kind of approaches, the system's efficiency could be greatly improved compared to a traditional MPSoC system that solely uses a static task mapping approach.

CHAPTER 3

Novel Hybrid Task Mapping Approaches

In the previous chapter, we have presented a static exploration approach for the complex design-time task mapping problem. In general hybrid task mapping approaches, the design-time optimised mappings derived from such static DSE will be applied at run-time according to the change of workload scenarios on a MPSoC system. However, with the increase of the number of target workload scenarios on the target MPSoC, the mappings that should be optimised and the memory usage for storing these mappings will become intractable. It means that there is a scalability problem with regard to the number of workload scenarios by exploring and applying static mappings at scenario level. Besides that, as mentioned in Section 1.4, the flexibility with regard to supporting new applications is another issue (design-time analysis needs to be redone entirely) in this kind of approaches. In this chapter, we will focus on solving these problems of most hybrid task mapping solutions for MPSoC systems with coarse-grained workload scenarios. Here, a coarse-grained workload scenario means that the execution duration of a workload scenario is long enough to neutralise or even ignore the system reconfiguration overhead caused by task remapping. Therefore, in this chapter, the actual run-time system reconfiguration cost (overhead of task migrations) during task remapping is not explicitly considered but will be further studied in the next chapter .

In this chapter, we will concentrate on the run-time optimisation of task mappings based on pre-optimised mappings derived at design time. For this purpose,

This chapter is based on:

- W. Quan and A. Pimentel, "A scenario-based run-time task mapping algorithm for mpsocs," in *Proceedings of the 50th Annual Design Automation Conference (DAC '13),* New York, NY, USA: ACM, 2013, pp. 131:1–131:6.
- W. Quan and A. D. Pimentel, "An iterative multi-application mapping algorithm for heterogeneous mpsocs," in *Embedded Systems for Real- time Multimedia (ESTIMedia), 2013 IEEE 11th Symposium on,* Oct. 2013, pp. 115–124.
- W. Quan and A. D. Pimentel, "A Hybrid Task Mapping Algorithm for Heterogeneous MPSoCs," in *ACM Trans. Embedd. Comput. Syst.,* vol. 14, no. 1, Jan. 2015, pp. 14:1–14:25.

the basic Sesame simulator should be extended to support the simulation of dynamic application behaviour. In the first section of this chapter, we will introduce the run-time support of the Sesame simulator, which has been extended with a run-time resource scheduling framework. With the extended Sesame simulator, we are able to study different mapping optimising techniques for MPSoC systems with complex and dynamic workload behaviour.

In the second section, a scenario clustering based task mapping approach is proposed to solve the scalability problem of hybrid task mapping techniques with regard to the number of workload scenarios. In this approach, we firstly divide the target workload scenarios into different scenario clusters and explore a mapping under the optimisation goal for each scenario cluster. After that, based on the cluster-level mapping information, the run-time system manager performs mapping customisation using a run-time on-the-fly heuristic to further optimise the performance of applications on the target MPSoC system. The run-time mapping customisation is triggered by the violation of application execution objectives. By using the proposed approach, the number of task mappings and consequently the time cost for mapping exploration at design time and the memory storage for storing the pre-optimised mappings at run time can be greatly reduced. In addition, better QoS with regard to application performance requirements can be achieved on the target MPSoC system.

To handle both the scalability and flexibility problem of general hybrid task mapping techniques, a novel hybrid task mapping approach is presented in the third section of this chapter. In this proposed approach, the scenario-level task mapping problem is solved by a divide-and-conquer technique where the complex scenario-level mapping problem is firstly broken down into small application-level mapping problems at design time, and the application-level mapping solutions are then dynamically combined and further optimised to give a complete solution for a workload scenario at run time. Different with the approach proposed in the second section where both the change of inter-application scenarios and the violation of application-specific performance objectives trigger system reconfigurations at run time, the system reconfiguration of this approach is triggered by the change of workload scenarios (both inter- and intra-application scenarios). It optimises both performance and/or energy consumption for newly detected workload scenarios according the system execution mode of the target system.

In the fourth section, based on the techniques proposed in the previous two sections, we combine these two approaches together to further improve the efficiency of MPSoC systems where the novel hybrid task mapping technique from the third section is applied for the mapping initialisation of a detected workload scenario and the run-time on-the-fly heuristic from the second section is extended for dynamic QoS management during the execution of a workload scenario on a heterogeneous MPSoC system.

After that, the related research and a short summary for the work of this chapter are presented.

3.1 Run-time Supports in Sesame

As discussed in section 2.1, Sesame provides the ability of modeling and simulating different applications on different architectures under different mappings.

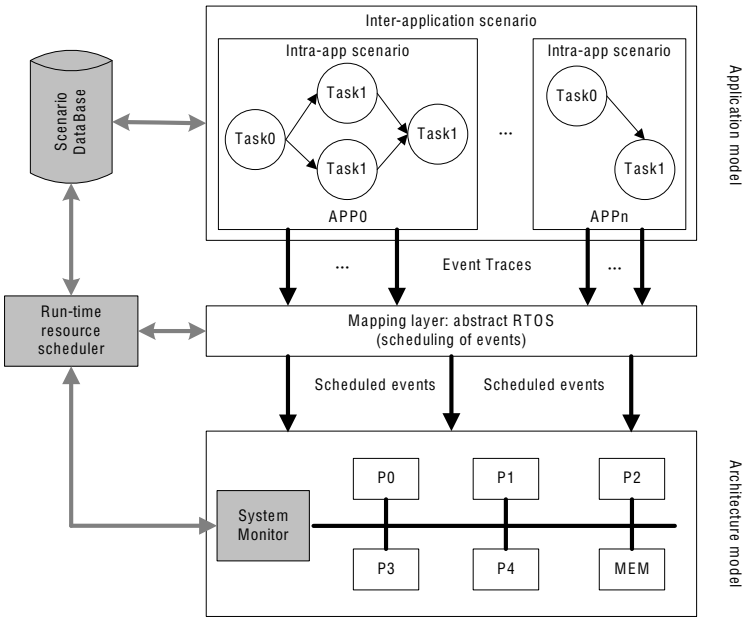


Figure 3.1: Extended sesame framework.

However, before starting a simulation in the Sesame environment, the task mapping information should be explicitly provided to the mapping layer based on the pre-defined target application and architecture model. And it can not be changed dynamically during simulation. Therefore, the original Sesame is unable to support the simulation of dynamic application behaviour and run-time mapping optimisation/customisation. To enable this property, we have extended the basic Sesame simulator with a run-time resource scheduling framework as illustrated in Figure 3.1.

Our extensions to Sesame include a Scenario DataBase (SDB), a Run-time System Monitor (RSM) and a Run-time Resource Scheduler (RRS). The SDB is used to store the information that will be used for run-time resource management on the target MPSoC system. The content of this SDB depends on the mechanism for dynamic resource management. For example, if a general hybrid task mapping approach is applied on the system, where the system resources are dynamically allocated for each workload scenario according to statically optimised mappings explored at design time, then the pre-optimised mapping for each workload scenario will be stored in the SDB. Besides the pre-optimised mappings, some other information such as the application and architecture details that are related to dynamic mapping re-optimisation, as well as the application-specific information like the performance objective and energy budget for dynamic QoS management could be stored in the SDB for better run-time system resource management.

The RSM is a modularised component that can be integrated into the target architecture for the purpose of detecting and identifying the active workload scenario, and also for collecting the statistics (e.g., performance of each application, system execution information, etc.) from the underlying system during the execution of a certain workload scenario. As introduced in Section 2.1.1, the application behaviour is captured by event traces in Sesame. Benefiting from the *STARTSCENARIO* and the *ENDSCENARIO* events as introduced in Section 2.1.1 that annotate the beginning and the end of each application respectively, we are

Table 3.1: Standard interfaces in the RRS

Interface	**Parameters**	**Function**
getMapping	*m_id:* mapping identification	load pre-optimised mappings from the SDB
getAppInfo	*a_id:* application identification, *i_ty:* information type	load application-specific information from the SDB
getArchInfo	*i_ty:* information type	load architecture information from the SDB
getStatistics	*i_ty:* information type	get system statistics from the RSM
mappingOpt	*tg:* trigger of system reconfiguration	generate new task mapping according to the system reconfiguration trigger type detected in the RSM and the strategy implemented in this function
reMap	*n_m:* new derived mapping	generate new mapping scheme for the mapping layer in Sesame

able to distinguish different workload scenarios. The mechanism of detecting and identifying a workload scenario works as follows. When a processing element in a Seseme system model encounters a *STARTSCENARIO* event of an application, it will register this application with its execution mode information in the RSM to notify that a new application has started execution on the system. Similarly, for an *ENDSCENARIO* event of an application, the processing element will unregister the application in the RSM. According to the registered application information in the RSM, the active workload scenario on the target system can be identified. Note that, for the application-specific statistics like the execution time of an application, it can be derived by adding the corresponding information such as the start and the end time of the application into the above mentioned processes. By monitoring the system run-time execution behaviour, the RSM is able to generate different system reconfiguration triggers, like the change of workload scenarios and the violation of certain application objectives, for the RRS to further optimise the resource allocation on the system.

The RRS is the main component for run-time resource management on a target system. It is in charge of deriving new resource allocation schemes (task mapping in our case) based on the reconfiguration triggers generated by the RSM and the mapping optimisation policy. We provide a standard interface for implementing different task mapping strategies in the RRS component. Consequently, it enable us to find a good task mapping strategy for the target applications on the target architecture. After a new mapping scheme is derived by the implemented mapping optimisation strategy, the RRS will remap the application tasks onto the target architecture and then continue the system execution under the new mapping scheme. Different with the RSM which is integrated into the architecture model in Sesame, the RRS works on top of the architecture model like a plug-in

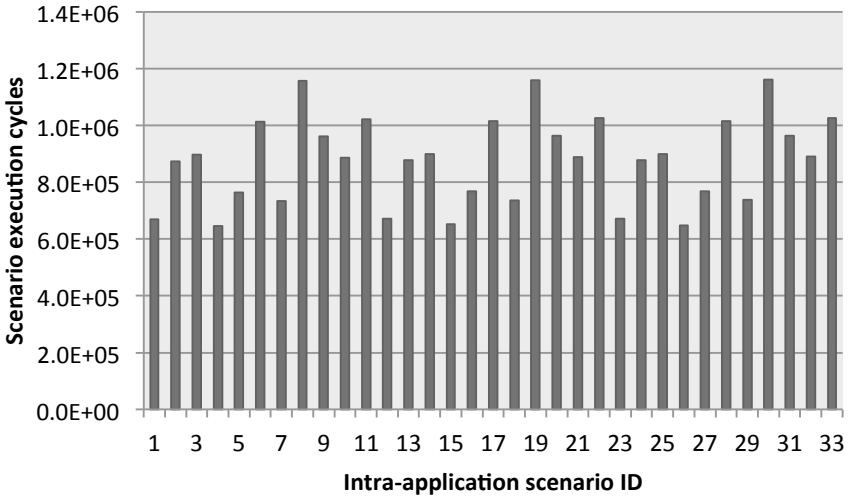


Figure 3.2: Intra-application scenario performance of an MJPEG application.

component to Sesame by standard interfaces. Table 3.1 illustrates some important interfaces for the RRS.

After extending Sesame with the above mentioned run-time resource scheduling framework, we are able to simulate an MPSoC system with dynamic application behaviour and investigate different run-time management mechanisms. In the following sections, several task mapping techniques for dynamic resource management are subsequently proposed to improve the system adaptivity of different MPSoC systems with complex and dynamic application behaviour.

3.2 Task Mapping with Scenario Clustering

Under general hybrid task mapping techniques, at design time of an embedded system, a designer could aim at finding the optimal mapping of application tasks to MPSoC processing resources for each workload scenario to maximally improve the optimising objectives on the target system. However, when the number of applications and application modes increase, the total number of workload scenarios will explode exponentially. Consequently, the time needed for exploring the mappings at design time will be intractable. Moreover, storing all these optimal mappings such that they can be used at run time by the system to remap tasks when a new scenario is detected would also be unrealistic as this would take up too much memory storage.

An approach to solve this problem is by clustering workload scenarios and only storing a single mapping per cluster of workload scenarios to facilitate run-time mapping [43]. Such clustering implies a significant time and space reduction needed to explore and store the mappings. In the work of this section, we try to use a scenario clustering based approach to improve the efficiency of a homogeneous system with dynamic multimedia application behaviour.

3.2.1 Motivation Example

As defined in Section 1.4, workload scenarios are the combination of inter-application scenarios and intra-application scenarios of the target applications. In this section, we consider a clustering method[1] in which we find and store a single mapping for each inter-application scenario that yields, on average, the best performance for all possible intra-application scenarios within the inter-application scenario. However, as we can see from Figure 3.2, using such a single mapping to represent an entire inter-application scenario shows considerable performance variations for the different intra-application scenarios that exist in this inter-application scenario. In this particular example, the inter-application scenario contains three simultaneously running multimedia applications: a Motion-JPEG (MJPEG) encoder, a MP3 decoder, and a Sobel filter for edge detection in images. The use of cluster-level mappings (i.e., mappings found to be good for an entire cluster of workload scenarios) can provide a run-time mapping system with enough information to quickly find an adequate mapping for a detected workload scenario but it will not immediately lead to finding the optimal system mapping for any identified workload scenario. Therefore, we propose a novel run-time Scenario-based Task Mapping algorithm (STM) that uses the cluster-level mapping information derived from design-time design space exploration (DSE) but, additionally, performs run-time mapping optimization by continuously monitoring the system and trying to perform (relatively small) mapping customisations to gradually further improve the system performance.

3.2.2 Problem Definition

In the case of a multi-application workload, the possible workload scenarios can be divided into inter- and intra-application scenarios. Let $A = \{app_0, app_1, ..., app_m\}$ be the set of all applications that can run on the system, and $M^i = \{md^i_0, md^i_1, ..., md^i_n\}$ be the set of possible execution modes for $app_i \in A$. Then, $SE = \{se_0, se_1, ..., se_{n_{inter}}\}$, with $se_i = \{app_0 = 0/1, ..., app_m = 0/1\}$ and $app_i \in A$, is the set of all inter-application scenarios. And $sa^i_j = \{app_0 = md^0_{j_0}, ..., app_m = md^m_{j_m}\}$, with $app_i \in A \wedge app_i = 1 \in se_i$ and $md^i_{j_x} \in M^i$, represents the j-th intra-application scenario in inter-application scenario $se_i \in SE$. The set of all workload scenarios can then be defined as the disjoint union $S = \sqcup_{i \in SE} SA^i$, with $SA^i = \{sa^i_1, sa^i_2, ..., sa^i_{n^i_{intra}}\}$.

As already explained in Section 3.2.1, we propose to perform the run-time mapping of applications in two stages. In the first stage, which is performed at design time, we cluster workload scenarios (similar to [43]) and perform DSE for each of these scenario clusters to find a mapping that shows the best average performance for that particular cluster. More specifically, in this section, we consider each $se_i \in SE$ as a different cluster of scenarios (i.e., we cluster all intra-application scenarios of an inter-application scenario). The mappings derived from design-time DSE are stored so they can be used by the run-time mapping algorithm to re-map applications when a workload scenario is detected that belongs to a different scenario cluster. Since these statically determined mappings may

[1] We note, however, that other clustering methods would also be possible and that our run-time mapping algorithm is independent on the clustering method used.

not be optimal for the current active intra-application scenario, the second stage of the run-time mapping algorithm tries to perform (relatively small) mapping customizations to gradually further improve the system performance. In our goal to optimize mappings, we recognize two kinds of objectives: system-level objectives and application-dependent objectives. System-level objectives, denoted as $O_\alpha = \{O_{\alpha 0}, O_{\alpha 1}, ...\}$, define the system-wide metrics such as system energy consumption, total system execution time, etc. Application-dependent objectives, denoted as $O_\beta = \{O_{\beta 0}, O_{\beta 1}, ...\}$, are mainly used to define the performance requirements of each separate application like throughput, latency, etc. As will be explained in the next section, the first stage of our run-time mapping approach uses system-level objectives to find mappings per scenario cluster. Here, we use system energy consumption and total workload scenario execution time as metrics: $E_{s_i}, s_i \in S$ represents the system energy consumption of workload scenario s_i and $X_{s_i}, s_i \in S$ is the execution time of scenario s_i. For the second stage, during which the mapping is gradually optimized, we apply application-specific objectives – in our case throughput requirements for each application – for the optimization process. However, to measure the results of the run-time optimization process, we also use the system-level metrics E_{s_i} and X_{s_i}.

Under the above definition and given the $KPN = (P, F)$ for each application and an $MPSoC = (PE, M)$ as described in Section 2.2, our goal is to continuously customize the mapping at run time such that the system-level (system performance and energy consumption) and/or application-specific (application performance) objectives under every workload scenario $s_i \in S$ are satisfied. In this section, our target architecture is a homogeneous MPSoC system.

3.2.3 Static Multi-objective Task Mapping Optimisation

Different with the DSE approach proposed in Section 2.3 where only single optimising objective is considered in the mapping exploration process, the static mapping optimisation problem of this chapter considers multiple optimising objectives such as performance and energy consumption. To this end, we have deployed the scenario-based DSE approach presented in [128], which is based on the well-known NSGA-II genetic algorithm and allows for effectively pruning the design space by only evaluating a representative subset of the target problem. As our target MPSoC platform is known, we can use a simplified version of the approach in [128]. The implementation of the NSGA-II genetic algorithm for our target mapping problem is explained below. With regard to the chromosome representation of the mapping problem, it uses the encoding approach introduced in Section 2.3.2.1.

3.2.3.1 Fitness Function

To find the mapping solutions optimised for both performance and energy consumption for a single workload scenario s_i in the mapping space in question, the Pareto Front of mapping performance and mapping energy consumption is generated by solving the following multi-objective optimization problem according to the problem definition of Section 3.2.2:

$$min[E_{s_i}, X_{s_i}]. \tag{3.1}$$

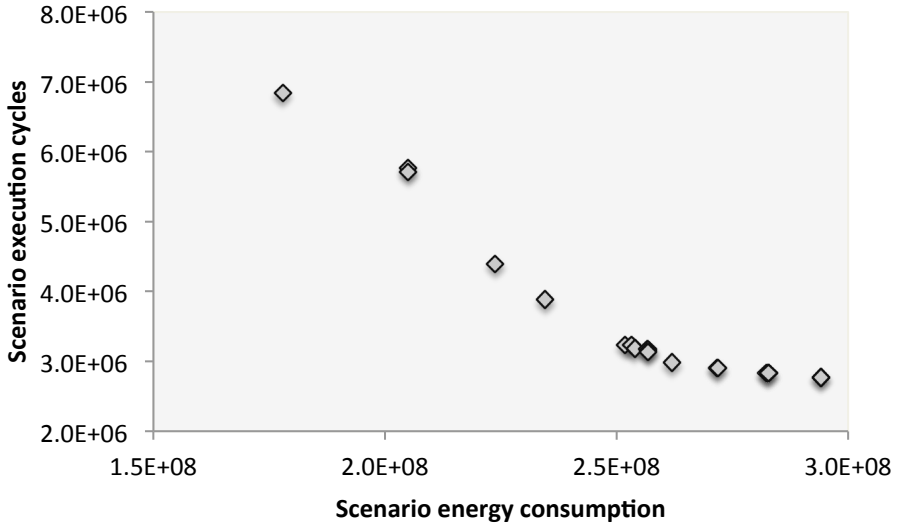


Figure 3.3: Pareto Front of a workload scenario

Evaluating the fitness value of each individual (i.e., design point) is performed using our Sesame simulator similar to our BEG algorithm in the previous chapter (see Section 2.3.2). After the GA-based DSE, a Pareto Front of solutions considering both the mapping performance and energy consumption is generated, as illustrated in Figure 3.3. Looking at the Pareto Front, one can easily obtain the mappings satisfying different objectives like mapping solutions with maximal performance, minimal energy consumption and maximal performance under a certain energy budget which will be introduced later in this chapter.

3.2.3.2 Operators for NSGA-II

To effectively search for global optimal mapping solutions, and escape possible local ones, the crossover and mutation operators are important components of a GA. With respect to the crossover operator, the most common methods are one-point crossover, two-point crossover and uniform crossover. For our multi-objective task mapping optimisation problem, the one-point crossover is used as it is simple and yields more or less the same effect as the other approaches. Regarding mutation, we use an operator that randomly selects an application task that is subsequently moved to a randomly selected processor.

3.2.3.3 Parameters for NSGA-II

Another important step is to set appropriate parameters (problem dependent) for the GA, such as population size, crossover and mutation probabilities, etc. The parameters of the NSGA-II genetic algorithm we have used for design-time DSE are listed in Table 3.2, which have been tuned for obtaining high-quality mappings for each workload scenario in our benchmark set considered in this chapter.

Table 3.2: Parameters of NSGA-II

parameter	value
initial population size	256
generation size	256
generations	512
crossover probability	0.8
chromosome-level mutation probability	0.2
gene-level mutation probability	0.05

3.2.4 Scenario Clustering Based Task Mapping

Based on the scenario clustering approach, our run-time task mapping algorithm (STM) uses the cluster-level mapping information and further optimises the mapping to satisfy the application-dependent optimising objectives.

The STM algorithm, which is outlined in Algorithm 3, can be divided into a static part and a dynamic part. The static part is used to capture application dynamism at the granularity of inter-application scenarios. For each inter-application scenario $se_i \in SE$, we have determined – using above introduced design-time DSE – a mapping that on average performs best for all intra-application scenarios SA^i of se_i. That is, for each se_i we search for a mapping by solving the following multi-objective optimisation problem:

$$min[\sum_{sa_j^i \in SA^i} E_{sa_j^i}, \sum_{sa_j^i \in SA^i} X_{sa_j^i}]. \tag{3.2}$$

The mappings derived from this design-time DSE are used by the STM algorithm as shown in lines 1-3 of Algorithm 3. When the system detects the execution of a different inter-application scenario, the static part of the STM algorithm will choose the corresponding mapping as derived from the design-time DSE stage and which has been stored in the previously mentioned *Scenario DataBase*. Because this database stores mappings for entire scenario clusters, its size can be controlled by choosing a proper granularity of scenario clusters (e.g., inter-application scenarios).

The dynamic part of our STM algorithm is active during the entire duration of an inter-application scenario. As explained in the previous section, it uses application-specific objectives, specified for each separate application, to continuously optimize the mapping. When the algorithm detects that an objective is unsatisfied, it will try to find a new task mapping for that particular application that missed the performance goal. If multiple applications miss their performance goal, then the STM algorithm will start optimizing the most problematic application first. The main steps of the dynamic part of the STM algorithm are described below.

3.2.4.1 Finding the Critical Task

The first step of the dynamic part of the STM algorithm is to find the so-called *critical task* for the application that missed its objective, as shown in lines 10-

Algorithm 3 STM algorithm

```
Input: KPN_{app_0,...,app_m}, MPSoC, O_α, O_β, μ, η
Output: New(μ, η)
list: TC, CIC, CIB, PU
pCIC = δ_c, pCIB = δ_b
1: if detectScenario() == true : //new inter-application scenario
2:   New(μ, η) = getMapping();
3:   return New(μ, η);
4: else :
5:   results[] = getStatistics();
6:   if objectiveUnsatisfied(results, O_α, O_β) != -1:
7:     taskCost(KPN_{app_i}, results, TC, CIC, CIB);
8:     peUsage(results, PU);
9:     while(1) :
10:      if (apptype = getType(KPN_{app_i})) == DATA_PARALLEL :
11:        critical = findDPCritical(KPN_{app_i}, CIC, CIB, pCIB, pCIC);
12:      else :
13:        critical = findCritical(KPN_{app_i}, CIC, CIB, pCIB, pCIC);
14:      reason = findReason(critical, CIC, CIB, pCIB, pCIC);
15:      if reason == POOR_LOCALITY :
16:        MCC[] = minCircle(KPN_{app_i}, results, critical);
17:        if GetSubstitute(PU, μ, η, MCC, apptype) == true :
18:          return New(μ, η);
19:        else failed;
20:      else if reason == LOAD_IMBALANCE :
21:        if GetSubstitute(PU, μ, η, apptype) == true :
22:          return New(μ, η);
23:        else failed;
24:      else :
25:        pCIB += ε;
26:        pCIC -= ε;
```

13 of Algorithm 3. The rationale behind this is that by remapping this critical task and possibly its neighbouring tasks (forming a bottleneck in the application), the resulting effect will be optimal. To find the critical task, the STM algorithm maintains three lists for the current task mapping. The first list stores the task costs (TC). For every application, it contains the cost of the application's tasks, where the cost is determined by the sum of the execution and communication times of a task. These task costs are arranged in descending order in the list. The two other lists concern the storing of two other metrics for each task: the proportion of task cost in the total busy time of the PE (i.e., processor) onto which the task is currently mapped (CIB), and the proportion of task communication time (read and write transactions) in the task cost (CIC).

Using the TC list, the algorithm checks the task at the top of the list to find the critical task, taking the following two conditions into account: 1) whether

or not the task's CIB proportion is lower than a specific threshold, defined by *pCIB*. Here, the rationale is that a high-cost task receiving only a small fraction of processor time may imply that the processor is overloaded. If the task satisfies this condition, then this task is considered as the critical task and the process of finding the critical task ends. Otherwise, the algorithm continues to check the other tasks in the TC list with lower costs until it finds the critical task. If there is no task in the application that satisfies the first condition, then the second condition will be used: 2) Whether or not the CIC proportion is higher than the threshold *pCIC*. The algorithm checks all the tasks using this second condition just like it did for the first condition. If all the tasks do not satisfy these two conditions, then the algorithm will, respectively, increase and decrease the pCIB and pCIC thresholds by ε, after which the above process is restarted again.

For data parallel applications[2], the process of finding the critical task has one additional test as compared to regular applications. This extra test (performed in the function *findDPCritical*) involves the check whether or not all data-parallel tasks are mapped onto different PEs. If there are data-parallel tasks that are mapped onto the same processor, then those tasks with higher task costs will be treated as critical tasks. Otherwise, the process of finding the critical task will be the same as for regular applications.

3.2.4.2 Remapping the Critical Task

After the critical task has been found, the STM algorithm tries to analyze the reason for missing the application's performance goal. In this respect, we recognize two different reasons: *poor locality* and *load imbalance*. Here, we use the process of determining the critical task to also determine the reason for not meeting the performance goal: If the CIC proportion of the critical task is higher than the value of the current pCIC threshold, then the algorithm assumes that poor locality is the reason. Otherwise it takes load imbalance as the reason for not meeting the application demands. This means that poor locality has a higher priority than load imbalance as a reason for not meeting the application demands, which is helpful to reduce the energy consumption due to communications.

Subsequently, the function *GetSubstitute* in the STM algorithm can follow different strategies to find a target PE to which the critical task will be remapped. The selection of remapping strategy depends on the reason for not meeting the application's performance demands as well as on the type of application (data parallel or not). The strategies that are used to find the substitute PE for data-parallel applications are similar to the ones for regular applications except that one additional condition is taken into account for finding the substitute PE: the substitute PE should not be a PE onto which its parallel tasks are mapped.

Poor Locality In the case of poor locality, the STM algorithm will try to find a better mapping for the application in question based on a *minimal cost circle (MCC)* approach. A situation that has been identified as "poor locality" is mainly due to the communication overhead between tasks. Evidently, if the communicating frequency between two tasks is very high or the communicating data size is

[2]Data parallel application: an application contains duplicate tasks that have the same functional behaviour but process different input data.

very large, then these two tasks should preferably be mapped onto the same PE or onto two different PEs that contain a more efficient interconnect between each other. The MCC strategy aims at redistributing the critical task and its neighbouring tasks over PEs such that communication overhead is reduced while trying to avoid creating new computational bottlenecks. To this end, it first finds the minimal cost circle based on Equation 3.3 (to find the index m, n) for the critical task $p_i \in P$ of the target application $KPN = (P, F)$:

$$min(Circle\_Cost(p_i)_{mn}),\ with\ 0 \leq n, m \leq |P|, m \leq i \leq n \tag{3.3}$$

where:

$$Circle\_Cost(p_i)_{mn} = \sum_{m \leq k \leq n} et_k + \sum_{m \leq k \leq n} \sum_{0 \leq j < |P|} ec_{kj} \tag{3.4}$$

where et_k denotes the execution time of task k (the neighbours of task i) for the PE onto which task k is currently mapped, and ec_{kj} denotes the communication overhead between tasks k and j under the current mapping. $|P|$ represent the total number of tasks in the target application.

This strategy is applicable for heterogeneous MPSoC architectures which will be introduced in the later section of this chapter. However, in this section, our focus is on homogeneous architectures using a shared bus interconnect. This means that each task will have a constant computational cost irrespective of the PE it is mapped on, and that communication overhead only involves internal communication within a single PE (i.e., when the communicating tasks are mapped to the same PE) or external communication between PEs via shared memory. Clearly, internal communication costs are much lower than external communication costs. Figure 3.4.a shows an example of an MCC (indicated by the dotted oval) that contains two tasks, including the critical task (grey task), whereas Figure 3.4.b illustrates an MCC that only contains the critical task itself.

After the MCC of the critical task has been determined, the function *Get-Substitute* will choose a substitute PE for all the tasks included in the identified MCC to achieve a new mapping. For this purpose, the PU list as shown in line 8 of Algorithm 3 is used, containing the processor utilisations for each PE. The substitute PE is the PE with the lowest utilization in the PU list that is different from the PE onto which the critical task is currently mapped. If the MCC solely consists of the critical task itself, then the critical task will be mapped onto the PE of a neighboring task that has the heaviest communication with the critical task. This is, e.g., shown in Figure 3.4.b, where the critical task will be mapped onto the same PE as the task with cost 70. Moreover, the substitute PE should be different than the PE the critical task is currently mapped on. Otherwise, the algorithm fails to find a new mapping. After the substitute PE has been found, the FIFO channels between the tasks that need to be remapped are either mapped as internal communication onto the new PE (if communicating tasks are mapped onto this PE) or onto the system bus.

Load Imbalance In the case a load imbalance has been identified as the reason for not meeting the application demands, a load balancing strategy is used to remap the critical task. The substitute PE should satisfy the condition that it is different from the current PE of the critical task and should have the lowest

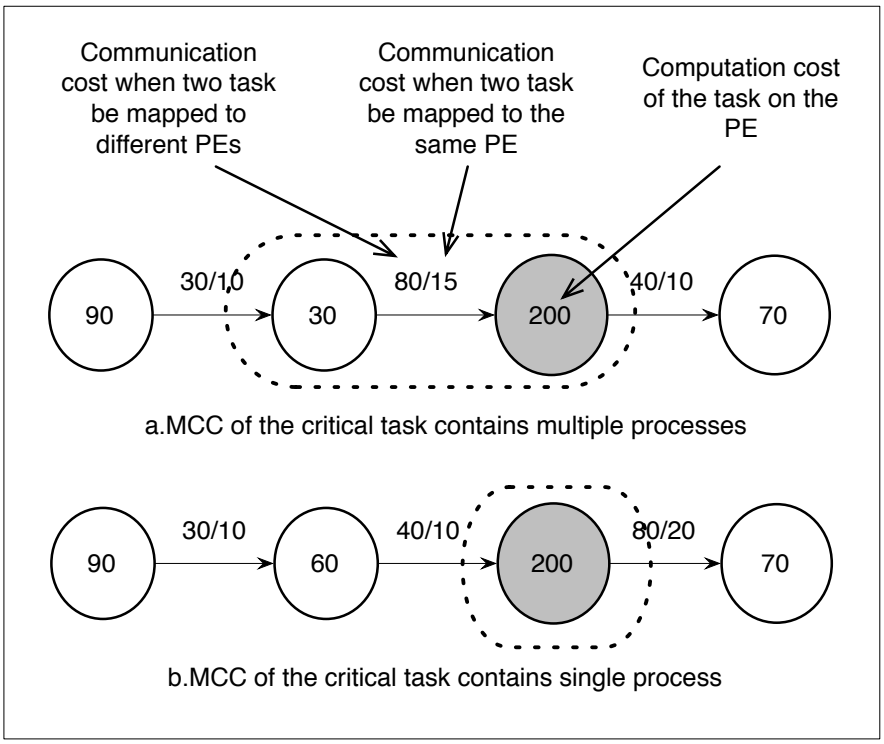


Figure 3.4: Examples of an MCC for a critical task (grey task).

processor utilization in the PU list. If such a substitute does not exist, then the algorithm cannot find a better mapping.

3.2.5 Experiments

3.2.5.1 Experimental Framework

To evaluate the efficiency of our STM algorithm and the mappings found at run time by this algorithm, we deploy the extended Sesame simulator as introduced in Section 3.1. In this extended simulator, the SDB is used to store the mappings for each inter-application scenario as derived from design-time DSE and also the application specific information such as the performance objective, the computation and communication cost of tasks on different hardware components. The RSM is in charge of recording the running statistics for each active application as well as monitoring system-wide statistics. The RRS uses the run-time task mapping algorithm and the statistics provided by the RSM to dynamically remap application tasks when needed.

3.2.5.2 Experimental Results

In this subsection, we present several experimental results in which we investigate various aspects of our STM algorithm and compare it to three well-known mapping algorithms: First-Fit Bin-Packing (FFBP) [31] which has been frequently adapted to do task mapping by means of modeling it as a bin-packing problem, Output-Rate Balancing (ORB) [22] and Recursive BiPartition and Refining (RBPR) [139].

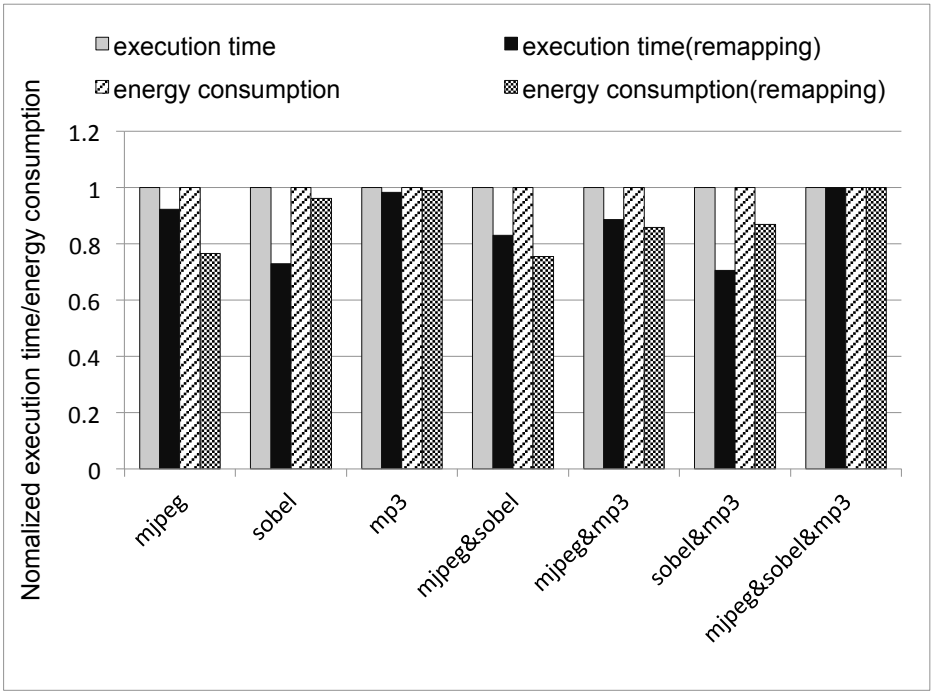


Figure 3.5: Performance and energy consumption of each inter-application scenario.

We modified these algorithms to fit our mapping problem and extended them to also allow for mapping data-parallel applications by constraining the data-parallel tasks so that they have to be mapped onto different processing elements. For the FFBP algorithm, the PE with the lowest utilization is taken as the first-fit bin and the computational cost of each task in the target application is considered as the object that needs to be packed into the bins.

For our experiments, we use the three typical multi-media applications that were already introduced in Section 3.2.1: MJPEG, Sobel and MP3. The KPN of the MJPEG application contains 8 processes and 18 FIFO channels, Sobel contains 6 processes and 6 FIFO channels, and MP3 contains 27 processes and 52 FIFO channels. In the Sobel and MP3 applications, data parallelism is exploited. Moreover, MJPEG has 11 intra-application scenarios, MP3 has 3 intra-application scenarios, whereas Sobel only has 1 intra-application scenario. This results in a total of 95 different workload scenarios. At design time, we have determined the on-average best mapping for each possible inter-application scenario as explained in Section 3.2.4. It means that only 8 mappings in total need to be explored and stored in the system memory. With respect to the target architecture, we modeled a homogeneous MPSoC containing 5 processors, connected to a shared bus and memory. The model also includes the required components for our run-time scheduling framework.

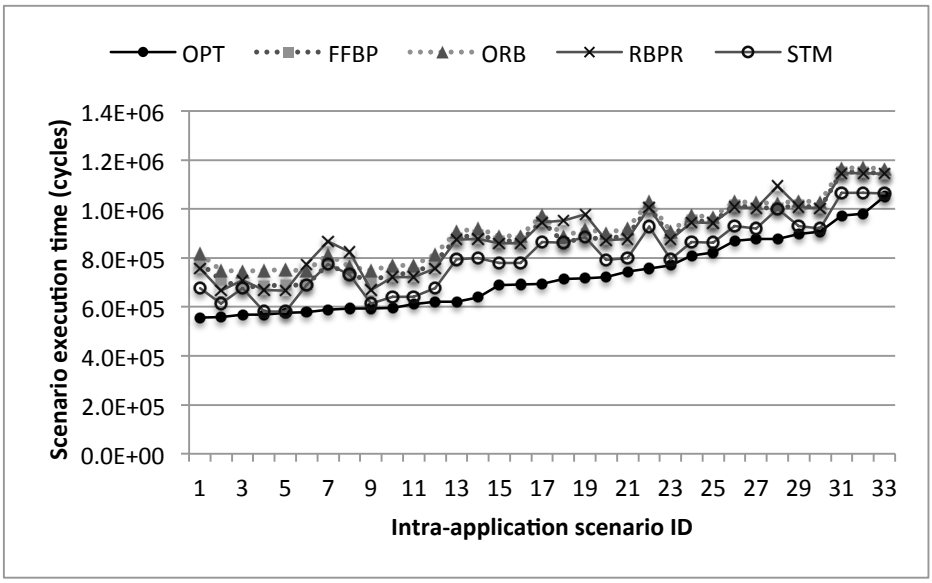
As there are just three applications and each application contains a limited number of intra-application scenarios, we are able to exhaustively evaluate all workload scenarios. For each workload scenario, we have simulated the system using two methods: one is deploying *only the static part* of our STM algorithm to deal with the dynamism at the level of different inter-application scenarios, whereas the other one is running all the workload scenarios under a single, fixed

mapping: the on-average best mapping found for the inter-application scenario in which all three applications are concurrently executing. The results of this experiment are shown in Figure 3.5. From this figure, we can see that the static part of our STM algorithm already yields both performance improvements and energy savings by dynamically adjusting the mapping based on the variation in inter-application scenarios. For this specific test case, the performance improvements for the different inter-application scenarios range from 1.69% to 29.49% and the energy savings range from 1.09% to 24.51%. Overall, for the execution of all 95 workload scenarios, the improvements in terms of performance and energy saving are 7.4% and 9.4%, respectively.

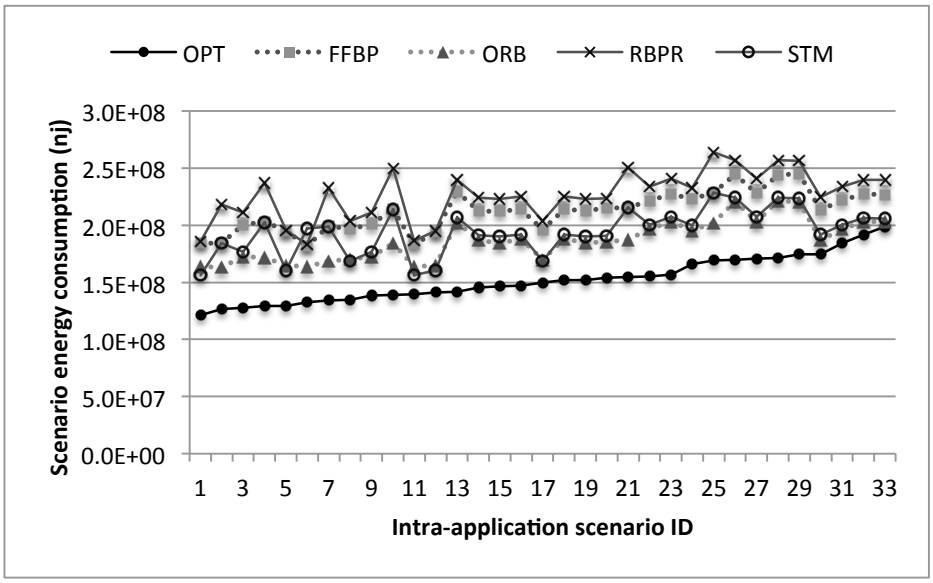
Figures 3.6a and 3.6b show the intra-application scenario execution times and energy consumption for the FFBP, ORB, RBPR and STM run-time mapping algorithms for a single inter-application scenario in which all three applications are concurrently executing. Moreover, these two graphs also contain the results when using optimal mappings (OPT) for each intra-application scenario (we derived these mappings in a design-time DSE experiment). The results in these two graphs have been ordered in a monotonically increasing fashion based on the results from the OPT mappings. Figure 3.7 shows the overall (for the entire inter-application scenario) performance, energy consumption and overhead. Here, the overhead includes the run-time calculation of new mappings when a mapping optimisation is triggered by the violation of performance objectives of the target three applications as well as the migration of tasks[3] in the case of the optimised mapping is different with the previous mapping. In this experiment, the target system is initialised under the pre-optimised mapping of the target inter-application scenario. We have enforced the system to enter a mapping optimisation process for each intra-application scenario in the target inter-application scenario by adjusting the performance objectives of the target applications. From Figure 3.6 and Figure 3.7, we can see that our STM clearly performs better than the other algorithms in terms of the execution time of scenarios. For several intra-application scenarios, the STM algorithm even approaches the OPT results. With respect to energy consumption and overhead, the STM algorithm also performs well: it ranks second closely behind the ORB algorithm. The reason for a low overhead of ORB is that it only needs to migrate a few tasks in our experiment which means a very low task migration cost.

In our last experiment, we used the full STM algorithm, including the static and dynamic parts and thus combining the dynamism of inter-application as well as intra-application scenarios, to test all the 95 workload scenarios of our three applications. Our algorithm could achieve a 11.3% performance improvement and an energy saving of 13.9% compared to an approach in which we run the applications using the (static) on-average best mapping for the inter-application scenario in which all three applications are active. Comparing these results to those when only using the static part of our STM algorithm (improvements of 7.4% and 9.4%, respectively; see above), this means that the dynamic part of the STM algorithm is capable of significantly further improving the mappings.

[3]Here, the task migration overhead is estimated by a simple analytic model that takes the task size and memory access speed into account. More accurate task migration overhead during task remapping will be studied in the next chapter.



(a) Simulated scenario execution time under different intra-application scenarios

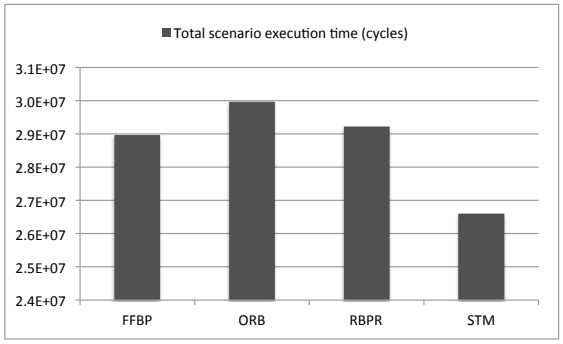


(b) Simulated scenario energy consumption under different intra-application scenarios

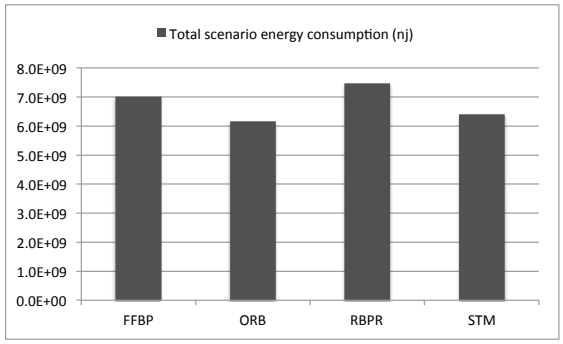
Figure 3.6: Comparing different run-time mapping algorithms under different intra-application scenarios.

3.2.6 Conclusion

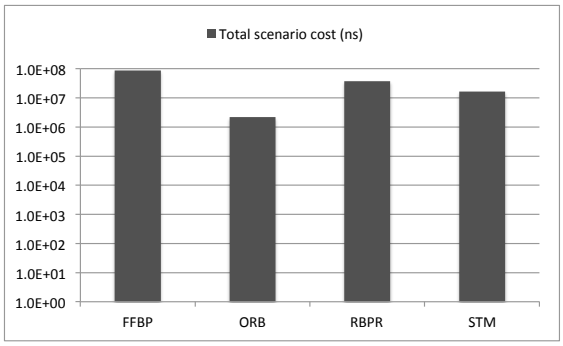
In this section, we have proposed a run-time mapping algorithm for MPSoC-based embedded systems to improve their performance and energy consumption by capturing the dynamism of the application workloads executing on the system. This algorithm is based on the idea of application scenarios and consists of a design-time and run-time phase. The design-time phase produces mappings for clusters of application scenarios after which the run-time phase aims to optimize these mappings by continuously monitoring the system and trying to perform (relatively small) mapping customizations to gradually further improve the system performance. In



(a) Total scenario execution time of all intra-application scenarios in the target inter-application scenario



(b) Total scenario energy consumption of all intra-application scenarios in the target inter-application scenario



(c) Total scenario cost of all intra-application scenarios in the target inter-application scenario

Figure 3.7: Comparing different run-time mapping algorithms under an entire inter-application scenario.

various experiments, we have evaluated our algorithm and compared it with three other algorithms. The results show that our algorithm can yield considerable improvements as compared to just using a static mapping strategy. Comparing our algorithm with three other, well-known run-time mapping algorithms, it shows a better trade-off between the quality and the cost of the mappings found at run time.

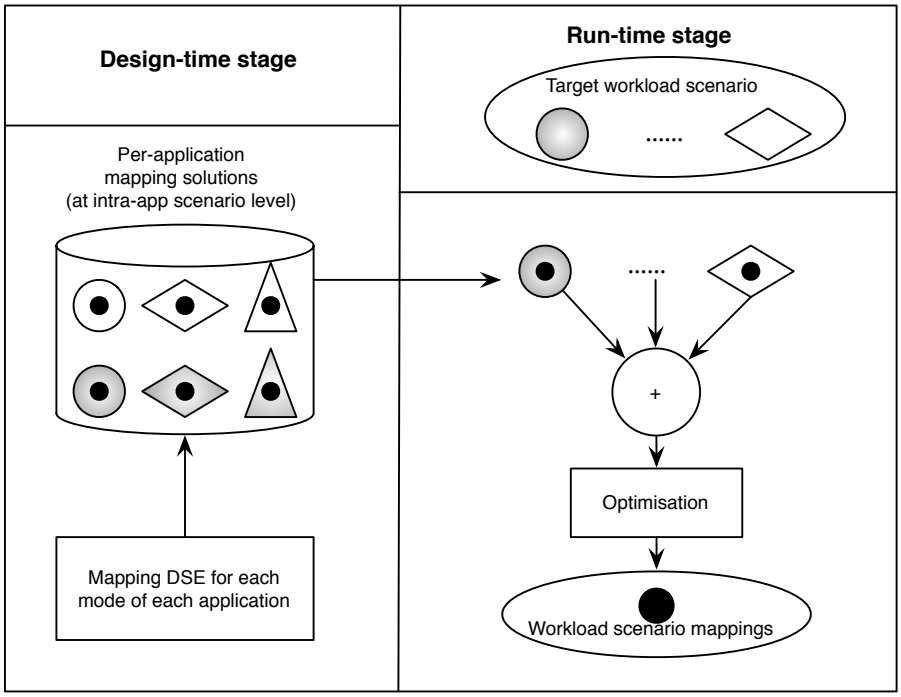


Figure 3.8: The workflow of the proposed hybrid task mapping approach.

3.3 A Novel Hybrid Task Mapping Approach

As discussed at the start of this chapter, besides the scalability with regard to the number workload scenarios, the flexibility of supporting new applications is another major issue in general hybrid task mapping approaches. The method in the previous section solves the scalability problem but does not solve the flexibility problem. Therefore, this section presents an approach that tries to address both issues.

For this purpose, we propose an Energy-aware Iterative multi-application Mapping (EIM) algorithm that operates at run time. Based on design-time statically derived optimal (or near optimal) mappings for each separate application, this algorithm will quickly find a near optimal mapping under the objectives of high performance and low energy consumption for the simultaneously running applications on heterogeneous platforms. By using our proposed approach, the scenario-level task mapping problem is solved by a divide-and-conquer technique as illustrated in Figure 3.8 where the complex task mapping problem is broken down into small task mapping problems at design time, and the small mapping solutions are then dynamically combined and further optimised to give a complete solution for the complex mapping problem at run time. It is achieved by splitting the handling of intra-application scenarios and inter-application scenarios according to the two stages mentioned above. The design-time phase takes charge of exploring three optimal mappings for each intra-application scenario of each application with respect to three different objectives: maximising the throughput, minimising the energy and maximising the throughput under a predefined energy budget. The run-time phase subsequently finds a mapping for the workload scenario that has emerged in the system by considering the active inter-application scenario at run

time. By using this method, the number of mappings that need to be determined at design time will be greatly reduced. Considering the previously mentioned example (see Section 1.4) of 10 applications with 5 execution modes for each application (over 60 million workload scenarios in total), only 150 mappings need to be found (and stored) at design time for the approach presented in the section for the optimisation objective of both performance and energy consumption[4], which is even smaller compared with using the approach proposed in the previous section where at least 1023 mappings should be explored and stored (one mapping for each inter-application scenario).

3.3.1 Problem Definition

As mentioned above, in our novel hybrid task mapping approach, we propose to perform the task mapping of applications in two stages. In the first stage, which is performed at design time, we perform DSE for each intra-application scenario of each application (denoted by scenario s_i in the whole workload scenario space S) to find three mappings that show the maximal throughput, minimal energy consumption and maximal throughput under a certain energy budget b_i respectively. Here, b_i is a user defined energy budget for workload scenario s_i. The mappings derived from design-time DSE are stored so they can be used by the second stage to get a final mapping – by directly using the stored mappings or by deriving a new one from the stored mappings – for the current system objective when a new workload scenario is detected. Here, we can distinguish two system objectives: maximal throughput and minimal energy consumption for each workload scenario, denoted as O_t and O_e respectively. These two objectives will be used for run-time mapping optimization. For the convenience of exploring the pareto front of objectives at design time, we change the objective of maximal throughput into a minimal objective $O_p = 1/O_t$, namely the scenario execution time. With regard to the run-time behaviour, we assume that our hardware platform can run in *two modes*: an energy-aware high performance mode (using a certain energy budget) and an energy saving mode. Consequently, the run-time system objectives for each workload scenario are O_{pb}, which means minimal scenario execution time (or maximal throughput) under the given energy budget, and O_e. Users can choose the running mode of the system, or the system itself can adaptively adjust the mode based on e.g. the battery usage.

Under these definitions and the definition of application model and architecture model provided in Section 2.2, our goal of this section is to find the optimal or near optimal mapping at run time for each detected workload scenario $s_i \in S$ with the objective to minimize O_{pb} or O_e (according to the system execution mode) on the target heterogeneous platform $MPSoC = (PE, M)$.

3.3.2 Iterative Multi-Application Mapping Optimisation

3.3.2.1 Design-time Mapping Exploration

Similar to the static mapping exploration problem discussed in Section 3.2, the design time mapping exploration of this section also considers multiple optimisa-

[4]If just a single optimisation objective is considered, only 50 mappings need to be explored at design time and stored on the target system

Algorithm 4 EIM algorithm

Input: $KPN_{app_{active}}$, $MPSoC$, scenario_id(s_i), sys_mode
Output: (μ, η)
1: (μ, η) = getInitMapping(s_i, sys_mode);
2: if singleAppActive(s_i) == true:
3: return (μ, η);
4: else:
5: switch(sys_mode):
6: case EA-HIGHPERF:
7: U = peUsage($KPN_{app_{active}}$, $MPSoC$, μ, η);
8: M_p = maxPUsage(U);
9: V_p = varPUsage(U);
10: b_i = eBudget(s_i);
11: return iterativePOpt(μ, η, M_p, V_p, b_i);
12: case ENERGYSAVING:
13: econs = energyCons($KPN_{app_{active}}$, $MPSoC$, μ, η);
14: return iterativeEOpt(μ, η, econs);
15: default:
16: return (μ, η);

tion objectives (performance and energy consumption). Consequently, the same scenario-based DSE approach based on the NSGA-II genetic algorithm as introduced in the previous section has been deployed to explore task mappings at design time for each intra-application scenario (execution mode) of each application. Notice that, the difference between the static mapping exploration of this section and the one in Section 3.2 exists in the mapping problem level where this section explores task mappings at application level but Section 3.2 explores task mappings at scenario cluster level.

By using the scenario-based DSE approach, task mappings with minimal O_p, minimal O_{pb} and minimal O_e for each intra-application scenario of each application are generated (derived from the Pareto Front of mapping solutions like Figure 3.3) and stored on the target system for run time mapping optimisation.

3.3.2.2 Run-time Task Remapping

At run time, the resource scheduler on the target system adopts our proposed EIM algorithm to further optimise the task mapping at scenario level when a new workload scenario is detected. The EIM algorithm, which is outlined in Algorithm 4, can be divided into a static part and a dynamic part similar to the STM of Section 3.2.4. The static part is used to capture the intra-application dynamism in those inter-application scenarios with only a single active application. For these inter-application scenarios $se_i \in SE$, we have determined – using design-time DSE – optimal or near optimal mappings (optimized for O_{pb} and O_e) for each intra-application scenario $sa^i_j \in SA^i$ of se_i.

When the system detects a new workload scenario, the algorithm will first choose the corresponding optimal mapping – as derived from the design-time DSE

stage and stored in the *scenario database* – for each application active in the detected workload scenario as the initial mapping. This process is implemented in the function of line 1 of Algorithm 4 which will be explained in detail in the next paragraph. As the database stores mappings for the intra-application scenarios of each single application, its size typically is relatively small.

If there is only a single application active in the workload scenario, then the initial mapping will be chosen from one of the following two statically derived mappings, based on the system execution mode: the mapping with the maximal throughput under a given energy budget (the mapping optimized for O_{pb}) or the mapping with the minimal energy consumption (the mapping optimized for O_e). Hereafter, as shown in lines 2-3 of Algorithm 4, the algorithm will directly return the initial mapping as the final mapping decision. Otherwise, if there are multiple applications active simultaneously, then the mapping with maximal throughput (the mapping optimized for O_p) or minimal energy consumption (based on the system mode) for each active application will be chosen as initial mappings. These initial per-application mappings will then simply be merged together to form the initial mapping for the complete workload scenario. Here, there are two reasons for not choosing the mapping with maximal throughput under a certain energy budget as the initial mapping in the energy-aware high performance mode. First, the communication locality behaviour of the mapping with maximal throughput under an energy budget typically is not as good as the one with maximal throughput without an energy budget. Our run-time algorithm exploits this locality incorporated in the initial per-application mappings for further improvement of the workload scenario mapping. Second, we will consider the energy constraints during the mapping optimization process at run time, so we do not yet have to consider an energy budget for the initial mapping in the case of an active multi-application workload scenario.

The dynamic part of our EIM algorithm is only used for those workload scenarios that contain multiple simultaneously active applications and is outlined in lines 4-16 of Algorithm 4. It aims at further optimizing the initial mapping found during the static part of the EIM algorithm, as described above. To this end, it distinguishes the system execution mode to take different strategies for mapping optimization. As described before, our target MPSoC system can run under energy-aware high performance and energy saving modes. We will use different strategies in these two modes targeting different optimization objectives. These two strategies in the dynamic part of the EIM algorithm are described below. For the propose of a better understanding of our algorithm, the metrics used in algorithms 5 and 6 are shown in Table 3.3.

Performance Optimization When the MPSoC system is running under the energy-aware high performance mode, our algorithm will optimize the mapping for the active multi-application scenario with the objective to minimize the system metric O_{pb}. Consequently, the optimal mapping for each scenario is the one that has the minimal O_{pb} among all the possible mappings under energy budget of b_i for workload scenario s_i. It is, however, extremely hard to find the optimal mapping for each workload scenario at run time because of the following reasons. Firstly, as one cannot obtain the true value of O_p before actually executing the application on the target platform, an estimated $O_p^{'}$ needs to be used to guide

Table 3.3: Metrics used in Algorithms 5 and 6

Metrics	**Description**
(μ_j, η_j)	the mapping proposed by app_j
M_p^j	the maximal usage in U under the mapping of (μ_j, η_j)
V_p^j	the maximal usage variation in U under the mapping of (μ_j, η_j)
L^j	the performance loss of a remapping from (μ, η) to (μ_j, η_j)
M_p^k	the minimal M_p among M_p^j
(μ_k, η_k)	the mapping with M_p^k
V_p^k	the V_p of the mapping (μ_k, η_k)
W_p^t	the minimal $V_p + L$ among $V_p^j + L^j$
(μ_t, η_t)	the mapping with W_p^t
V_p^t	the V_p of the mapping (μ_t, η_t)
E_w	the minimal mapping energy consumption among E_{ij}
(μ_w, η_w)	the mapping with E_w

the algorithm to find the optimal mapping. Here, there exists of course a clear accuracy/overhead trade-off between different estimation techniques. Efficient but less accurate run-time mapping-performance estimation techniques may lead to sub-optimal mappings, while the high overhead of more accurate techniques may neutralize the performance benefits of the mapping optimization itself. Secondly, the mapping problem is NP-complete, as was mentioned before. It is unrealistic for a run-time mapping algorithm to explore the entire searching space to determine the optimal mapping for a scenario. An alternative method is using heuristics to search a part of the mapping space which may contain the optimal or a near optimal mapping.

To solve the above problems, we change the objective of performance into two other metrics: M_p and V_p that represent the maximal usage and usage variation in U, where U is an array of processor usages (calculated by Equation 2.1) with a total number of $|PE|$ elements, according to the observation found in Section 2.3.1 that M_p and V_p correlate with O_p. These two metrics will be used to optimize the bottleneck of the application pipeline and balance the system workload. In this case, we do not need to use the metric O_p as the optimization objective, thereby addressing the first of the two above problems. Regarding the second problem, by using an optimization heuristic based on the metrics M_p and V_p, we aim at finding an optimal or near optimal mapping in a computationally efficient fashion. The rationale behind this heuristic is that a better mapping for the objective of high performance usually has smaller M_p and V_p values (see Section 2.3.1). For the purpose of restricting the energy consumption of the resulting mapping, we use the estimated energy consumption of a mapping $tm_j = (\mu_j, \eta_j)$ for workload scenario s_i given by Equation 3.5 and the energy budget b_i calculated by Equation 3.6. Here, the index e in E_{ke} represents the energy-optimized mapping stored in memory for app_k, to control the searching space of possible mappings. The details of Equation 3.5 will be explained in the next subsection. In Equation 3.6, the first part α is a user defined constant scaling factor set for the energy budget

Algorithm 5 IPO algorithm

```
//performance optimization for workload scenario s_i
iterativePOpt(μ, η, M_p, V_p, b_i):
1: for each active app_j:
2:     (μ_j, η_j) = getPSubstitute(μ, η);
3:     if (μ_j, η_j) != (μ, η):
4:         U = peUsage(KPN_app_active, MPSoC, μ_j, η_j);
5:         M_p^j = maxPUsage(U);
6:         V_p^j = varPUsage(U);
7:         L^j = perfLoss(app_j, μ, η, μ_j, η_j);
8: M_p^k = min(M_p^j);
9: if M_p^k < M_p:
10:    (μ*, η*) = (μ_k, η_k);
11:    iterativePOpt(μ*, η*, M_p^k, V_p^k, b_i);
12:else:
13:    W_p^t = min(V_p^j + L^j);
14:    (μ*, η*) = (μ_t, η_t);
15:    if (μ*, η*) == (μ, η):
16:        return (μ, η);
17:    else:
18:        iterativePOpt(μ*, η*, M_p^t, V_p^t, b_i);
```

and the second part represents the estimated minimal energy consumption for a workload scenario.

$$E_{ij} = E_p^{'} + E_m^{'} \tag{3.5a}$$

$$E_p^{'} = \sum_{pe_k \in PE_{active}} (DP_k * U_k + SP_k * max(U)) \tag{3.5b}$$

$$E_m^{'} = DM * \sum_{c_t \in C_{ij}^{mem}} (ec_t) + SM * max(U) \tag{3.5c}$$

where PE_{active} is the set of processors active for scenario s_i under mapping tm_j. C_{ij}^{mem} is the set of application FIFO channels in scenario s_i that are mapped onto the shared memory on the target system under mapping tm_j. U_k is the usage of processor $pe_k \in PE$. SP_k and DP_k refer to the static and dynamic power consumption for processor pe_k. And SM and DM respectively represents the static and average dynamic power consumption (for read/write transactions) of the shared memory on the target MPSoC system.

$$b_i = \alpha * \sum_{active\_app_k \in s_i} E_{ke} \tag{3.6}$$

The mapping algorithm for the energy-aware high performance mode is outlined in Algorithm 5, which will be executed in an iterative fashion. The starting mapping used in this algorithm is the one derived from Algorithm 4. In each iteration, it first proposes a new mapping for each active application as shown in line

2 of Algorithm 5. In this process, the algorithm searches the mapping space using the following greedy pattern: it checks the processors in U in descending order to determine whether the KPN application in question has a task or a bundle of adjacent, communicating tasks[5] resident on this processor. If so, then the algorithm finds a possible substitute processor for the task/adjacent tasks that satisfies the following conditions:

1. The $M_p^{'}$ of the new mapping is smaller than the M_p of the old mapping
2. If the previous condition cannot be satisfied, then the algorithm tries to find a substitute processor for which the resulting $M_p^{'}$ is equal to M_p and $V_p^{'}$ is smaller than V_p. If the first condition was satisfied, then this condition will never be used in this particular iteration
3. The estimated energy consumption of the new mapping should be smaller than the energy budget b_i.

The above process proposes new mappings for those applications that satisfy the conditions (for the other applications, the mapping remains unaltered). These newly proposed mappings are either a mapping that has a minimal $M_p^{'}$ (if condition 1 has been satisfied) or a mapping with minimal $V_p^{'}$. However, in the above process, it can also be the case that there are multiple new mappings proposed for an application, e.g. when there are multiple tasks (or task bundles) that can be remapped and for which the above conditions hold. In these cases, we use another metric, L, to decide on the final proposed mapping, where the value of L needs to be minimized. The metric L tries to capture the performance loss of a task remapping for the application in question[6] and is calculated using Equation 3.7.

$$L = \sum_{p_k \in B_i^j} ((et_k^j - et_k^i) + \sum_{c_{kt} \in C_k} (ec_{kt}^j - ec_{kt}^i)) \tag{3.7}$$

Here, we mark the task/task bundle that needs to be remapped from pe_i to pe_j as B_i^j. C_k represents the set of communication channels connected between p_k and other tasks in the target workload scenario.

After the algorithm has proposed a new mapping for each application, the next step is to select the *most effective* among these remapping proposals to be used for the next optimization iteration of the algorithm based on the metrics of each new mapping calculated in lines 4-7 of Algorithm 5 or return a mapping as the final one. This whole process is shown in lines 8-18 of Algorithm 5. If no new mapping has been proposed for any of the applications in the workload scenario in the previous step, then the input mapping will be returned as the final optimized result. Otherwise, we use the following conditions to select the most effective remapping for the next iteration of the algorithm:

1. If there is one and only one proposed mapping that has the minimal $M_p^{'}$ and this $M_p^{'}$ is smaller than the M_p of the original mapping, then this mapping will be passed to the next mapping optimization iteration (lines 9-11 in Algorithm 5).

[5]Mapping such a task bundle to a single processor is the outcome of the design-time mapping optimization to reduce communication overhead.

[6]We note that L can be negative, implying that the task/task bundle has a higher affinity with the processor it is proposed to be mapped on.

Algorithm 6 IEO algorithm

```
//energy optimization for workload scenario s_i
iterativeEOpt(μ, η, E):
1: for each active app_j:
2:     (μ_j, η_j) = getESubstitute(μ, η);
3:     E_ij = energyCons(KPN_app_active, MPSoC, μ_j, η_j);
4: E^w = min(E_ij);
5: if E^w ≥ E:
6:     return (μ, η);
7: else:
8:     (μ*, η*) = (μ_w, η_w);
9:     iterativeEOpt(μ*, η*, E^w);
```

2. If the first condition has not been satisfied, then the proposed mapping with $min(V_p^{'} + L)$ will be taken as the input mapping for the next iteration (lines 12-18 in Algorithm 5). The rationale behind this is that the algorithm tries to gradually optimize the mapping for the entire workload scenario while keeping the performance loss for a single application due to task remappings as small as possible (i.e., taking into account the processor affinity of the tasks proposed to be remapped).

Energy Optimization The algorithm used in the energy saving system mode is shown in Algorithm 6, which is similar to the algorithm for the high performance mode. It will iteratively optimize the mapping with the objective O_e for a unit of input workload (e.g., frame in the domain of multi-media applications). For the purpose of energy savings, we need not only to consider the dynamic energy consumption but also the static energy consumption. The energy consumption E_{ij} of a mapping (μ_j, η_j) for workload scenario s_i is calculated by Equation 3.5, where $E_p^{'}$ is the dynamic and static energy consumed by all active processors and $E_m^{'}$ represents the dynamic and static energy consumption of the shared memory. This relatively simple energy model is built on several assumptions of the target architecture: 1) the power model used for the shared memory in the system already includes the power consumption of the bus connected to it; 2) for simplicity, we ignore the energy consumption caused by resource contention and communication delays. Consequently, the system active time for a specific workload scenario is simply assumed to be $max(U)$, which is subsequently used to calculate the static energy consumption. Note that the application of techniques such as dynamic power management (DPM) and dynamic voltage scaling (DVS) are beyond the scope of this work.

The mechanism for searching the mapping space to find the energy optimized mapping is implemented in the function listed on line 2 of Algorithm 6. In each iteration, it greedily finds the mapping with minimal energy consumption for each active application in the workload scenario by just remapping a single B_i^j (task/-task bundle). Similar to Algorithm 5, Algorithm 6 first proposes a new mapping for each of the active applications, after which the mapping with minimal energy consumption among the proposed mappings will be used in the next optimisation

Table 3.4: Studied application workload scenarios.

Inter-app scenario	Workload scenario
A1	mjpeg 7
A2	sobel 0
A3	mp3 2
A1A2	mjpeg 7, sobel 0
A1A3	mjpeg 7, mp3 2
A2A3	sobel 0, mp3 2
A1A2A3	mjpeg 7, sobel 0, mp3 2

iteration. However, if the condition on line 5 of Algorithm 6 is satisfied, then the input mapping will be returned as the final optimization result.

3.3.3 Experiment

In this subsection, we present a number of experimental results in which we investigate various aspects of our EIM algorithm using the extended Sesame simulator of Section 3.1 (similar to Section 3.2.5.1). More specifically, in each system execution mode, we compare the algorithm to three different run-time mapping algorithms using the optimization objective of the system mode. For the high performance mode, we compare our EIM algorithm to the following algorithms: Task Processor Affinity (TPA) [83] which uses the affinity between tasks and processors to greedily determine a mapping without considering resource contention, and Output-Rate Balancing (ORB) [22] which aims at balancing the computation and communication load of each processor. For the energy saving mode, we compare our algorithm to TPA and Iterative Energy-Aware Task Mapping (IEATM) [106, 49]. Moreover, we also compare the run-time mapping results to the results of optimal mappings for each workload scenario. These optimal mappings have been statically determined by means of design-time DSE as introduced in Section 3.3.2.1.

For our experiments, we use the multi-media applications considered in the experiments of Section 3.2.5: MJPEG, MP3 and Sobel (95 different workload scenarios in total) which are denoted as A1, A2 and A3 respectively in Table 3.4 and Figures 3.9 and 3.11. At design time, we have determined three pre-optimised mappings for each intra-application scenario in each application as explained in Section 3.3.2.1. That means that we need to store 45 optimal mappings in system memory (i.e., the scenario database).

With respect to the target architecture, we target a heterogeneous MPSoC containing 5 different processors with different computational and energy characteristics, connected to a shared bus and memory. For this target MPSoC system, we assume that these 5 processors can execute all application tasks. However, we want to stress that our approach is not restricted to this assumption: dedicated processors could also be used in the target system. In that case, some additional information like the possible target processor of each task is needed for deriving a correct mapping.

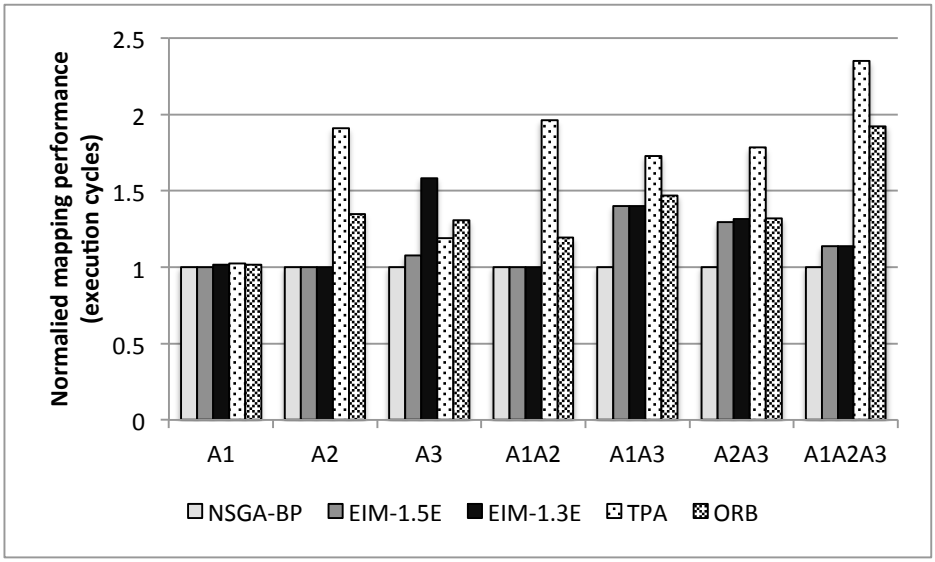
3.3.3.1 Performance Optimization Experiments

The experiments in this subsection concern the evaluation of our run-time mapping algorithm in high performance mode, considering different inter- and intra-application workload scenarios.

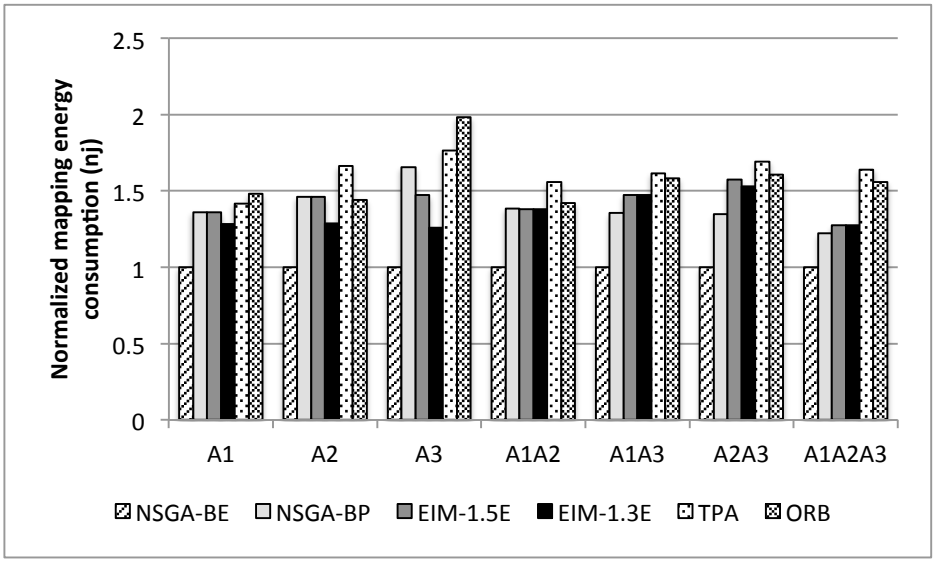
In the first experiment, we study the run-time mapping behaviour in the occurrence of different inter-application scenarios. To this end, we focus on the subset of workload scenarios that have the heaviest computational demands in each inter-application scenario. These workload scenarios are listed in Table 3.4, where the first column specifies the encoded name (in terms of A1, A2 and A3) for each inter-application scenario and the second column specifies the intra-application scenarios (labeled by the integer following the application name) used to form the workload scenario. For the scaling factor α of the energy budget in our EIM algorithm (see equation 3) we use the values 1.5 and 1.3 in our experiments.

The experimental results are shown in Figure 3.9. In Figure 3.9a, we compare the performance of the mappings resulting from the EIM, TPA, and ORB algorithms as well as from NSGA-II-based design-time DSE. The energy consumption of these mappings is shown in Figure 3.9b. In these two figures, the bars of NSGA-BP and NSGA-BE respectively represent the mappings with best performance and minimal energy consumption found by the NSGA-II-based design-time DSE. These are used as a baseline for comparison. From Figure 3.9a, we can see that our EIM algorithm in most cases produces a better mapping for the tested workload scenarios than the TPA and ORB algorithms. For the workload scenarios in which only a single application is active (i.e., bars for A1, A2 and A3) our EIM algorithm directly uses the mapping from design-time DSE, which results in a mapping performance that is very close or even equivalent to the optimal mapping. However, although the mappings have similar performance, they could still have a different energy consumption behaviour. In the case of our EIM algorithm, we use the energy budget in the search for an efficient mapping to limit the energy consumption of the resulting mapping. Consequently, and as shown in Figure 3.9b, the EIM algorithm can yield mappings for single-application workload scenarios that are more energy efficient than the ones obtained by NSGA-BP.

In the workload scenarios with multiple simultaneously active applications, we can see that the EIM algorithm yields clear performance improvements compared to the other three run-time task mapping algorithms, especially in the case of workload scenario A1A2A3. By setting the parameter α of our EIM algorithm to different values, we can notice that in some workload scenarios, like A1, A3 and A2A3, the mapping performance with a higher energy budget is better than the one with a lower energy budget. However, in other workload scenarios, there is no such behaviour. This can be explained by the fact that for the latter workload scenarios the energy budget is big enough for the algorithm with a lower energy budget to find a mapping that is as good as the one found by EIM with a higher energy budget. In Figure 3.9b, we can see that even if we have an energy budget in our EIM algorithm, the actual energy consumption of the final mapping may still exceed the energy budget: like for EIM-1.5E in the A2A3 workload scenario and for EIM-1.3E in a few other workload scenarios. This is caused by estimation inaccuracies of the energy model used in our algorithm. Even if the estimated energy consumption of a new mapping is under the predefined energy budget, the



(a) Performance of mappings from different algorithms

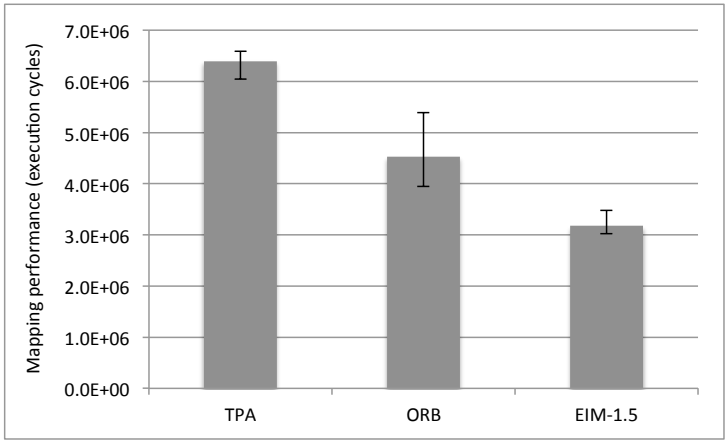


(b) Energy consumption of mappings from different algorithms

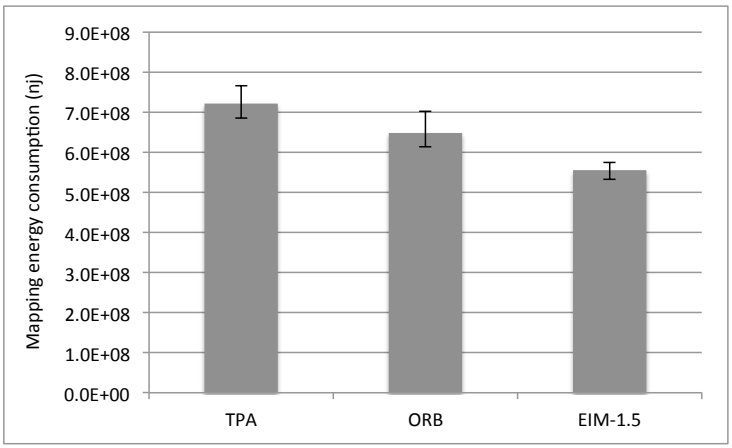
Figure 3.9: Algorithm comparison in high performance mode (inter-application scenarios).

actual resulting system energy consumption after the remapping has taken place may still not fully satisfy our desired energy budget.

In the second experiment, all the intra-application scenarios for one particular inter-application scenario, namely A1A2A3, are considered as the experimental workload. This means that there are 33 workload scenarios in total. We compare the average performance and energy consumption of the optimized mappings as obtained by the different algorithms. The results are shown in Figure 3.10a and Figure 3.10b respectively. The error bars in the graphs show the variability of the results. From Figure 3.10a, we can see that the mappings from our EIM algorithm with a scaling factor $\alpha = 1.5$ achieve the best average performance among the investigated algorithms. Even the worst mapping performance among all the 33 workload scenarios of our EIM algorithm is still better than the best



(a) Performance of mappings from different algorithms



(b) Energy consumption of mappings from different algorithms

Figure 3.10: Algorithm comparison in high performance mode (intra-application scenarios).

one in any of the other two algorithms. Comparing the performance of each final mapping obtained by our EIM algorithm with the ones from TPA and ORB in all our tested 33 workload scenarios, we measure ranges of 56.3%-66.6% and 11.5%-42.3% of performance improvement respectively. Figure 3.10b shows the average energy consumption of the final mappings used in Figure 3.10a. The results in this figure illustrate that the mappings from our EIM algorithm have the lowest average energy consumption. Considering the energy consumption of each final mapping, our EIM algorithm achieves, respectively, a 20.7%-29.6% and 6.1%-19.0% improvement for energy savings compared with TPA and ORB.

3.3.3.2 Energy Optimization Experiments

Considering the energy saving system mode, we also investigate our run-time mapping algorithm considering different inter- and intra-application workload scenarios. The results of the different mapping algorithms when the primary objective is energy optimization and when using the subset of inter-application scenarios from Table 3.4 are shown in Figure 3.11. From this experiment, we can see that

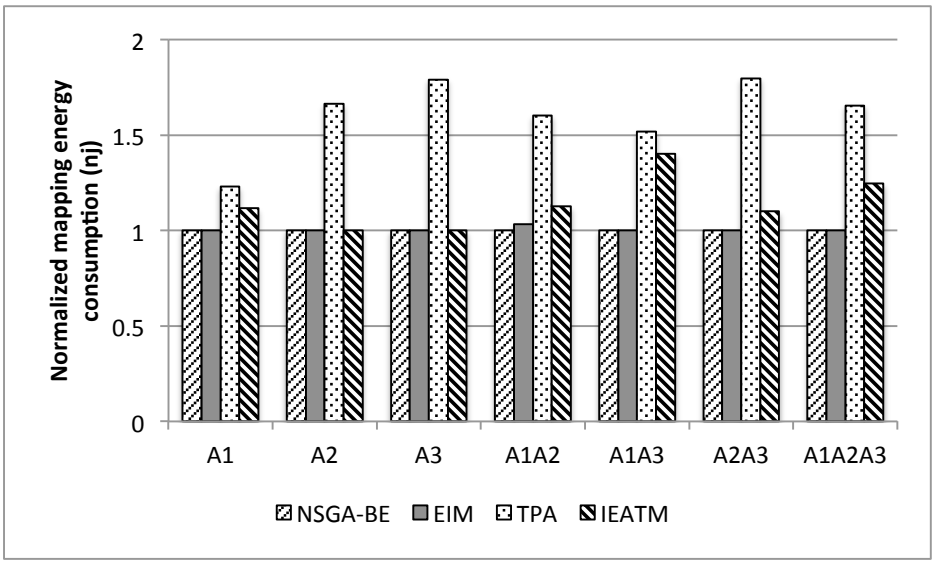


Figure 3.11: Algorithm comparison in energy saving mode (inter-application scenarios).

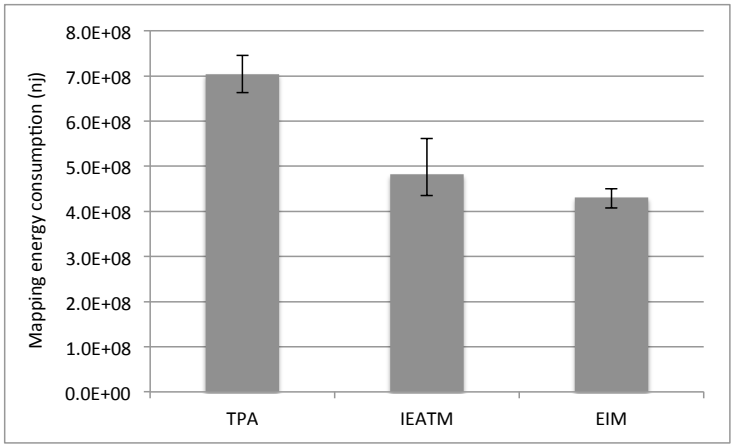


Figure 3.12: Algorithm comparison under energy saving mode (intra-application scenarios).

our EIM algorithm can efficiently produce near optimal (in terms of energy consumption) mappings. Comparing the investigated mapping algorithms, besides the NSGA-BE, our EIM algorithm overall shows the best results. In the single-application workload scenarios, EIM simply uses the mapping optimized at design time. For the multi-application scenarios, we can notice that our EIM algorithm clearly improves on the other algorithms, finding mappings that are almost as good as the ones obtained by NSGA-BE.

Figure 3.12 shows the results of average energy consumption of mappings optimized for energy consumption by the different mapping algorithms when considering all 33 workload scenarios of inter-application scenario A1A2A3. The results in this figure illustrate that the EIM algorithm performs much better than the other two algorithms. For each single workload scenario, the EIM algorithm achieves

Table 3.5: Run-time cost of different algorithms for performance optimization.

Algorithms	**Total task migration number**	**Normalized total algorithm computation time**
TPA	12	1.0
ORB	36	22.5
EIM1.5	25	651.2

energy improvements of in between 50.5%-54.2% and 5.5%-20.2% as compared with TPA and IEATM respectively.

3.3.3.3 Run-time Cost

Here, we would like to give an intuition of the run-time cost of our approach in terms of the number of tasks that need to be migrated and the computation cost of the algorithm. In this experiment, the intra-application scenarios in inter-application scenario A1A2A3 with only application MP3 changing its execution mode will be considered as the targeting scenarios. This means that the execution mode of each application is: MJPEG (7), Sobel (0) and MP3 (0/1/2). These three scenarios will be executed in sequence to find out the total task migration number and total algorithm computation time by applying different approach. The total number of tasks in each scenario is 41. Table 3.5 shows the cost of different approaches for mapping performance optimization, where the total algorithm computation time of each approach is normalized to the one of TPA. From the results, we can see that the baseline approach TPA has the minimal run-time cost in both migration cost and algorithm computation time. Our proposed approach EIM has the highest algorithmic computation time. However, we believe that the computation cost of our EIM algorithm is still acceptable as it just needs a few milliseconds (on an CPU with 2.17GHZ) to optimize the mapping for each workload scenario in our test case. Here, we would like to note that we have not yet performed any effort to optimize our EIM algorithm to reduce its computational cost. The run-time cost of the approaches for mapping energy optimization is listed in Table 3.6, where our approach shows the highest total task migration cost while the algorithmic computation time is relatively low as the diversity of the pre-optimized mappings is small.

Compared to the algorithmic computation cost, the run time task migration overhead typically is more substantial for MPSoC systems. From both experimental results, we can see that the task migration cost of our EIM is relatively heavy. For this reason we make the assumption that each workload scenario will execute for a long enough time so that the system is able to benefit from our EIM algorithm. Further research will be provided in the next chapter to exactly determine at which switching granularity of workload scenarios a dynamic task mapping algorithm could benefit from remapping.

Table 3.6: Run-time cost of different algorithms for energy optimization.

Algorithms	**Total task migration number**	**Normalized total algorithm computation time**
TPA	8	1.0
IEATM	9	64.0
EIM	16	62.2

3.3.4 Conclusion

In this section, we have proposed a run-time mapping algorithm, called EIM, for MPSoC-based embedded systems to improve their performance and energy consumption by capturing the dynamism of the application workloads executing on the system. This algorithm is based on the idea of application scenarios and consists of a design-time and run-time phase. The design-time phase produces mappings for intra-application scenarios targeting different optimization objectives after which the run-time phase aims to continuously monitor the changes in workload scenarios on the underlying system and trying to perform iterative mapping optimization to improve the system performance and/or energy consumption based on the optimal mappings of corresponding applications explored in the first stage. Combining these two steps, the proposed approach can dynamically find a near optimal mapping for multiple executing applications, while it is also capable of running single applications under the optimal mapping (derived from design-time DSE) with respect to different optimization objectives. In various experiments, we have evaluated our algorithm and compared it with other run-time mapping algorithms. The results clearly confirm the effectiveness of our algorithm.

3.4 Improving MPSoC's Adaptivity with Hybrid Task Mapping

In the second section of this chapter, the scenario clustering based task mapping approach (STM) was proposed for a homogeneous MPSoC system to solve the scalability problem of general hybrid task mapping approaches. In this approach, task remapping is triggered by the change of inter-application scenarios and the violation of application specific objectives during the execution of intra-application scenarios in a certain inter-application scenario. However, the drawback of this approach is that it can't solve the flexibility problem with regard to supporting new applications on the target system where the design-time analysis needs to be redone entirely. To solve both the scalability and flexibility issues, in the third section, we proposed a new hybrid task mapping approach which solves the scenario-level mapping problem by a divide-and-conquer method (EIM) for a heterogeneous MPSoC system whenever the scenario on the system changes. This approach optimises task mappings right after new workload scenarios are detected. It does not, however, contain any further mapping optimisation during the execution of a certain workload scenario as it is provided by the STM algorithm.

In this section, these two run-time mapping algorithms will be combined together for a heterogeneous MPSoC system to improve the system efficiency where

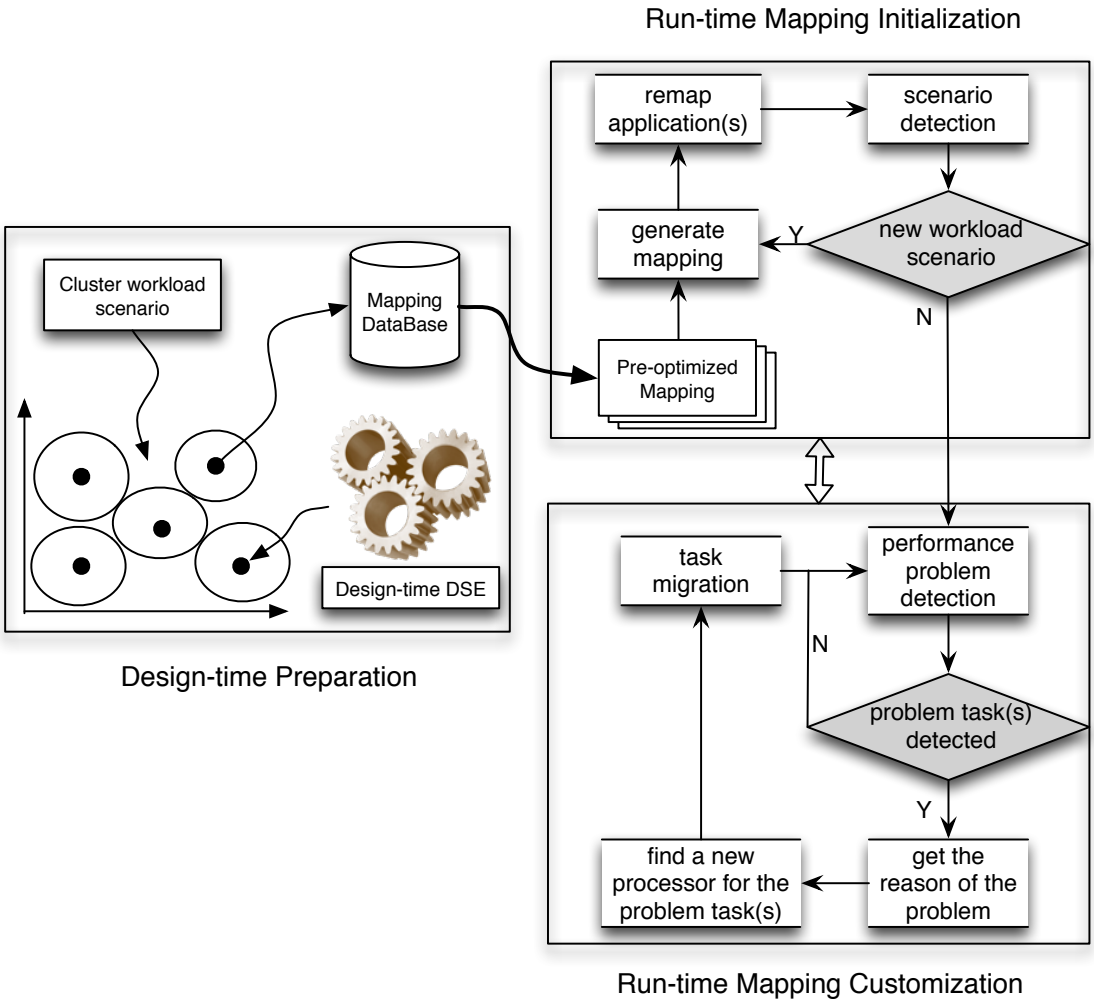


Figure 3.13: The workflow of HTM

the novel hybrid task mapping technique (with energy-aware high performance mode) of the third section is applied for scenario mapping initialisation and the run-time on-the-fly mapping heuristic (the dynamic part of STM) of the second section is extended (on a heterogeneous system) for dynamic QoS management during scenario execution on the target system. This combined task mapping approach is referred as Hybrid Task Mapping (HTM). By using this HTM approach, our goal[7] is to optimise the mapping of each detected workload scenario $s_i \in S$ for minimising the performance objective of O_{pb} (energy consumption objective included, see Section 3.3.1) and where each application also satisfies its own objective O_β. In this section, O_β is the application specific performance requirement similar to Section 3.2.2. The target application set and architecture are the same as the ones considered in Section 3.3

3.4.1 Workflow of HTM Approach

As shown in Figure 3.13, the entire workflow of our HTM approach can be divided into three steps: design-time preparation, run-time mapping initialization

[7]Based on the problem definition of Section 3.2.2 and Section 3.3.1.

and run-time mapping customization. In the step of design-time preparation, two optimized mappings for each intra-application scenario of each application are prepared by exploring the corresponding mapping space. These pre-optimized mappings will be stored in system memory for run-time mapping initialization/optimization. At run time, when the system detects a new workload scenario, our HTM algorithm will try to produce a good mapping for the active applications in the scenario using (a combination of) the stored per-application mappings derived from the design-time preparation step. Note that, according to the assumption (coarse-grained workload scenarios) we made at the start of this chapter, each workload scenario will execute long enough to justify a possible remapping of application tasks. Otherwise, a trade-off needs to be made between the cost of remapping and the mapping performance improvement, which will be discussed in the next chapter. This process of determining a new mapping for all applications when a new workload scenario has been detected, is referred to as mapping initialization. The objective of this process is to maximize the system throughput under the predefined energy budget (or, in other words, minimize O_{pb}). The mapping initialisation, which uses the stored mapping information of isolated applications, may not immediately lead to finding the optimal system mapping for a complete identified workload scenario (i.e., the combination of applications that form the scenario). Therefore, during the execution of a certain workload scenario, the HTM algorithm will try to actively further improve the mapping performance when application-specific objectives are (about to be) violated. To this end, it continuously monitors the system and tries to perform relatively small mapping customizations to gradually further improve the system performance. Evidently, to reduce migration overheads, the algorithm aims at keeping the number of required task migrations as low as possible. This process is called mapping customization. The details of these three steps will be explained in the following subsections.

3.4.1.1 Design-time Preparation

At design time, the mappings with minimal O_p and O_{pb} will be searched by the scenario-based DSE approach of Section 3.2.3 for all intra-application scenarios in each isolated application (i.e., in those inter-application scenarios with only a single active application). As shown in Figure 3.13, it would also be possible to cluster intra-application scenarios of applications, and only determine mappings with minimal O_p and O_{pb} for an entire cluster of intra-application scenarios like the work in Section 3.2. This would further reduce the number of mappings that need to be explored and stored. However, in this section, we assume that mappings with minimal O_p and O_{pb} are searched for all separate intra-application scenarios of applications. Different with the design-time DSE in Section 3.3.2.1 where three mappings need to be explored for each intra-application scenario of each separate application, here, we only need to explore two mappings because the energy optimisation objective O_e considered in the previous section is ignored in this section.

3.4.1.2 Run-time Mapping Initialisation

In the mapping initialization stage, we use the EIM algorithm with the energy-aware high performance mode introduced in Section 3.3.2.2 to find a good initial

mapping for a newly detected workload scenario. This mapping optimisation process is triggered by the change of workload scenarios. The mapping derived from the EIM algorithm will be initialised on the target system for the detected workload scenario. During the execution of a workload scenario, the mapping customisation process will be applied to further optimise the mapping. As the EIM algorithm can be directly applied in this work, we will not provide more details about how it works.

3.4.1.3 Run-time Mapping Customisation

After the mapping is initialized for the active workload scenario, the system will monitor the execution of this workload scenario. As mentioned before, the application-specific objective is used to determine whether or not a performance problem arises (triggering the mapping customisation process) at run time. Here, we assume that the target MPSoC should, in principle, be dimensioned such that it can accommodate all possible target applications but that a particular application's performance objective may be violated due to a bad mapping. When the system detects the violation of application specific objectives, the dynamic part of the STM algorithm (see Section 3.2.4) is applied to customise the current mapping. Notice that, in the second section of this chapter, the STM algorithm was applied for optimising task mapping on a homogeneous MPSoC system. However, in this section, our target system is a heterogeneous MPSoC system. Consequently, there is a little difference between the STM implementation in Section 3.2.4 and this section.

According to the workflow of the dynamic part of the STM algorithm, the first step is to find the critical task in the problematic application under the current task mapping and also the reason that causes the problem. The second step is to remap the critical task (or task bundle) onto the target system. In the first step, the STM algorithm for the heterogeneous MPSoC system of this section has the same behaviour compared with the one for a homogeneous MPSoC system that is introduced in Section 3.2.4.1. However, in the second step, the difference between these two versions of the STM algorithm exists in the process of remapping the critical task under the reason of *poor locality*. It means that the MCC approach is the main difference between these two versions where the MCC as expressed in Equation 3.8 for this section (find the index m, n and z) is more complex than the one in Section 3.2.4.2 (find the index m and n).

$$min(Circle\_Cost(p_i)^z_{mn}),\ with\ 0 \leq m, n < |P|, m \leq i \leq n,\ 0 \leq z < |PE| \quad (3.8)$$

where:

$$Circle\_Cost(p_i)^z_{mn} = \sum_{\substack{m \leq k \leq n \\ p_k \mapsto pe_z}} et^z_k + \sum_{\substack{m \leq k \leq n \\ p_k \mapsto pe_z}} \sum_{0 \leq j < |P|} ec^z_{kj} \quad (3.9)$$

where et^z_k denotes the execution time of task k (neighbours of task i) for PE z (might different with the PE onto which the task currently mapped), and ec^z_{kj} denotes the communication overhead between tasks k and j (the task communicates with task k). $|P|$ and $|PE|$ represent the total number of tasks in the target

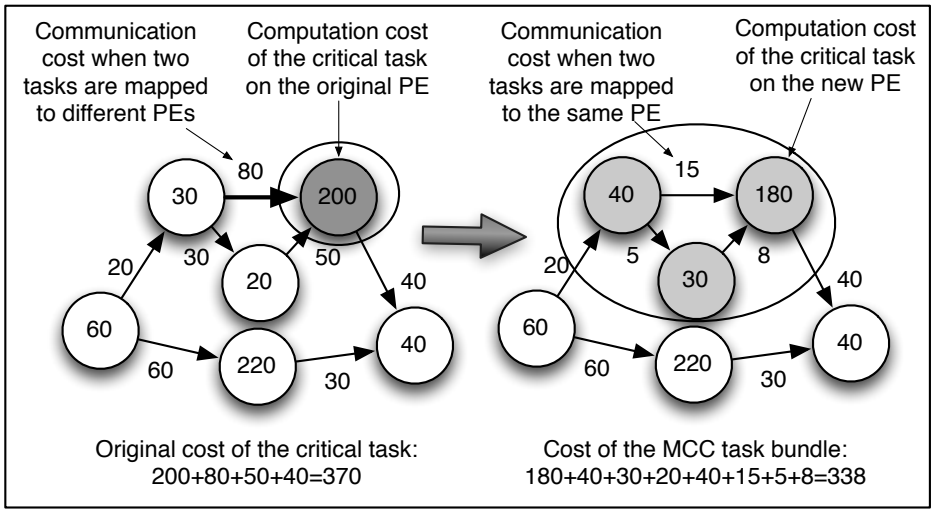


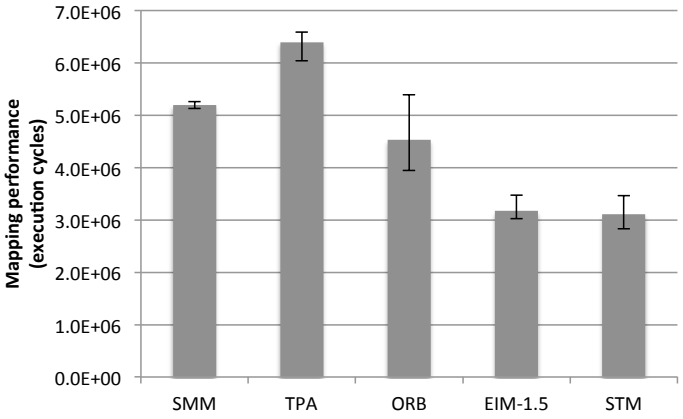
Figure 3.14: Example of an MCC for a critical task (gray task on the left-hand side)

application and the total number of processors on the target system. The symbol $a \mapsto b$ means a mapped onto b.

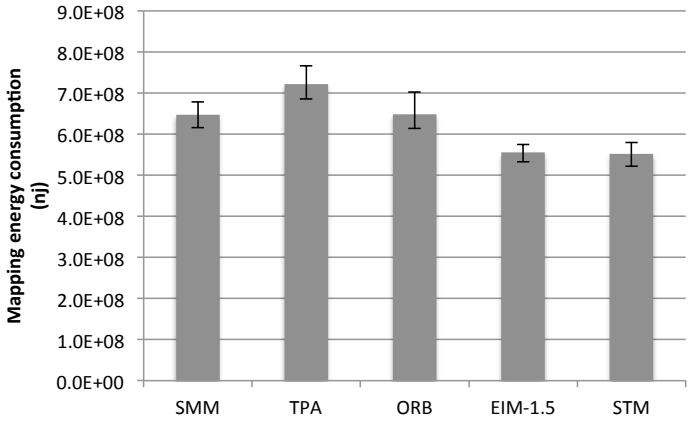
To find the MCC for the critical task on a heterogeneous MPSoC system, the heterogeneity of processors should be taken into account as the execution time of a task is not the same on different processor. Figure 3.14 shows an example of an MCC (indicated by the oval on the right-hand side) consisting of a task bundle of three tasks, including the critical task (gray task). As the task to processor binding is part of the calculation of the MCC of the critical task, implying that the binding with the minimal MCC is known after this calculation, the substitute PE is the processor used in this binding. However, if the MCC solely consists of the critical task itself, then the critical task will be mapped together with the neighbouring task with which the critical task has the heaviest communication to the processor that yields the minimal task cost for the combined tasks. After the substitute PE has been found, the FIFO channels between the tasks that need to be remapped are either mapped as internal communication onto the new PE (if communicating tasks are mapped onto this PE) or onto the system bus.

3.4.2 Experiments

In this subsection, we present the experimental results in which we investigate whether the extended STM algorithm is able to further improve the system efficiency based on the optimised mapping by the EIM algorithm on the previously mentioned Sesame simulator. The experiment setups are similar to those of Section 3.3.3 where our three multi-media applications and the heterogeneous MPSoC are used again in this work. For all the three target applications, there are 15 intra-application scenarios (MJPEG : 11, Sobel : 1 and MP3 : 3) in total. It means that we need to store 30 pre-optimised mappings in the memory of the target heterogeneous MPSoC system. As the results of mapping initialisation by the EIM algorithm is already presented in Section 3.3.3, we will not further study the mapping initialisation part of the HTM algorithm. To study the behaviour



(a) Performance of mappings from different algorithms



(b) Energy consumption of mappings from different algorithms

Figure 3.15: Comparing the quality of mapping solutions derived from different run-time mapping algorithms for the intra-application scenarios of A1A2A3

of the STM part of our HTM approach, the most complex inter-application scenario where all the target three application active simultaneously (A1A2A3 in Table 3.4) will be considered as the target workload scenario for investigating the behaviour of the extended STM algorithm. For the purpose of comparison, the TPA and ORB algorithms as mentioned in Section 3.3.3 are again considered in this work. Moreover, we also compare the run-time mapping results of our HTM approach to the results of Simple Mapping Merge (SMM) which simply merges together the (statically derived) optimal mappings of each active application for the corresponding intra-application scenario.

Figures 3.15a and 3.15b show the scenario execution time and energy consumption of mappings found by the SMM, TPA, ORB, EIM and STM algorithms in all the intra-application scenarios of a particular inter-application scenario, namely A1A2A3. In the case of the STM algorithm, the algorithm uses and tries to improve on the results of the EIM algorithm, as sketched in Figure 3.13 (i.e., the HTM algorithm). Using inter-application scenario A1A2A3, there are 33 workload scenarios in total that are considered as the application workload in this experiment. The error bars in the graphs show the variability of the results. From

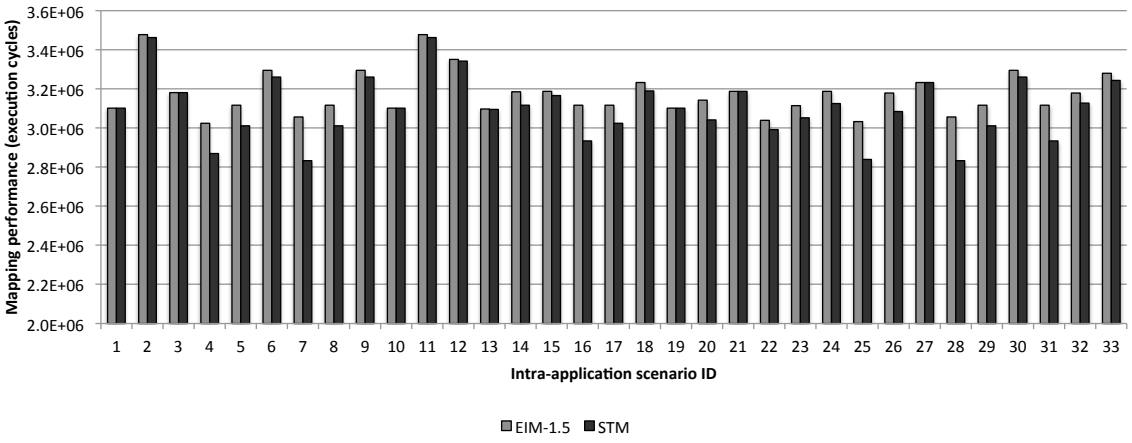


Figure 3.16: Final mapping comparison of EIM and STM for all intra-application scenarios of A1A2A3

Figure 3.15a, we can see that the mappings from our STM and EIM algorithm with a scaling factor $\alpha = 1.5$ achieve the best average performance among the investigated five algorithms. Note that the mapping customization process is applied during the execution of a certain workload scenario after the mapping initialization process. The STM algorithm is therefore used to further optimize the mapping solutions derived from the EIM algorithm. Comparing the results from STM and EIM, we found that the STM algorithm can achieve an additional performance improvement of 2.2% on average for all 33 considered intra-application scenarios. The reason for this relatively small performance improvement is twofold. First, the mapping derived by EIM for each new workload scenario is already a near optimal solution, which implies that the potentials for further improvement by the STM algorithm are limited. Second, the STM algorithm is designed for the situation in which the (user-defined) application-specific performance objective is violated. However, as the mapping is already optimised by the EIM algorithm, such performance objective violations for an application will typically not be very large. When there is a (small) performance objective violation, the STM algorithm tries to find a new mapping by only making small changes to the old mapping in an on-the-fly manner. If the new mapping satisfies the pre-defined performance objective, then the STM algorithm will stop[8]. Figure 3.16 shows the details of the mapping performance comparison between the STM and EIM algorithms.

Figure 3.15b gives the average energy consumption of the final mappings as shown in Figure 3.15a. The results in this figure illustrate that the mappings from our STM and EIM algorithms have the lowest average energy consumption, where the STM algorithm achieves an additional energy improvement of 0.5% on average compared to the EIM algorithm. In this experiment, we have demonstrated that the STM algorithm is capable of further improving the mapping performance compared with the EIM algorithm, without sacrificing the energy consumption of the

[8]In case the violation cannot be remedied by the STM algorithm, the user can be notified and/or a strategy for graceful termination of applications could be used.

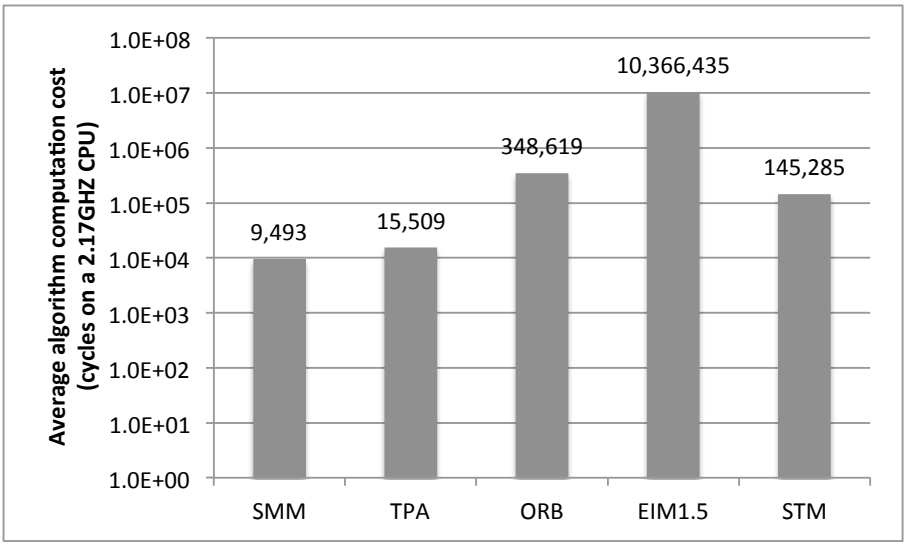


Figure 3.17: Average algorithm computation cost (cycles on a 2.17GHZ CPU) for intra-application scenarios of A1A2A3

mapping. Overall, compared with the mapping solutions derived from the SMM, TPA and ORB algorithms, the average performance and energy consumption of the final mapping solutions generated by applying the HTM algorithm (the mapping is first optimized by EIM followed by STM at run time) for the 33 workload scenarios in A1A2A3 improve by 67.2%, 105,9%, 45.9% (performance) and 14.6%, 23.5%, 14.9% (energy) on average respectively.

With regard to the run-time computational overhead, we investigate the computation cost of the run-time stage (EIM and STM) of our HTM algorithm and compare it to the overhead of the other run-time mapping algorithms. The results of an experiment in which we average the algorithmic overhead for executing different intra-application scenarios of A1A2A3 (the most complicated workload scenario where all three applications are active) are shown in Figure 3.17. Note that the time unit (cycles on a 2.17GHZ CPU) used in this figure is different from the time unit used for the mapping performance as presented in the previous figures, which are based on simulation cycles measured by the Sesame simulator. From Figure 3.17, we can see that the SMM approach has the smallest algorithmic cost. As in this approach there is no actual computation of a new mapping, it just involves memory access time to retrieve the pre-optimized mapping from the SDB for each active application. On the other hand, the EIM part of our HTM algorithm has the heaviest computational cost to find a new mapping, which happens at the detection of a new workload scenario (mapping initialization). For the other part of the HTM algorithm – the STM algorithm – the computation cost is much smaller.

3.4.3 Conclusion

Based on the EIM and STM algorithms, a new hybrid task mapping approach, called HTM, has been proposed for MPSoC-based embedded systems to improve

their performance by capturing the dynamism of the application workloads executing on the system. Our approach is based on the idea of application scenarios and consists of three steps: design-time preparation, run-time mapping initialization and run-time mapping customization. The design-time preparation exploits optimal mappings for each mode of each application which will be stored on the target platform for further mapping optimization. At run time, the mapping initialization process dynamically optimizes the mapping of the running application(s) with the objective of maximizing the throughput under a predefined energy budget based on the optimal mappings of the corresponding applications stored on the system when a new workload scenario emerges. During the execution of a certain workload scenario, mapping customization is performed to further improve the performance of the mapping under an application-dependent objective. In various experiments, we have evaluated our algorithm and compared it with other run-time mapping algorithms. These experiments indicate that our proposed approach can achieve considerable performance improvements (45.9% - 105.9%) and energy savings (14.6% - 23.5%) compared with the other algorithms for workload scenarios in which multiple applications are simultaneously active.

3.5 Related Research

In recent years, much research has been performed in the area of run-time task mapping for embedded systems. In the context of performance optimization, the authors of [27] propose a run-time mapping strategy that incorporates user behaviour information in the resource allocation process. An agent based distributed application mapping approach for large MPSoCs is presented in [5]. The work of [45] proposes a run-time spatial mapping technique to map streaming applications onto MPSoCs. In [17], dynamic task allocation strategies based on bin-packing algorithms for soft real-time applications are presented. A run-time task allocator is presented in [48] that uses an adaptive task allocation algorithm and adaptive clustering approach for efficient reduction of the communication load. The approach proposed in [106] produces multiple mappings for each application with a trade-off between resource requirement and throughput. Mariani et al. [71] proposed a run-time management framework in which Pareto-fronts with system configuration points for different applications are determined during design-time DSE, after which heuristics are used to dynamically select a proper system configuration at run time. A similar approach is presented in [116] targeting a generic architecture. In [138], the authors propose a lightweight run-time manager, linked with an automated design-time exploration and incorporated in the host processor of the platform, to dynamically and efficiently configure the applications according to the available platform resources. Compared with these algorithms, our task mapping approaches proposed in this chapter take an application scenario-based approach, and take computational and communication behaviour embodied in design-time optimized mappings into account when optimizing the mapping at run time. Recently, Schor et al. [105] also proposed scenario-based run-time mapping approaches in which mappings derived from design-time DSE are stored for run-time mapping decisions. However, [105] does not address the reduction of mapping storage (all workload scenarios are stored) and does not dynamically optimize the mappings at run time.

3.6 Summary

To solve the scalability and flexibility problem of general hybrid task mapping approaches, in this chapter, we proposed several task mapping algorithms that use the information derived from the NSGA-II-based design-time DSE to further optimise the task mapping at run time. The scenario clustering based task mapping approach introduced in the second section uses the STM algorithm to dynamically optimise task mappings for our multi-media application set on a homogeneous MPSoC system at run time. This approach is able to solve the scalability problem with regard to the number of workload scenarios and also apparently improves the system efficiency (performance and energy consumption). Considering both the scalability and flexibility problems, a novel hybrid task mapping approach is proposed for a heterogeneous MPSoC system in the third section using a divide-and-conquer method where the scenario-level task mapping problem is broken down into application-level task mapping problems at design time, and the application-level mapping solutions are then dynamically combined and further optimised to give a complete solution for a workload scenario at run time. Combining the advantages of these two approaches, the HTM approach has been evaluated for a heterogeneous MPSoC system to improve the system efficiency where the EIM algorithm is applied for workload scenario mapping initialisation and the STM algorithm is extended for dynamic QoS management during scenario execution on the target system. For evaluating these approaches, we adapted the Sesame simulator as introduced in the previous chapter and extended it with a run-time resource scheduling framework. The experimental results derived from this extended Sesame simulator confirm the effectiveness of our proposed approaches.

CHAPTER 4

Self-adaptive MPSoC Systems with Adaptivity Throttling

In the previous chapter, we have proposed several hybrid task mapping approaches to solve the scalability and flexibility problem of general hybrid task mapping techniques. However, as mentioned at the start of the previous chapter where the target scenarios on the MPSoCs were assumed to be coarse-grained workload scenarios, the run-time system reconfiguration cost during the process of dynamic task remapping was not explicitly taken into consideration. This assumption is also used in most general hybrid task mapping approaches in which the system manager typically always tries to reconfigure the system resources when a new workload scenario has been detected. In reality, the above assumption might not be applicable as a part of the target scenarios on a MPSoC system could be fine-grained workload scenarios that are only active for a short duration (different workload scenarios rapidly succeed one another at run time). For such fine-grained workload scenarios, the reconfiguration overhead may easily eliminate the benefits of reconfiguring the system: the reconfiguration itself may take longer than the performance gain that is obtained after reconfiguration. Consequently, in the case of fine grained workload scenarios on the target MPSoC system, a blind system reconfiguration may actually degrade the system performance, especially

This chapter is based on:

- W. Quan and A. Pimentel, "A system-level simulation framework for evaluating task migration in mpsocs," in *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '14),* New York, NY, USA: ACM, 2014, pp. 13:1–13:9.
- W. Quan and A. D. Pimentel, "Towards Self-adaptive MPSoC Systems with Adaptivity Throttling," in *Proceedings of the 15th Int. Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS '15),* Samos, Greece, July, 2015. (to appear)
- W. Quan and A. D. Pimentel, "A Run-time Self-Adaptive Resource Allocation Framework for MPSoC Systems," in *Proceedings of the 22nd European Conference on Circuit Theory and Design (ECCTD '15),* Norway, August, 2015. (to appear)
- W. Quan and A. D. Pimentel, "Towards Scalable Scenario-based Run-time Adaptive MPSoC Systems," *Submitted.*

on heterogeneous MPSoC systems. This problem is referred to *blind adaptivity* as introduced in Section 1.4. In this chapter, we focus on solving this problem in hybrid task mapping approaches.

For this purpose, the system reconfiguration cost should be carefully considered for dynamic task remapping at run time. To study the reconfiguration cost during run-time task remapping, in the first section of this chapter, we extend the Sesame simulation framework with the ability of run-time system reconfiguration cost evaluation. This extended Sesame simulator supports a flexible and efficient modeling, simulation and exploration of different system reconfiguration mechanisms and policies in MPSoCs. It is used as the experimental environment for evaluating the work of this chapter.

After that, in the second section, we propose a run-time self-adaptive scheduler/manager to handle the above mentioned *blind adaptivity* problem for MPSoC systems. The smart system scheduler tries to predict whether or not reconfiguration of the system actually is beneficial based on the active workload scenario and the status of the hardware platform. According to this prediction, the system will either be reconfigured or not. By using this technique, which is referred to *adaptivity throttling*, unnecessary system reconfigurations can be avoided, and consequently the system efficiency can be improved.

In the third section, the technique proposed in the second section combined with the novel hybrid task mapping approach from Chapter 3 is used in a run-time self-adaptive resource allocation framework to further improve the efficiency of MPSoCs. By combining these two techniques, the previously discussed issues of general hybrid task mapping approaches: scalability, flexibility and blind adaptivity can be solved. That makes a MPSoC system able to be fully adaptive to the dynamic behaviour of target workload scenarios. After that, a short summary is presented at the end of this chapter.

4.1 A System-level Task Migration Simulation Framework

To fulfil the computation requirements of modern sophisticated applications, MPSoC architectures have in recent years received much attention in the embedded systems domain. As mentioned in the previous chapters, MPSoC systems often require to support an increasing number of applications and standards, where multiple applications may concurrently execute and contend for resources. Consequently, to exploit the full potential and capabilities of such architectures, the mapping of multi-application workloads onto the processing elements of the MPSoC becomes increasingly challenging. This is also due to the fact that the initial task mapping may need to change at run time for different reasons such as the requirements of supporting application dynamism (the change of application execution mode), fault tolerance, load balancing or thermal balancing. For this reason, the concept of task migration has been gaining research attention in the domain of MPSoC design [14]. Task migration is the transfer of the execution of a process (task) from one processing element to another. The concept originates from the massive deployment of distributed systems (and, in particular, distributed operating systems) in the parallel computing domain.

The main idea of task migration is the transfer of task context (state)[1] and its address space between processors [38]. In the domain of MPSoC systems, two main aspects should be carefully considered to support task migration in different architectures, namely *what* and *how* to transfer during migration. Regarding the problem of *what* to transfer during migration, it depends on the architecture of the target system, i.e. homogeneous versus heterogeneous. Different processor types have different ISAs, address widths, register file sizes, etc.. Migration of tasks between heterogeneous cores requires different program codes and task contexts. Even between homogeneous cores, the data that need to be transferred during migration varies among different migration mechanisms. The second problem – *how* to transfer – is related to the organization of the memory (shared and/or distributed) and interconnection (bus, NoC, etc.) of the target system. This determines what kind of communication technique (load/store instructions or message passing) should be used for task migration.

Besides the architecture related task migration mechanism described above, the policies of task migration – determining *when* to migrate task(s), *which* task(s) will be migrated and *where* these tasks will be migrated to – are also very important. Such policies may highly depend on the goal of task migration (fault tolerance, load balancing, thermal balancing, etc.). To design migration-enabled MPSoC systems, a system designer needs to be able to determine what mechanism and policy of task migration are the best choices for the target system already during the early stages of design. To this end, this section presents a system-level simulation framework that supports the flexible and efficient modeling, simulation and exploration of different task migration mechanisms and policies in MPSoCs. Using a number of experiments, in which we study task migration in both shared-memory and message-passing MPSoC architectures, we also demonstrate the flexibility and capabilities of our simulation framework.

4.1.1 Task Migration Mechanisms for Different Architectures

In this subsection, we will introduce several well-known task migration mechanisms for different hardware architectures. Here, system architectures can be divided into three categories according to the system memory organization: Uniform Memory Access (UMA) [50], Non-Uniform Memory Access (NUMA) [67] and NO Remote Memory Access (NORMA) [14, 38].

In a UMA system architecture, all the processors uniformly share the physical memory. The cost of accessing the memory is the same for all the processors in the system. A typical example of this architecture are the tightly coupled shared memory SMP (Symmetric Multi-Processor) systems, where all the processors run a single copy of an operating system that coordinates global activities. In SMP systems, task migration only needs to transfer the task's context between processors, while the address space of the migrating task does not need to be transferred since it is located in the shared memory that is shared by all processors. In this

[1]The context of a task or a process is the minimal set of data used by this task/process that must be saved to allow either task interruption and/or migration at a given instant, and a continuation of this task at the point it has been interrupted/stopped and at an arbitrary future instant.

case, the cost of task migration is relatively low compared to the other multiprocessor architectures. In contrast to UMA, the memory access time in a NUMA architecture depends on the memory location. A processor can access its own local memory faster than non-local memory (e.g., memory local to another processor). In this kind of architecture, besides the task's context, the address space contents of the migrating task may also need to be transferred between different memories. Clearly, such task migrations come at the cost of a performance penalty and increased on-chip communication.

In the two previous types of architectures, processors share a single address space (i.e., the memory is physically/logically shared among processors) and therefore the data transfer of task migrations can be done via *load* and *store* instructions. This is, however, not possible for NORMA architectures where processors have a private memory (and address space) and do not grant access to their memory by other processors. In NORMA architectures, the task migration must therefore be coordinated using messages (i.e., message-passing) between processors. As a consequence, the cost of task migration in NORMA architectures typically can be high due to the need of transferring the migrating data over relatively slow, multi-hop communication channels like in a NoC.

For scalability reasons, future MPSoC systems will increasingly be equipped with distributed memory, i.e., either use NUMA or NORMA architectures. This means that the impact of task migration is not negligible with respect to system performance. To reduce the task migration cost, several task migration mechanisms like the *eager-copy*, *pre-copy* and *post-copy* techniques [9, 101] have been proposed. For heterogeneous architectures, as different processor types require different program code and task contexts, the complexity of task migration is much higher than for their homogeneous counterparts. State-of-the-art systems solve this through checkpointing [19, 77, 84] and application-level save-restore mechanisms [19], while there exists no mechanism that is fully transparent to applications [52].

4.1.2 Task Migration Supports in Sesame

For the purpose of studying and evaluating the impact of different task migration schemes on the overall performance of a MPSoC system, our Sesame simulator is again deployed. However, in the original Sesame simulator, applications are directly mapped onto the hardware platform via a mapping/virtual layer and cannot be changed dynamically during simulation, which limits the study of task migration mechanisms. To support the modeling and simulation of task migration in Sesame, we have modified its structure to resolve this constraint. More specifically, a new middleware layer which is in charge of run-time task migration has been added to the Sesame framework. This middleware layer, called the Run-time Task Migration Middleware (RTMM), removes the direct mapping relationship between applications and hardware resources, as shown in Figure 4.1. To coordinate task migration-related activities at the architecture level on behalf of the RTMM, a Run-time Resource Scheduler (RRS) module is provided in the architecture model layer. This RRS manages the hardware resources either in a centralized or distributed (in large-scale systems) manner. It takes care of collecting statistics (e.g., performance of each application, system execution

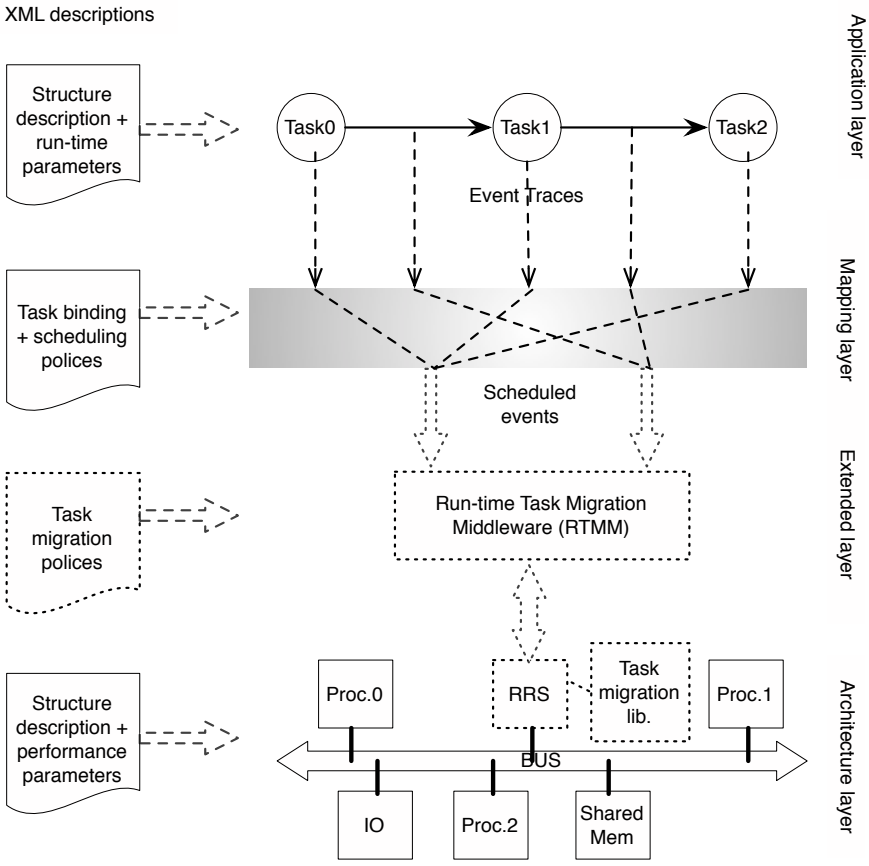


Figure 4.1: Extended Sesame's layered infrastructure

information, etc.) from the underlying system during the simulation process, triggering the RTMM layer to make a decision when task migration is needed and sending migration commands to specific processors based on the decision of the RTMM. To facilitate the simulation of different task migration mechanisms, we provide a task migration library that implements several *migration micro instructions* as shown in Table 4.1. Using these micro instructions, different migration mechanisms can easily be modelled. The first two instructions are designed for systems that use shared memory whereas the other two instructions are used for message-passing systems. In these instructions, the parameter *mig_mode* is used to indicate different migration schemes like migrating task code only, task context only or both. Figure 4.2 illustrates how the micro instructions can be used in the RRS to support task migration for a system with shared memory. After having received the new mapping scheme calculated by the RTMM, the RRS will start the task migration process for the task(s) that will be migrated by issuing the micro instructions according to the migration mechanism implemented in the architecture. Here, the MIG_STORE triggers the storing of all migrating data into shared memory, while the MIG_LOAD triggers the loading of this data at the destination processor. For message-passing systems, the MIG_SEND and MIG_RECEIVE micro instructions will be sent to the processors from/to which a task is migrated, which will initiate a message-passing data transfer (realizing the actual migration) between these two processors.

Table 4.1: Micro instructions provided in the task migration library

Instruction	Parameters
MIG_STORE	old_pe, address_shmem, mig_mode, mig_datasize
MIG_LOAD	new_pe, address_shmem, mig_mode, mig_datasize
MIG_SEND	old_pe, new_pe, mig_mode, mig_datasize
MIG_RECEIVE	new_pe, old_pe, mig_mode, mig_datasize

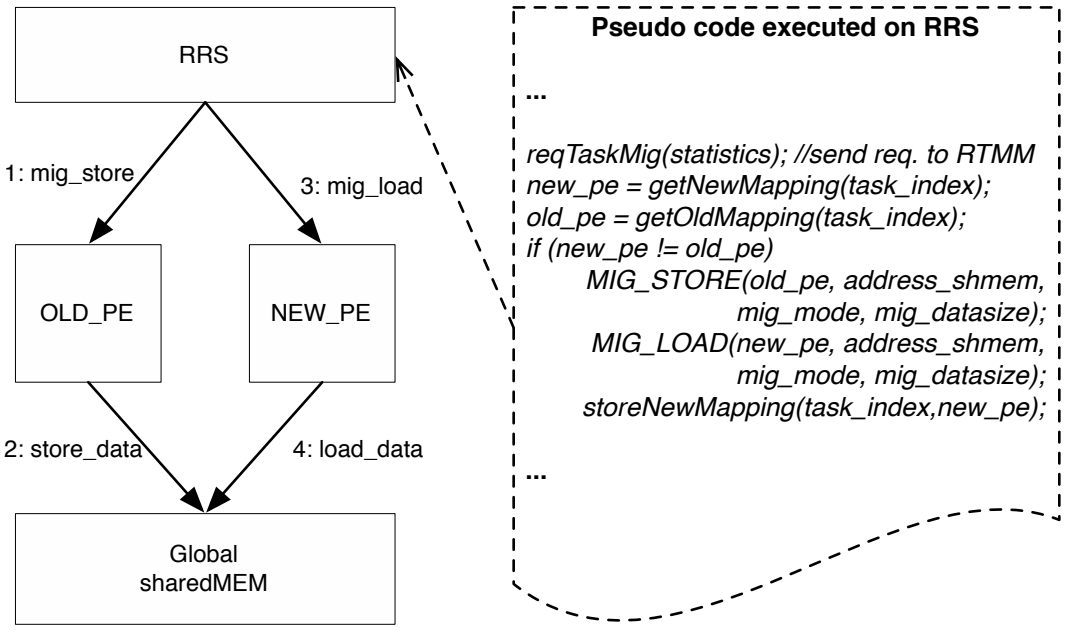


Figure 4.2: A simple example of task migration implementation on a shared memory system

As mentioned before, task migration can be implemented for different purposes such as a violation of application performance constraints, load balancing, fault tolerance and so on. To trigger task migrations, our framework supports different types of approaches that can be implemented in different layers, ranging from the application level to the architecture level. For example, at application level, explicit migration check-points can be inserted in the application code. In this case, the task migrations are controlled by the application designer. At the architecture level, each processor could also issue task migration requests to the RRS, triggered by e.g. the detection of undesired (execution) behaviour like a timing violation, hardware fault or overheating. Beside these, the RRS can also trigger a task migration based on the statistics it has collected. In our framework, the task migration process is performed by means of coordinated actions between the RTMM and RRS. The exact responsibilities and workflow for each of these two components is shown in Figure 4.3. At run time, the RRS will continuously monitor the execution of applications and collect the running statistics of the tar-

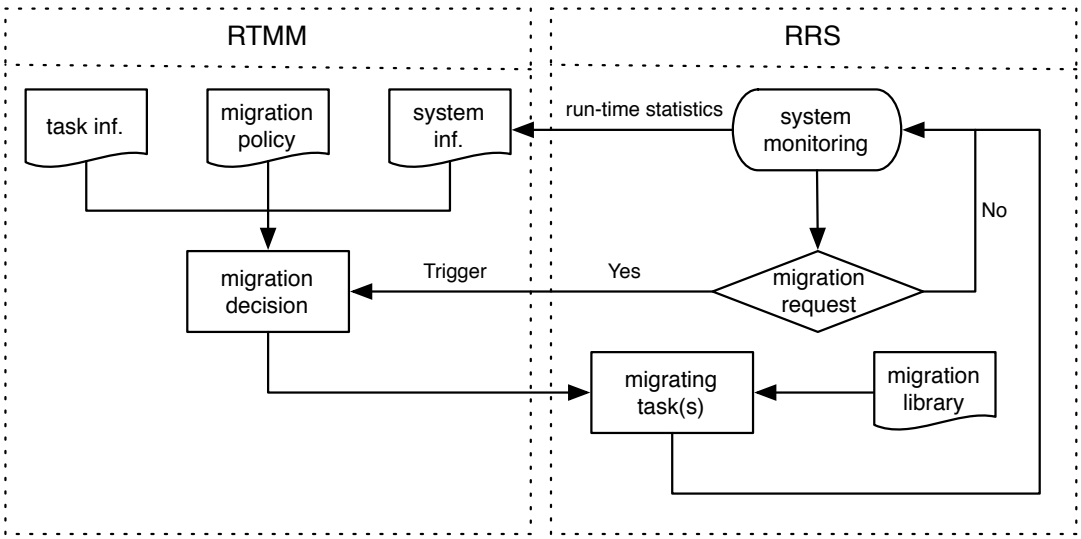


Figure 4.3: The task migration workflow of our framework

get system. Whenever there is a pre-defined migration condition detected (e.g., a performance deadline violation), the RRS will send a task migration request to the RTMM. Currently, our simulation framework will stall all application processes until the migration decision has been taken. The RTMM, after receiving a migration request from the RRS, will make a migration decision based on the task migration policy implemented, information of active tasks and the execution statistics collected by the RRS. Here, the migration decision includes the information of which task(s) should be migrated and where the task(s) should be migrated to. It does not involve information about how the task will be migrated. This is under the control of the RRS. Given this task migration decision, the RRS will start the task migration events according to the migration mechanism that has been implemented using the migration micro instructions. Notice that, the actual migration of a task can only occur when the task is in a pre-defined migratable state, e.g. frame boundaries for our multi-media applications.

Using the extended Sesame simulator, it is possible to evaluate the impact of task migrations on system performance for different MPSoC architectures. To achieve this flexibility, the following support for modeling and simulating different task migration-enabled architectures is available at each layer.

- **Application model layer:** In our Sesame simulator, applications are modeled using KPN computation model which can be implemented in any high level programming language. To emit events to the architectural model, the application processes are annotated. By default, the Sesame framework supports the generation of read, write and execute events as mentioned in Section 2.1.1. To support the study of application-level task migration mechanisms, we have added the possibility to instrument the code of processes such that special task migration events can be generated, which trigger task migrations in the RTMM. Besides the (instrumented) application source code, a structural application description (using an XML-based language [30]) is provided to the simulator. This description includes the topology of the

processes in the target application(s) and the execution parameters of each process.

- **Mapping layer:** This layer determines the mapping of processes (i.e. their event traces) onto architecture model components by dispatching application events to the correct architecture model component, as can be seen in Figure 4.1. It also includes the mapping of communications at application level onto communication resources in the architecture model. The mapping layer has two additional purposes. First, the event dispatch mechanism in the mapping layer provides a variety of static and dynamic policies to schedule application tasks (i.e., their event traces) that are mapped onto shared architecture model components. Second, the mapping layer is also capable of dynamically transforming application events into (lower-level) architecture events to facilitate flexible refinement of architecture models as introduced in Section 2.1.4. In the extended Sesame simulator, the dispatching of trace events to architecture components is now controlled together with the RTMM / RRS tandem. The RTMM forwards events from the mapping layer to the RRS in the architecture model layer, where the latter is in charge of actually dispatching the trace events to the processor component onto which the application task in question is currently mapped. The mapping description that acts as input for this mapping layer includes the task migration related parameters like the minimal task context size and compiled task code size.
- **RTMM layer:** In this layer, the migration policy (algorithm) is implemented based on the goal of task migration. The policy defines *which* task(s) should be migrated and *where* the task(s) should be migrated to. The designer can implement different policies like the task remapping algorithms proposed in Chapter 3 to test which one is the best for the design goal at hand.
- **Architecture model layer:** This layer models the (non-functional behaviour of the) MPSoC hardware architecture, and can be generated using a system library that provides the template models for different components like processors, memories, communication channels and interconnects, etc. Also, designers can add and customise their own system components. To support task migration, the RRS component should be integrated into the architecture model. We also provide a template RRS implementation. In this template, one could use the micro instructions as described before to support different task migration schemes based on the target system architecture. Besides the architecture model, an architecture description file should be provided. It describes the structure of the architecture and includes the non-functional properties of hardware components like frequency, power, bandwidth/latency of communication channels, and so on.

4.1.3 Task Migration Case Studies

In this section, we present two case studies in which we model task migrations in two very different systems, shown in Figure 4.4, to demonstrate the flexibility and wide application scope of our simulation framework. The target applications

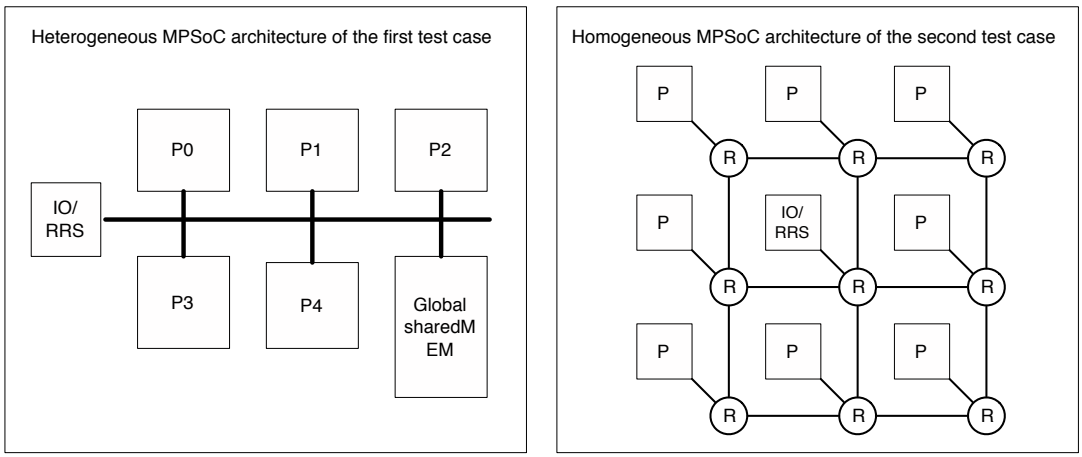


Figure 4.4: The MPSoC architectures used for the case studies

Table 4.2: The parameters for application and architecture description in the simulator

Parameters	**Explanation**
T_i^j	execution cycles of task i on processor j
CS_i^j	code size of task i on processor j
TC_i^j	minimal task context size of task i on processor j
S_m, S_b	size of memory m or buffer b in the system
B_m, B_c	bandwidth of memory m or comm. channel c
L_m, L_c	latency of memory m or comm. channel c
F_k	frequency of hardware component k on the system

used in our experiments belong to the multi-media application domain. Each application has a (soft) real-time performance constraint which can be used to trigger task migrations as shown in the first experiment. For the application and system architecture description, the parameters needed for simulation are listed in Table 4.2. If needed, these parameters can be calibrated by designers using low(er)-level simulators or measurements on real systems. We would like to stress that this section does not focus on the actual assessment of state-of-the-art task migration policies, but instead our aim is to demonstrate the flexibility and wide application scope of our simulation framework. Therefore, in the two case studies, we have chosen to implement only relatively simple task migration policies in the RTMM. The details of each of the two experimental cases will be explained in the following subsections.

4.1.3.1 A Heterogeneous UMA MPSoC

Target Application and System Architecture In this experiment, the target application is the Sobel filter for edge detection in images (frames) as introduced in the previous chapter. The target MPSoC system is shown in the left part

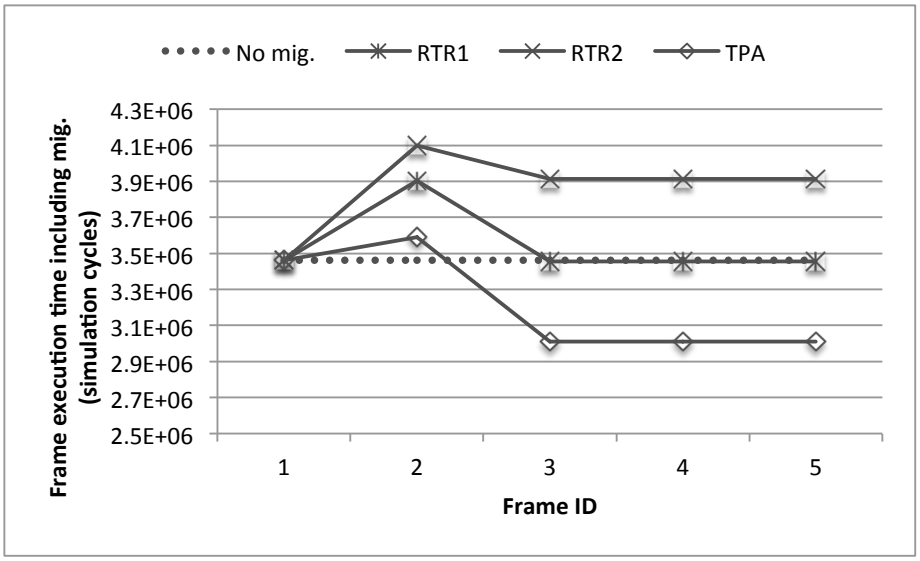


Figure 4.5: Task migration impact on application performance for the heterogeneous MPSoC

of Figure 4.4. In the heterogeneous MPSoC, 5 processors with different architectures are connected by a bus. A global memory and a IO processor are shared by these processors. The RRS has been integrated in the IO processor. The IO processor can collect application performance statistics like Frame Execution Time (FET) at the end of each processed frame (i.e., the time between a frame is read and written by the IO processor). Based on these statistics, the RRS can trigger a task migration event when needed. Initially, all processes except the two IO processes in the Sobel application are mapped onto processor *p0* of the heterogeneous MPSoC (the IO processes are mapped onto the IO processor).

Migration Mechanism and Policy As the target architecture in this experiment is a heterogeneous MPSoC with shared memory, the binary code of each task for each processor might be different. Here, we assume that the compiled code of each task for each processor is preserved in the global shared memory. Under this assumption, we have modeled a *task recreation* mechanism [38] in this experiment to support task migration. During task migration, the migrating task will be killed on the original processor and the task state information will be saved in global memory. The destination processor will load the binary code and state information from shared memory to restore the task. Also, the communication channels connected with the migrating task will be redirected to the new processor by remapping them.

Regarding the task migration policy in this experiment, we have modeled the following two algorithms to make the task migration decision: a Random Task Remapping (RTR) which generates a random mapping for task remapping[2], and a Task Processor Affinity (TPA) [83] as introduced in Section 3.3.3.

[2]RTR randomly decides which task(s) should be migrated and where the migrating task(s) should be migrated to.

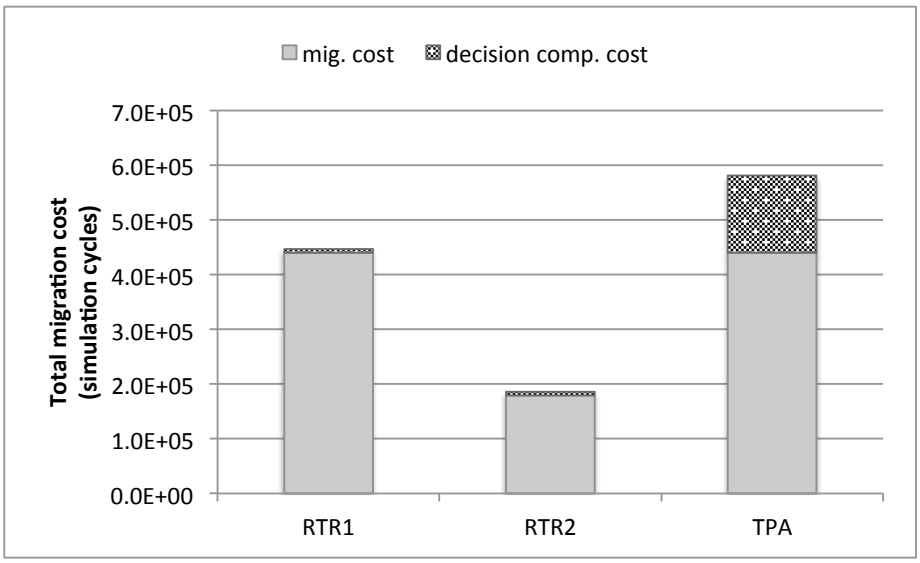


Figure 4.6: Task migration cost of different algorithms on the heterogeneous MPSoC

Experimental Results In this experiment, we use a single picture as the continuous input stream of frames for the Sobel application. The migration trigger in this experiment is a violation of the application performance constraint. To this end, we have set a performance constraint for Sobel, using the Frame Execution Time (FET) metric, such that it enforces a task migration request after the first frame has been processed. The migration request is handled by the RTMM, which subsequently applies the implemented migration policy to make a task migration decision. During this process, all the tasks of the Sobel application will stall and wait for the task migration decision. After the migration has finished, the system continues to process the subsequent frames and monitor the execution of the application.

Figure 4.5 shows the experimental results of the migration impact to the application performance by using different task migration policies (algorithms) for our heterogeneous MPSoC. The marked lines labeled as *RTR1* and *RTR2* are the results of migrating tasks based on two different migration decisions computed by the RTR algorithm. The dotted line represents the execution using the initial mapping and without task migration. In Figure 4.6, we also show the task migration overhead for each policy, which includes two main elements: the computational cost of the task migration decision and the task migration cost itself. The computational cost of the task migration decision has been measured on a real CPU and then normalised to the simulation frequency of our simulator. In Figure 4.5, we clearly notice the task migration taking place after the first frame since the higher FET values for the second frame include the task migration overhead. After the second frame, the FET values stabilize again, i.e., no further task migrations are triggered. From the results, we can also see that the migration cost of *TPA* is the highest among three task migration scenarios. Here, *RTR2* has migrated 2 tasks, whereas *RTR1* and *TPA* both migrated 4 tasks. Consequently, *RTR2* has the smallest task migration overhead overall. However, after migration, the resulting mapping as derived by *TPA* clearly shows the best performance.

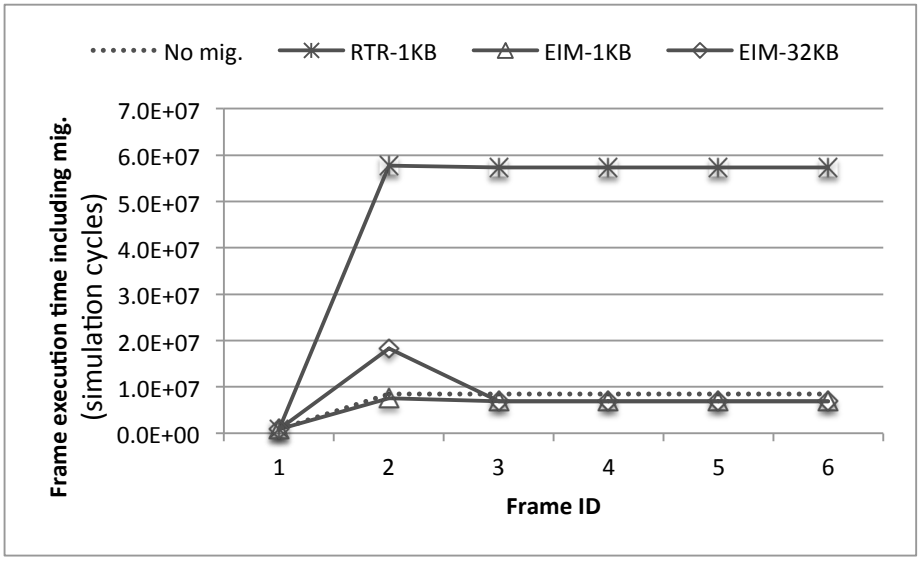


Figure 4.7: Task migration impact to application performance on the homogeneous MPSoC

4.1.3.2 A Homogeneous NORMA MPSoC

Target Application and System Architecture The applications used in this experiment are the three multi-media applications: MJPEG, MP3 and Sobel as introduced in the experiments of previous chapter.

With respect to the target system in this experiment, the architecture is shown in the right part of Figure 4.4. In this system, 8 homogeneous processors and an IO processor are connected by a 2D mesh NoC. Similar to the system described in the last experiment, the RRS has also been integrated into the IO processor. Initially, we again map all processes except the IO processes onto processor *p0*.

Migration Mechanism and Policy In this experiment, the target system architecture is a homogeneous MPSoC system with private memories connected to the processors (i.e., no remote memory access). As all processors have the same architecture, a task's context can be shared among processors. Therefore, we have modeled a *task replication* mechanism [38] to support task migration in this system. The idea is that each processor on the system has a replica of all tasks. Only one copy of a task can be active and running on a specific processor while the other copies are suspended and reside in memory of the other processors. When a task migration is needed, the task is suspended in the source processor and resumed in the destination processor, using the context of the migrating task. Also, the communication channels connected to the migrating task will be redirected to the new processor by the RRS. So, using this migration mechanism, only the context of the migrating task needs to be migrated between processors. This greatly reduces the communication overhead of task migration at the cost of increased memory usage because of the storage of task copies.

The task migration policies used in this experiment are slightly different than in the previous experiment. Since the target architecture is a homogeneous MPSoC, each task has the same task execution time on each processor on the system.

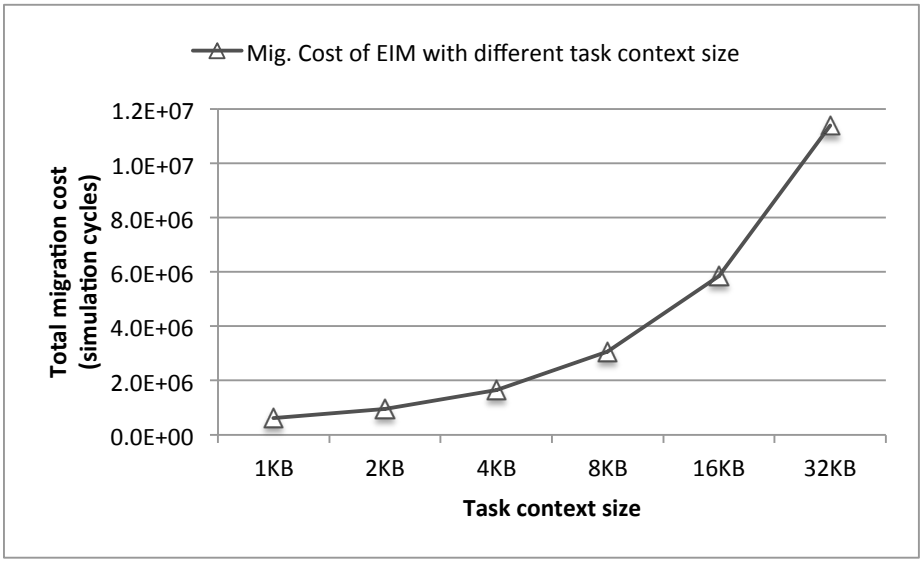


Figure 4.8: Task migration cost of algorithm EIM with different task context size on the homogeneous MPSoC

Consequently, the TPA algorithm would not be very effective in this case. As a substitute, we have modeled the EIM algorithm from Section 3.3 without considering its energy constraint.

Experimental Results Figure 4.7 shows the results of the migration impact to the application performance for our homogeneous MPSoC. In this experiment, the three afore-mentioned applications are mapped onto the target system. For the purpose of results comparison, we also use the concept of *frame* to define the workload of applications. Here, we combine one unit workload (e.g., one picture) of each application together as one frame for all applications. In this case, the frame execution time of multiple applications is defined as the maximal frame execution time among applications (each application processes its own unit of workload). For example, the frame execution time F of our three target applications for processing one frame workload is represented as Equation 4.1.

$$F = max(F_{mjpeg}, F_{sobel}, F_{mp3}) \tag{4.1}$$

where F_{mjpeg}, F_{sobel} represents the frame execution time of processing one picture for MJPEG, Sobel respectively and F_{mp3} means the execution time of processing one short piece of encoded MP3 song.

Similar to the previous experiment, a single input frame will be reused continuously to act as input stream for each application. In this experiment, task migration is used for dynamic resource reallocation, and therefore task migration is triggered when the system workload changes. For example, a new application enters the system or an active application finishes and quits the system. In Figure 4.7, there is only a single application (MP3) active in the system during Frame ID '1'. At the end of this frame, two other applications, namely Sobel and MJPEG, are activated on the system and execute from frame '2' until '6'. This means that a task migration is triggered during Frame ID '2', as is clearly

visible in Figure 4.7. This figure shows the results for both the *RTR* and *EIM* migration decision algorithms. For EIM, the migration impact with different task context sizes have been considered (a 1KB and a 32KB context). From the results shown in Figure 4.7, we can see that the applications show poor performance after task migration when applying *RTR*. This can be explained by the fact that the communication costs of the NoC are high, especially for the applications that are communication intensive. In our case, it is better to map the MJPEG and Sobel applications each onto a single processor to reduce the communication cost. However, the RTR algorithm will generate mapping decisions without considering the communication cost at all. On the other hand, by applying the *EIM* algorithm, the final mapping has good performance behaviour. Moreover, comparing the *EIM-1KB* and *EIM-32KB* curves, it can be seen that the application performance during task migration (Frame ID '2') can be substantially influenced by the task context size. To study how the task migration overhead is affected by the size of the migrating data in more detail, we have measured the migration overhead for different task context sizes. The results of this experiment are given in Figure 4.8. Clearly, the cost of transferring task context on our target homogeneous system linearly increases with the size of task context. However, in Figure 4.8, the task migration overhead also includes the computation cost of the EIM algorithm. From this figure, we can see that the task migration cost increases slowly with the task context size when it is under 8KB. However, when the task context size is bigger than 8KB, the task migration cost has a near linear relationship with the task context size. The reason is that when the task context size is small (like below 8KB in our test case) the task migration cost is dominated by the computation cost of the EIM algorithm. However, when the task context size increases to a certain amount, the cost of transferring task context dominates the task migration cost.

4.1.4 Related Research

Task migration has been traditionally studied in distributed systems for dynamic load balancing [117, 65, 24]. However, with the increasing popularity of MPSoCs in modern embedded systems, task migration has also gained research attention in this domain and has been studied for different purposes. For the purpose of thermal balancing, Cuesta et al. [33] provide three task migration policies to optimize the thermal profile of MPSoCs by dynamically balancing the weight of the on-chip thermal gradients, maximum temperature and effect of the underlying floorplan on heat dissipation properties of each core. In the work of [81], task migration-based thermal balancing policies are proposed to modulate power distribution between processing cores to achieve temperature flattening. The authors in [41] use proactive task migration among neighboring cores to balance the thermal profile for many-core systems. For the purpose of load balancing, in [14], task migration combined with intelligent initial placement are used to maintain load balancing in the MPSoC system. The authors in [18] analyze the impact of task migration in a NoC based MPSoC system. In their work, task migration is triggered after the resource allocation heuristic which tries to balance the system on demand is executed. To support fault tolerance on MPSoC systems, task migration is also a required technique [20, 75, 28]. The idea in [20, 75] is to improve dependability of

the system by exploiting the migration method in case of run-time faults in the processing cores. In [28], a system-level fault-tolerance technique for application mapping, which aims at optimizing the entire system performance and communication energy consumption, is proposed. To this end, application components running on a faulty core are migrated altogether to available non-employed spare cores.

In the context of task migration mechanisms supported in MPSoC systems, quite some work has been done on task migration at application level, middleware level and architecture level. In [14], the authors propose a user-managed migration scheme based on code checkpointing and user-level middleware support as an effective solution for many MPSoC application domains. The work in [1, 100] implements task migration in a middleware layer which is built on top of the uClinux operating system running on a prototype multicore emulation platform. To support heterogeneity in task migration, [89] provides a middleware, called Low Level Virtual Machine (LLVM), to postcompile the tasks at runtime depending on their target processor. At the architecture level, [2] discuss the possible architectural support for MPSoC systems to allow dynamic task migration. In [12], the authors propose a hybrid memory organization for NoC-based MPSoC systems as the way to minimize the energy spent during the code transfer when task migration or dynamic task allocation needs to be performed.

With regard to task migration simulation, in [113, 111, 112], the authors extended Sesame to have a system-level simulator for run-time task mapping in the context of reconfigurable systems. In their simulator, tasks can be migrated between a general purpose processor and a reconfigurable accelerator which enables the system to be more efficient in terms of various design constraints such as performance, chip area and power consumption. However, no details of the task migration implementation are shown in this work. Different with this simulator, our proposed simulator provides a general purpose task migration framework which is not limited by the target architecture, the task migration purpose and task migration mechanism. To the best of our knowledge, this section presents the first effort in the direction of establishing a generic simulation infrastructure that allows for the efficient exploration of a wide range of migration mechanisms and policies in MPSoCs.

4.1.5 Conclusion

Task migration is a useful technique that can be used in MPSoC systems for different purposes such as load balancing, thermal balancing, fault tolerance, improving system energy efficiency and so on. Therefore, investigating the suitability of specific task migration schemes for a target system architecture is an important design step that needs to be addressed as soon as the early stages of design. For this purpose, in this section, we have presented a high-level simulation framework that allows for simulating and exploring different task migration mechanisms and policies for a wide range of different system architectures. Using two case studies, we have demonstrated the flexibility and wide application scope of our simulator. To this end, the case studies evaluate different task migration policies and mechanisms for vastly different target architectures. The experiments point out that our task migration simulator can provide designers with useful insights on

the suitability of a specific migration scheme for the target system and allows for exploring different migration policies.

4.2 Dynamic Task Mapping with Adaptivity Throttling

To cope with run-time dynamic application behaviour, MPSoC systems could dynamically adapt the mapping of application tasks onto the underlying system resources to improve the system's performance. However, such performance improvement comes at the cost of a system reconfiguration in which application tasks may have to be migrated between processors. This trade-off implies that reconfiguring the system is only beneficial when the performance gains outweigh the reconfiguration overhead. To address this problem for MPSoCs, this section presents a scenario-based run-time resource management framework with the ability of adaptivity throttling that uses the history of application scenario execution behaviour to predict the actual benefit of a system reconfiguration to allow for explicitly deciding (at runtime) whether or not to reconfigure.

4.2.1 Objective Formulation

Our target applications belong to the domain of streaming applications (like multimedia applications) that continuously process an incoming stream of data elements. To capture the duration of a workload scenario in this case, we use the concept of *scenario frames*. Here, we define one *scenario frame* as the time it takes for each active application within a specific workload scenario to process a single unit (frame) of data (e.g., processing a single MP3 frame, an H264 frame, etc.). This means that the frame execution time p_i^j of a workload scenario s_i under mapping tm_i^j is defined as the maximum of frame execution times of all active applications within the scenario:

$$p_i^j = max(p_{app_k}^{ij}) \tag{4.2}$$

where $p_{app_k}^{ij}$ represents the frame execution time of application app_k that is active in scenario s_i under mapping tm_i^j. Consequently, the total execution duration of scenario s_i under mapping tm_i^j is calculated as $p_i^j * n_i$ where n_i is the number of scenario frames executed on the system.

When a new workload scenario has been detected, this means that one or more applications may have stopped and/or new ones have started. Here, we assume that when a new application is started, it is added to the system using a predetermined, default task mapping. Given a newly detected scenario, the complete task mapping of those applications that persist in the new scenario and any newly added applications needs to be reconsidered. Remapping of the application tasks in the newly detected workload scenario can be beneficial performance wise (i.e., every workload scenario has an optimal task mapping) but this depends on both the actual performance gain of reconfiguring the system and the reconfiguration costs. The reconfiguration costs include two parts: 1) the overhead of the resource scheduler which includes the time of finding a new mapping and making a reconfiguration decision, and 2) the task migration cost that may occur during system reconfiguration. Here, we denote the reconfiguration costs for scenario s_i

to change from mapping tm_i^j to $tm_i^{j'}$ as $c_i^{jj'}$, which includes the time of finding the new mapping $tm_i^{j'}$ and making the reconfiguration decision for the new mapping. This implies that the system reconfiguration benefit B can now be expressed as:

$$B = (p_i^j - p_i^{j'}) * n_i - c_i^{jj'} \tag{4.3}$$

Our objective is to maximize the system performance for a sequence of workload scenarios S'. This means that we want to maximize the total system reconfiguration benefit $\sum_{s_k \in S'} b_k * B_k$, where $b_k \in \{0, 1\}$ is the migration decision made by the run time scheduler and B_k is the reconfiguration benefit of workload scenario $s_k \in S'$. Obviously, the solution to this problem is $b_k = 0$ if $B_k <= 0$ and $b_k = 1$ if $B_k > 0$. Consequently, to achieve our objective, the system needs to correctly predict the system reconfiguration benefit B for each workload scenario.

4.2.2 Task Mapping with Adaptivity Throttling

4.2.2.1 Design-time Preparation

To achieve the objective of run-time adaptivity for a MPSoC system, in our proposed approach where a general hybrid task mapping technique is deployed, a design-time preparation stage is needed to provide the necessary information for the run-time manager. For each possible workload scenario, the optimal mapping needs to be found at design time. To this end, we have deployed the scenario-based DSE approach proposed in Section 2.3 in those cases where the mapping problem is intractable for exhaustive DSE[3]. The design-time DSE yields a performance optimized mapping for each workload scenario, which are stored in system memory for run-time usage.

To determine the performance gain of reconfiguring the system (remapping application tasks) versus keeping the currently active mapping in case of a newly detected workload scenario, we must know the performance of both the current mapping and the pre-optimized mapping of the newly detected scenario. In the work of this section, we assume that these two performance numbers have also been determined at design time and are stored in system memory. This means that the frame execution time of each workload scenario under its pre-optimized mapping (as found by design-time DSE) is stored. Moreover, we assume that we also have a look-up table (generated at design time) to determine the performance of a newly detected scenario under the old mapping of the previously active application scenario. Evidently, the memory consumption of this look-up table will be problematic when the number of target applications is large. This problem will be solved in the next section by using a light-weight run-time mapping performance predictor. Finally, we also statically store the size of data that needs to be migrated during task migration for every task in all target applications for the run-time prediction of migration costs.

[3]Exhaustive DSE: exhaustively explore every possible mapping in the mapping space to derive the optimal solution for the target optimization goal. We use it for small-scale task mapping problems to derive the best solution at design time.

4.2.2.2 Run-time Resource Reconfiguration

To construct the adaptive resource scheduler as introduced before, the problem that needs to be solved is to correctly predict the system reconfiguration benefit B. It consists of three parts: the performance improvement $p_i^j - p_i^{j'}$, the reconfiguration cost $c_i^{jj'}$ and the workload execution duration n_i. These three parts are unknown before the system reconfiguration. Thus, prediction models should be used to determine a reconfiguration decision based on the benefit B. In this section, we mainly focus on the prediction of the workload scenario duration n_i.

Mapping Performance Prediction As explained in the previous section, the performance of each workload scenario under the mappings derived at design time are stored on the system. In this case, the run-time scheduler does not need to dynamically predict the performance of workload scenarios which saves additional overhead, at the cost of additional design-time preparation and the system storage needed for storing the performance information.

Reconfiguration Cost Prediction The reconfiguration cost of our target MPSoC consists of two parts: the overhead of the resource scheduler and the task migration cost during system reconfiguration. When a new workload scenario is detected, the system scheduler will first determine a new mapping (in our case the stored pre-optimized mapping) for this workload scenario, and then make a reconfiguration decision. The overhead of these two steps can be determined by means of measurements. However, the other part of the reconfiguration cost, the cost of task migration, should still be predicted. As mentioned before, the amount of data that needs to be migrated between processors for each task is known at design time. Subsequently, we use a simple linear analytic model for the migration cost:

$$CMig = (\sum_{t_i \in T_{mig}} ms_i)/r_{mem} \tag{4.4}$$

where T_{mig} is the set of migrating tasks, ms_i represents the amount of migrating data for task t_i, and r_{mem} is the memory access speed. This model is based on two assumptions: the migrating data is transferred via the MPSoC's shared memory and the resource scheduler sequentially controls task migrations.

Reconfiguration Decision Prediction For a newly detected workload scenario s_i, the potential performance improvement due to system reconfiguration can be calculated by using the stored mapping performance information that was derived at design time. Combining the performance improvement and the (predicted) system reconfiguration cost, the scheduler can determine a lower bound bn_i for the execution duration of this workload scenario:

$$bn_i = c_i^{jj'}/(p_i^j - p_i^{j'}) \tag{4.5}$$

According to the derivation in Section 4.2.1, one can easily see that the system should be reconfigured only if the execution duration of s_i is larger than bn_i ($B > 0$). In this section, we propose an Accumulated Statistical Metric Model (ASMM), which is based on the Statistical Metric Model (SMM) [103], to predict

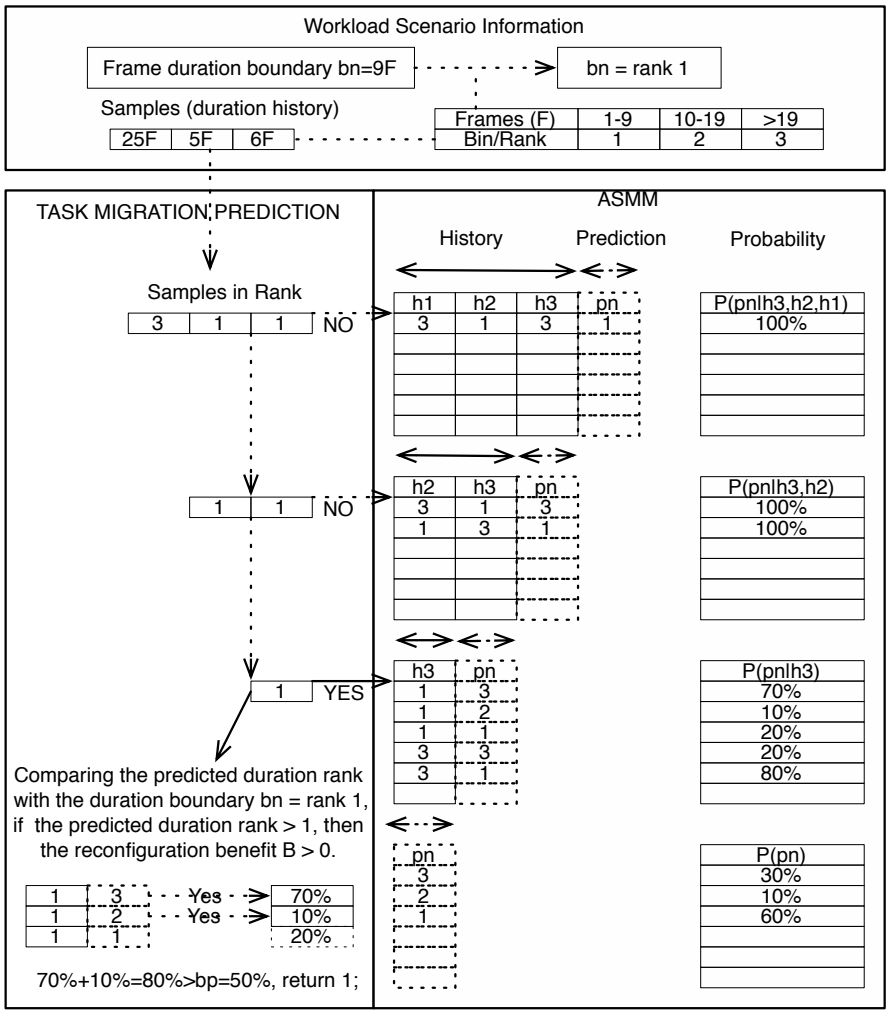


Figure 4.9: ASMM example with 3 history samples.

the scenario execution duration. The SMM is a probability distribution over application patterns of varying length. It models the conditional distribution on the identity of the *ith* (quantized) sample given the identities of all previous (quantized) samples in a metric sequence. The difference between our ASMM and the original SMM concerns the way of how the prediction value is generated, as will be explained below.

Using our ASMM, we build a metric model based on the probability distribution of scenario execution duration for each workload scenario, which means each workload scenario has its own ASMM. When a new workload scenario is detected, the system scheduler uses the ASMM of this workload scenario to predict its duration. Figure 4.9 gives an example of the ASMM-based prediction of a reconfiguration decision when using 3 history samples. In our problem, the samples of scenario execution duration are measured in the number of *frames* (F). These frame numbers are quantized using a limited number of bins (or *ranks*) to reduce the complexity of our predictor, see the upper part of Figure 4.9 for an example. The lower right part of Figure 4.9 shows a simple instance of our ASMM. It includes three kinds of tables: the execution duration history tables, duration prediction tables and tables with probabilities for all possible duration predictions.

The first two types of tables store *rank numbers*. The width and depth of the history tables usually determine the prediction ability of the ASMM and should be set based on the target problem. At the end of a workload scenario, all tables of the workload scenario are updated according to its actual execution duration. We refer the interested reader to [103] for further details about (A)SMMs.

To illustrate the ASMM-based reconfiguration decision, please consider the lower left part of Figure 4.9. The input of our ASMM is the (quantized) execution duration sample history (top of Figure 4.9) of the detected scenario and the duration bound bn_i. According to the detected workload scenario, the corresponding ASMM will be used to determine the reconfiguration decision. In our example, the ASMM first checks the history table with 3 history samples to see if there is a pattern match regarding the scenario's duration history. If there is no match, like in the case of our example, the ASMM will continue to search the history tables with a smaller width of history samples (i.e., using a shorter history). This process continues until there is a history pattern match or it ends up at the direct duration prediction without any execution duration history (the table at the bottom of the ASMM in Figure 4.9). In both cases, the duration bound bn_i is compared with all the possible duration prediction values and for those prediction values bigger than bn_i their probabilities will be accumulated: hence the name *Accumulated* SMM. This accumulated probability represents the chance of $B > 0$ if the system is reconfigured for this workload scenario. Only if the accumulated probability is larger than the probability bound bp (set by the designer), the ASMM will return a positive reconfiguration decision. In our example, the probability bound bp is set to 50%.

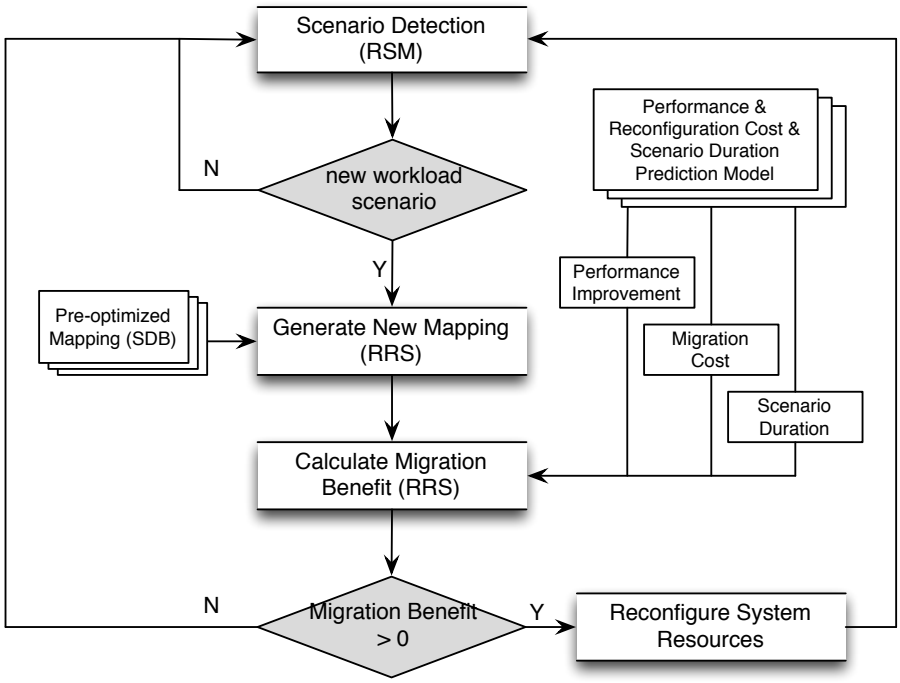
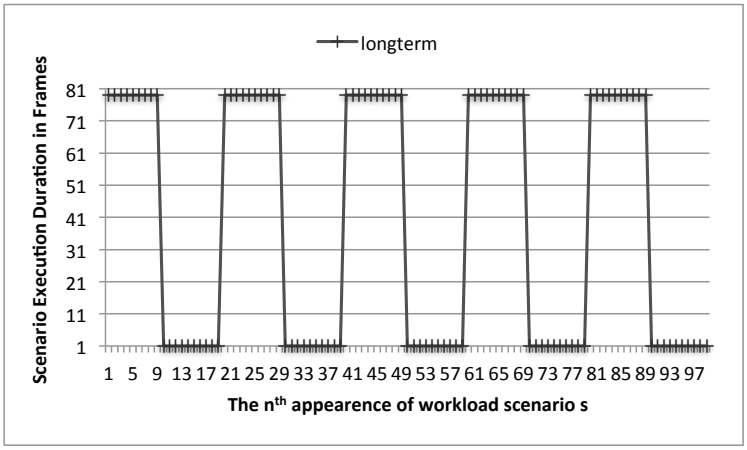
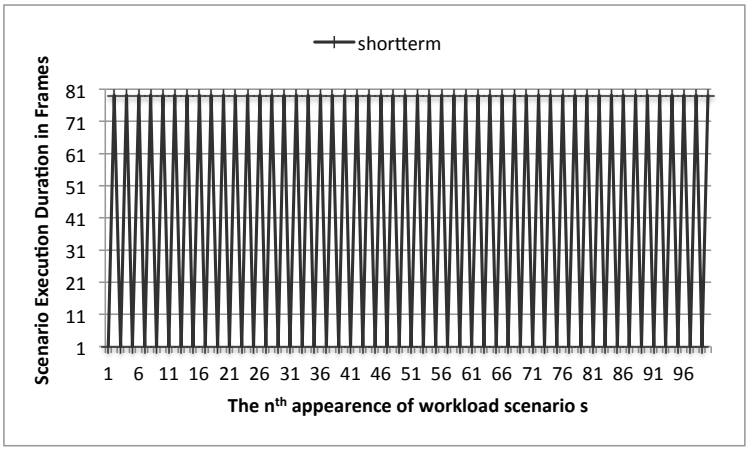


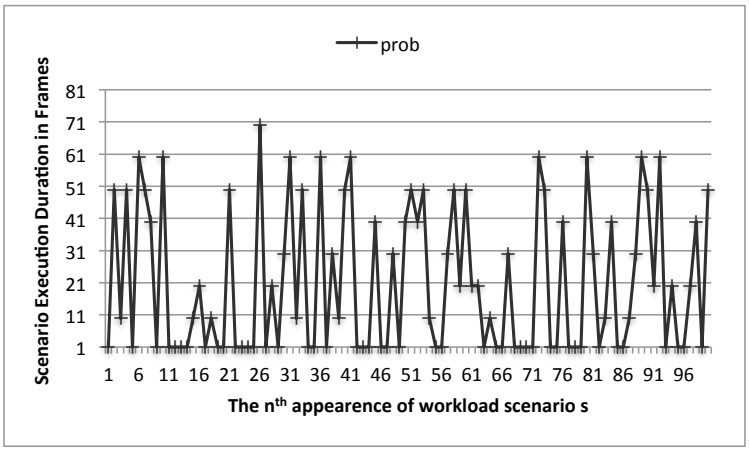
Figure 4.10: Workflow of run-time adaptive resource scheduler.



(a) Scenario sequence with *longterm* distribution



(b) Scenario sequence with *shortterm* distribution



(c) Scenario sequence with *prob* distribution

Figure 4.11: The scenario duration distributions used for generating workload scenarios.

Workflow of the Adaptive Resource Scheduler Based on the above description of the proposed adaptive resource scheduler with the ability of adaptivity throttling, the workflow of such adaptive MPSoC scheduler is illustrated in Figure 4.10. To achieve the adaptivity goal, several components such as: a Scenario

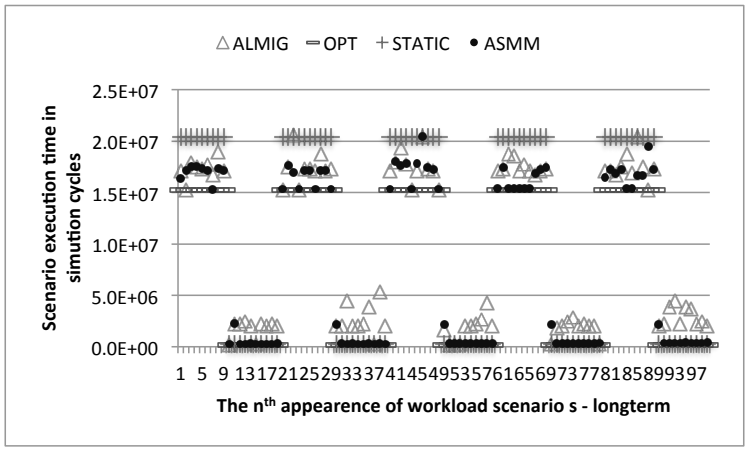
DataBase (SDB), a Run-time System Monitor (RSM) and a Run-time Resource Scheduler (RRS) should be integrated on the target MPSoC system. The SDB is used to store the task mappings as derived from design-time DSE as well as the mapping performance information of each workload scenario and the migrating data size of each task. The RSM is in charge of detecting and identifying the active workload scenario, and also collects statistics (e.g., the actual execution duration of a workload scenario) from the underlying system during the execution of a certain workload scenario. The RRS uses our proposed approach (general hybrid task mapping with adaptivity throttling) to do resource reconfiguration for the identified workload scenario. When the RSM detects a new workload scenario, the RRS obtains the pre-optimized mapping for the current active scenario from the SDB. Hereafter, the RRS makes a reconfiguration decision based on the reconfiguration benefit B of changing the current mapping to the new mapping. According to this decision, the RRS will either reconfigure the system based on the new mapping or continue the system's execution under the current mapping. When the current workload scenario finishes, the RRS updates the information in the corresponding ASMM based on the actual execution duration collected by the RSM.

4.2.3 Experiments

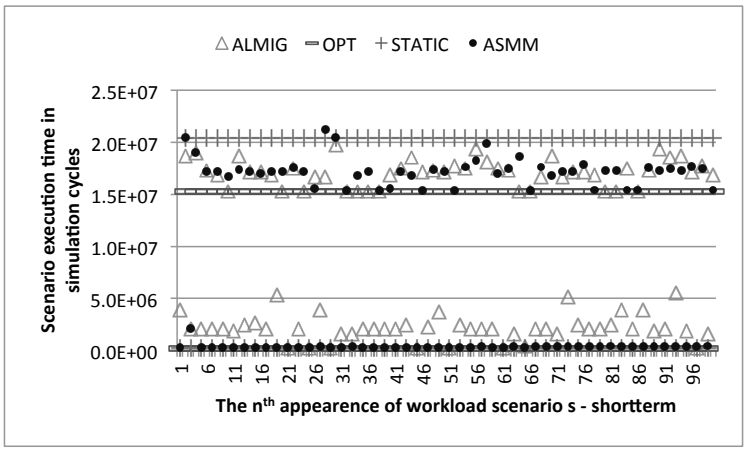
4.2.3.1 Experimental Setup

To evaluate the efficiency of our proposed run-time adaptive resource scheduler, we deploy the extended Sesame simulator introduced in Section 4.1 in which the resource scheduler integrated in the architecture model and the task migration middleware cooperate to realise the simulation of adaptivity throttling for a MPSoC system.

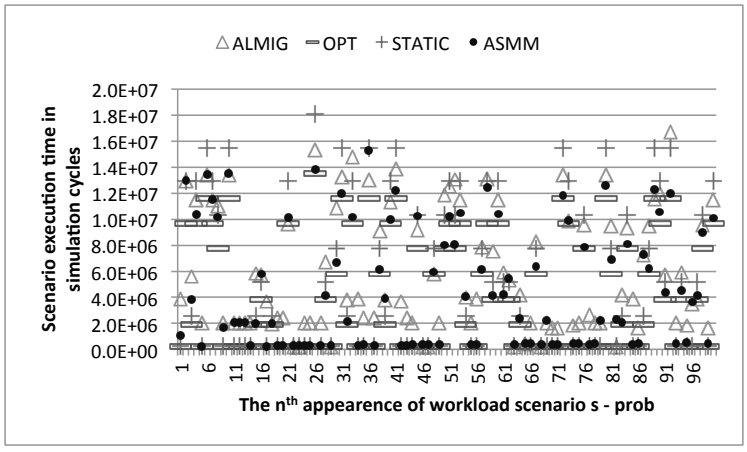
In our experiments, we aim at showing how the proposed technique improves the system performance. To this end, it is important to assess system performance under a variety of different workload scenario behaviours. The actual functionality of the applications within these scenarios is, on the other hand, of lesser importance for this purpose. Therefore, we use synthetic streaming applications within workload scenarios to simplify the simulation process. In our experiments, we prepare five synthetic streaming applications where each application contains only 1 execution mode. In this case, the total number of workload scenarios is 31 ($2^5 - 1$). The number of tasks in each application ranges from 4 to 8. We assume that each task can be executed on each processor of the target MPSoC using the corresponding pre-compiled code (stored in the shared memory). The task execution time and migration data size of each task on each processor have been randomly generated and range between 10,000 and 100,000 time units (simulation cycles) and between 50K and 500K Bytes respectively. Communications between tasks range from 1,000 to 10,000 Bytes in size. Regarding the target architecture, we target a heterogeneous MPSoC containing 5 different processors with different computational characteristics, connected to a shared bus and memory. This target MPSoC is similar to the heterogeneous MPSoC considered in Section 4.1.3.1, consequently a similar *task recreation* mechanism for run-time task migration is used in our experiments of this section.



(a) Execution pattern under *longterm* distribution



(b) Execution pattern under *longterm* distribution



(c) Execution pattern under *longterm* distribution

Figure 4.12: Execution pattern of different resource scheduling approaches.

To model dynamic application behaviour over time (e.g. due to user behaviour), we generate three workload scenario sequences. These sequences are generated in two steps. The first step is to choose a workload scenario from the total 31 workload scenarios considered in our experiment. Each workload scenario has the same probability to be selected. The second step is to generate the du-

ration in *frames* of the selected workload scenario. This process iterates until a pre-defined total frame number (100,000 frames in our case) has been achieved for the scenario sequence. As the workload scenarios considered in our test case need around 40 frames on average to neutralize the reconfiguration cost, we limit the duration of each workload scenario to a value between 1 and 80 frames. In our three scenario sequences, the duration of each workload scenario and frequency of changes to this duration are generated using different distributions as shown in Figure 4.11. Here, the x-axis represents the nth appearance of one specific workload scenario and the y-axis represents the execution duration in *frames* for that particular appearance of the workload scenario. In the *longterm* distribution, the duration of a workload scenario is either long or short and does not frequently change, whereas in the *shortterm* distribution the scenario execution duration does frequently change. Like in *shortterm*, the frequency of changes in the *prob* distribution is high but the actual scenario execution duration now has been generated from the following probability distribution: 1→30%, 11→10%, 21→10%, 31→10%, 41→10%, 51→10%, 61→10%, 71→10%. That is, a workload scenario has a probability of 30% that it will be executed for only 1 frame and 10% for each of the other duration times.

With regard to the parameters of our ASMM, the maximal width and depth of the ASMM table is 4 (2-history, 1-prediction and 1-probability) and 1024 respectively, which is large enough for our test cases. The probability bound *bp* for our ASMM is 50%. This will not lead to either a pessimistic or aggressive decision. As we limit the scenario execution duration from 1 to 80 frames, we divide the scenario execution durations into 8 bins/ranks in our ASMM, each containing a frame range of 10.

4.2.3.2 Experimental Results

In the first experiment, we compare the scenario execution time (including the run-time reconfiguration cost) of each of the three workload scenario sequences under four resource scheduling approaches. The first approach (ALMIG) is executing each workload scenario under its pre-optimized mapping stored in the SDB. This means that the run-time scheduler will always reconfigure the system based on the pre-optimized mapping whenever a new workload scenario appears. The second approach (STATIC) is executing all the workload scenarios under a single pre-optimized mapping that has been optimized to work best – on average – for all workload scenarios. In STATIC, no system reconfiguration takes place. The third approach (ASMM) uses our ASMM-based run-time adaptive resource scheduler. Finally, as a baseline, we also compare to the ideal case (OPT) that applies ALMIG but for which all run-time reconfiguration costs have been discarded. At the beginning of each simulation, the target system is initialised by the mapping of the STATIC approach.

Figure 4.12 shows the results of the scenario execution time of the first 100 appearances of a single selected workload scenario in our three workload scenario sequences. Clearly, in all cases the OPT approach performs best, and STATIC performs worst in those cases where the system should have been reconfigured. We can also see that the scenario execution time is influenced by the reconfiguration cost. For example, consider the bottom parts of the three graphs, i.e., small

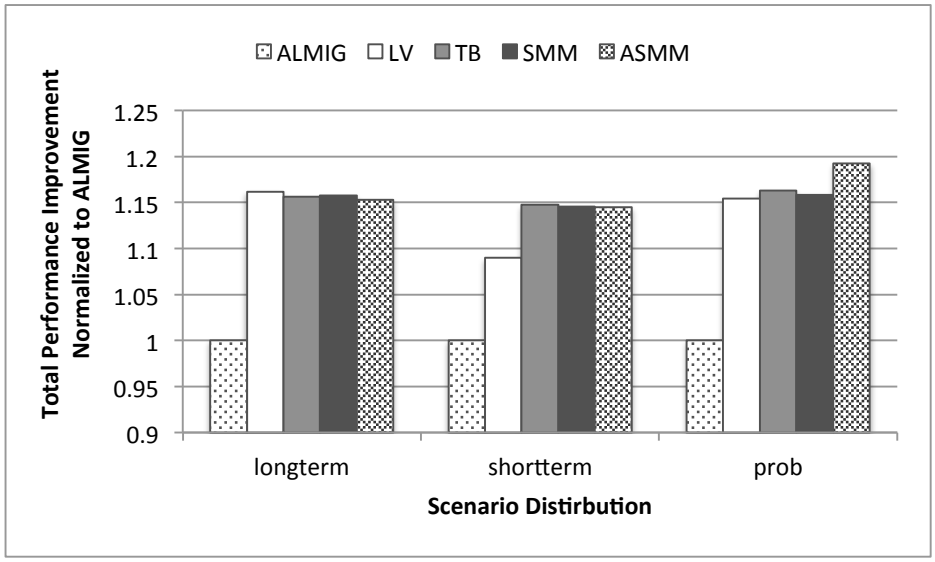


Figure 4.13: Performance comparison of different predictors.

scenario execution durations. Here, one can clearly see that ASMM consistently outperforms ALMIG since the latter is negatively affected by the reconfiguration overhead whereas our ASMM approach is not because it predicted that reconfiguration is not beneficial. For larger scenario execution durations (top parts of the three graphs), the performance of ALMIG and ASMM is similar (i.e., they both reconfigure the system).

In the second experiment, we compare the total execution time of each workload scenario sequence under our adaptive resource scheduler with different scenario duration predictors. We compare three application behaviour predictors to our ASMM approach: a Last Value predictor (LV), a Table-Based predictor (TB) [51] and the original SMM [103]. For a fair comparison, the parameters used for TB and SMM are the same for ASMM. Figure 4.13 shows the results of this experiment. In this figure, the total scenario execution time is normalized to the ALMIG approach. As one can see, in the *longterm* case, all predictors show a good performance as they are all able to accurately predict the execution pattern. For the *shortterm* distribution, the LV predictor shows a poorer performance improvement. This is mainly because of a high prediction error (nearly 100%) of the LV predictor in this distribution. In the *prob* test case, our ASMM predictor clearly outperforms the other three predictors. This is because the scenario execution does not have a fixed pattern anymore which induces a high prediction error for the LV and TB predictors. Compared with the SMM predictor, our ASMM uses the accumulated probability to determine a reconfiguration decision, which increases the chance of making the right decision when reconfiguration is actually needed.

The overhead of our approach involves the run-time computational overhead and the system memory consumption. Comparing the scenario execution time of *ASMM* and *STATIC* in the bottom part of each figure in Figure 4.12, we can see that the *ASMM* results are close to those for *STATIC*. This means that the run-time overhead of our approach does not have a major impact on system performance. In terms of memory usage, our approach needs 880 *bytes* and 920 *bytes*

for storing the total of 31 mappings and the application/architecture information, respectively. The memory usage of our ASMM predictor is dynamic and based on the execution pattern of workload scenarios. In the worst case, for each ASMM with the parameters considered in our experiments, it needs 584 *bytes* to record all possible patterns. However, during the simulation, only a small part of the worst memory usage was actually used for our test cases.

4.2.4 Related Research

In recent years, much research has been performed in the area of run-time task remapping for MPSoC systems to achieve better performance. However, they rarely consider the problem of whether the system will benefit from the resource reconfiguration when the workload of the system changes frequently on MPSoC systems. This problem might be caused by the user behaviour. In this case, it is better to consider the user behaviour [29, 48] or system execution history [103] to further improve the system efficiency. Sarikaya et al. [103] proposed SMM to predict the run-time application behaviour, and applied this technique to an adaptive dynamic power management scheme. In our approach, we modified their SMM to predict the scenario duration which is part of our adaptive run-time framework. In [29], the user behaviour information is used to adapt the strategy used for resource allocation at run time. Based on the user behaviour, the online machine learning model will predict which kind of communication contention should be minimized on an NoC based MPSoC system. The authors of [48] proposed a customer-aware task allocation and scheduling for MPSoCs. In their approach, an initial task allocation and scheduling (TAS) solution under the objective of minimizing the energy consumption and system lifetime for each execution mode is generated at design time. At run time, they conduct online adjustment of the TAS based on the processor usage history to guarantee the lifetime reliability and/or reduce the energy consumption. Different to these previous efforts, we use the user behaviour/system execution history to control the resource allocation process.

4.2.5 Conclusion

To increase the efficiency of MPSoC systems, we have proposed a run-time adaptive resource scheduler that reconfigures the system based on past and future (predicted) application workload behaviour. At design time, we explore performance optimal task mappings for different workload scenarios. These pre-optimized mappings are used at run time by the resource scheduler to reconfigure the system resources. The decision of whether or not to reconfigure is made based on the scenario execution pattern. By using the proposed approach, the system can adapt its behaviour according to e.g. user behaviour. Experimental results confirm the effectiveness of our approach.

4.3 A Run-time Self-Adaptive Resource Allocation Framework

In the previous section, we have illustrated how to solve the blind adaptivity issue of general hybrid task mapping approaches for fine-grained workload scenarios on

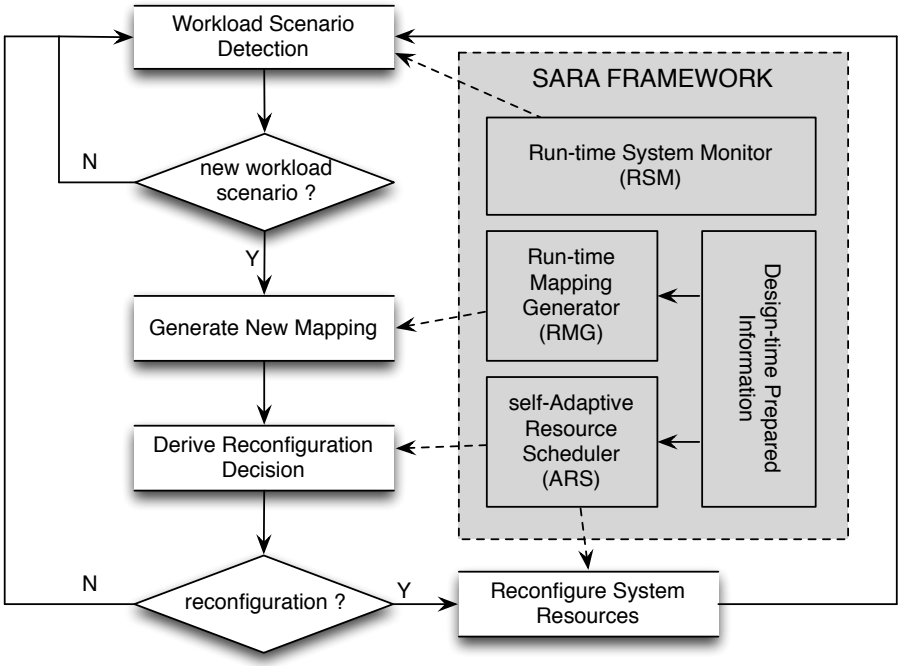


Figure 4.14: The structure and workflow of SARA framework

MPSoC systems. However, the method proposed in the previous section assumed that the optimal mapping of each target workload scenario has been explored at design time and stored on the system for run-time usage. Consequently, the system manager can directly apply the pre-optimised mapping for a newly detected workload scenario to make a reconfiguration decision. As mentioned before, this assumption is only applicable when the number of target workload scenario is relatively small. It means that the approach proposed in the previous section still suffers from the scalability and flexibility problem with an increasing number of target applications. Therefore, in this section, we combine the technique of adaptivity throttling proposed in the previous section and the novel hybrid task mapping approach from Section 3.3 together into a scalable self-adaptive MPSoC management framework –Scenario-based Adaptive Resource Allocation framework (SARA)– to to solve the issues of general hybrid task mapping approaches: scalability, flexibility and blind adaptivity, thereby considerably improving the system performance.

4.3.1 Scenario-based Run-time Adaptive Resource Allocation Framework

Similar to the adaptive resource allocation framework proposed in the previous section, our SARA framework consists of three components: a Run-time System Monitor (RSM), a Run-time Mapping Generator (RMG) and a self-Adaptive Resource Scheduler (ARS). The RSM is in charge of detecting the active workload scenario on the target MPSoC system and dynamically collecting system statistics. The RMG and ARS are responsible for the system adaptation and address the issues of general hybrid task mapping approaches as explained above. Figure 4.14

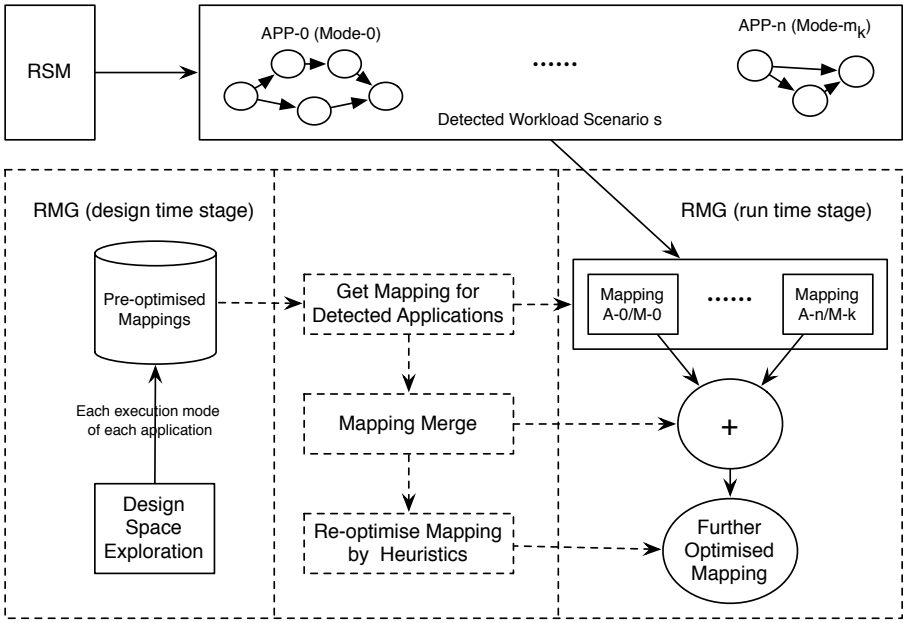


Figure 4.15: Overview of mapping generation in RMG

shows the high-level system workflow of the SARA framework. When the RSM detects a new workload scenario, the RMG will generate a new mapping for the detected (active) scenario. Hereafter, the ARS makes an adaptation decision by predicting the benefit of changing the current mapping into the newly proposed one (by the RMG). According to this decision, the ARS will then either reconfigure the system based on the new mapping or continue the system's execution under the current mapping.

4.3.1.1 Scalable Run-time Task Mapping

In the SARA framework, we solve the task mapping problem by using the hybrid task mapping technique of Section 3.3 which prepares *partial task mappings* for workload scenarios at design time and completes the mappings for the entire scenario at run time using the RMG component. Figure 4.15 gives an overview of how the RMG generates a new mapping for the workload scenario detected by the RSM. At design time, a performance-optimized task mapping (and, if needed, also a power-optimized mapping) for each execution mode of each *application in isolation* is determined by our scenario-aware DSE techniques introduced in Section 2.3 (in case of optimising the mapping performance) and Section 3.2.3 (in case of multiple mapping optimisation objectives). This significantly reduces the time and memory requirements needed for, respectively, finding and storing the pre-optimized task mappings at design time. Moreover, if a new application needs to be supported on the target MPSoC system, this would only require providing the pre-optimized mappings of this new application to the RMG without redoing the entire process of design-time mapping preparation for all possible (new) workload scenarios.

At run time, after the RSM has detected a new workload scenario, the RMG will first merge the pre-optimized mappings of each separate, active application

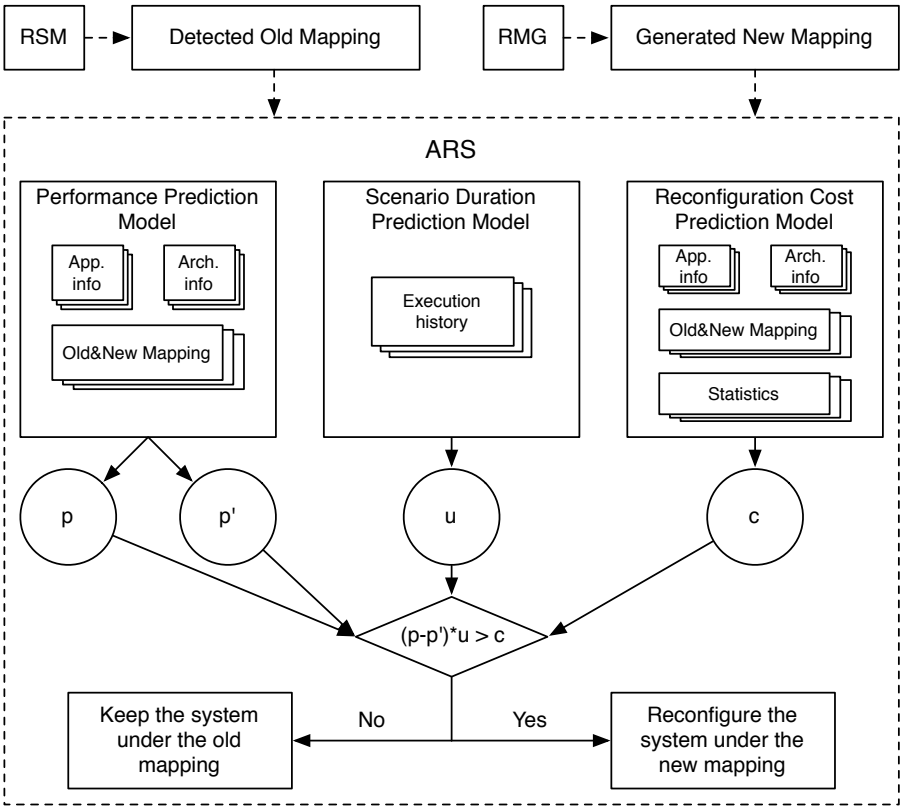


Figure 4.16: Overview of adaptive system reconfiguration in ARS

in the detected workload scenario to form a first-order mapping for the entire scenario. Subsequently, the RMG will then further optimize this first-order mapping by using run-time mapping optimization heuristics, based on e.g. a load balance algorithm or a dynamic mapping optimization algorithm such as the STM or EIM proposed in Chapter 3.

4.3.1.2 Adaptivity Throttling

The ARS component of our SARA framework uses the adaptivity throttling technique proposed in Section 4.2 to solve the blind adaptivity issue of MPSoC systems, and consequently improves the adaptivity of MPSoC systems with regard to the granularity of target workload scenarios. Figure 4.16 illustrates how the ARS component conditionally reconfigures the target system based on the outcome of the prediction models (i.e, $(p - p') * u > c$). In this figure, the information about the target applications and hardware architecture used in the performance prediction model as well as the reconfiguration cost prediction model should have be prepared at design time, and depends on the type of models used for these predictions (as discussed below).

The prediction models in ARS cannot be computationally intensive as they have to efficiently make a reconfiguration decision at run time. For the performance and reconfiguration cost prediction, relatively simple regression models or analytical models such as the performance model from [98, 97] can therefore be applied. The prediction of scenario execution duration is the most important

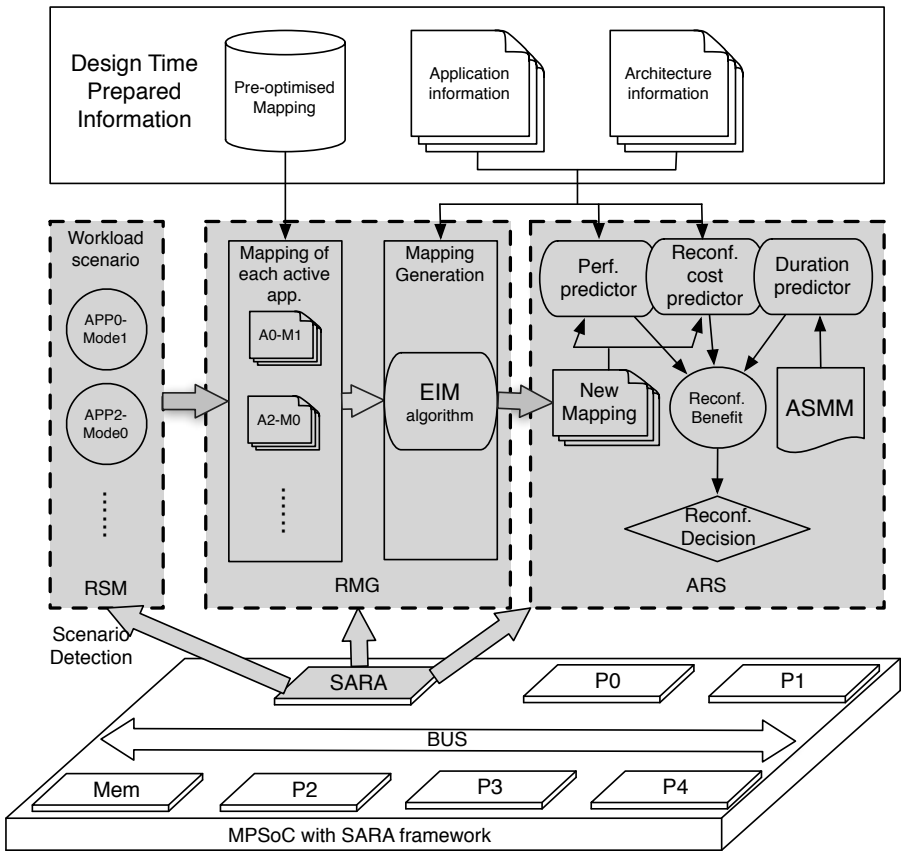


Figure 4.17: An instance of SARA for the target MPSoC system

part of the ARS. It is a dynamic parameter that could be heavily influenced by user behaviour. A commonly used predictor for such kind of parameters, which has also been used in our ARS component, is the history-based predictor such as a last value predictor, table-based predictor and the Statistical Metric Model (SMM) [103]. They can predict the future value of a parameter – in our case the duration of a newly detected workload scenario – based on its history information.

After having derived a reconfiguration decision based on the three predictive models for performance, reconfiguration cost and scenario duration, the ARS will either reconfigure the system according to the new mapping or keep the old mapping. By applying such adaptivity throttling in the ARS, our adaptive MPSoC system is able to cope with fine-grained workload scenarios for which it is not beneficial to reconfigure the system.

4.3.2 Implementation of SARA on the Target Heterogeneous MPSoC

In this section, we present an instance of our SARA as shown in Figure 4.17 on a heterogeneous MPSoC system with shared memory. With this SARA instance, our optimization goal is to maximize the system performance for a sequence of workload scenarios. For this SARA instance, the details of design-time preparation, the run-time mapping optimizing heuristic in the RMG and the prediction models in the ARS will be detailed in the following subsections.

4.3.2.1 Design Time Preparation

For the design time mapping optimization, the same DSE approach from Section 4.2.2.1 is deployed. After applying the design-time DSE, the performance-optimized mappings are stored in system memory for run-time usage.

Besides the performance-optimized mapping for each execution mode of each isolated application, the execution time of each task on each processor, the communication times between tasks on different communication channels of the target system and the migrating data size between processors for each task should also be analyzed at design time and stored on the target system for the purpose of mapping performance prediction and system reconfiguration cost prediction.

4.3.2.2 Run-time Mapping Optimizing Heuristic

In the RMG, we have adopted the EIM algorithm as the mapping optimizing heuristic. This algorithm aims at generating mappings with good quality in terms of system throughput (in this section, we have not considered the algorithm's ability to also include an energy consumption constraint).

4.3.2.3 Mapping Performance Prediction

For the target heterogeneous MPSoC system with shared memory, we use a simple linear analytic model to dynamically predict the performance of different task mappings for workload scenarios. At design time, the execution time of each task and the communication time between tasks for one unit of workload of each application have been analyzed and stored on the target system. Using this information, the performance of a certain workload scenario s_i under a target task mapping tm_i^j is derived by equation 4.2 where the performance of each active application app_k is predicted by equation 4.6.

$$p_{ij}^k = CC_{ij}^k + BK_{ij}^k \tag{4.6a}$$

$$CC_{ij}^k = \sum_{0 \le m < t} et_{ij}^{km} + \sum_{0 \le n < l} ec_{ij}^{kn} \tag{4.6b}$$

$$BK_{ij}^k = \sum_{q \neq k} \sum_{t_i^{qs} \in T_{ij}^{qk}} et_{ij}^{qs} + \sum_{q \neq k} \sum_{c_i^{qs} \in C_{ij}^{qk}} ec_{ij}^{qs} \tag{4.6c}$$

where CC_{ij}^k represents a conservative estimate (no concurrency is taken into account) of the total execution time of app_k in scenario s_i under mapping tm_i^j of all t tasks and l communications in app_k. BK_{ij}^k is the total time of tasks $t_i^{qs} \in T_{ij}^{qk}$ and communications $c_i^{qs} \in C_{ij}^{qk}$ from other active applications that contend for resources with app_k. Here, T_{ij}^{qk} and C_{ij}^{qk} is the set of tasks and communications from app_q that are mapped onto the same resource (under tm_i^j) with any task and communication of app_k respectively.

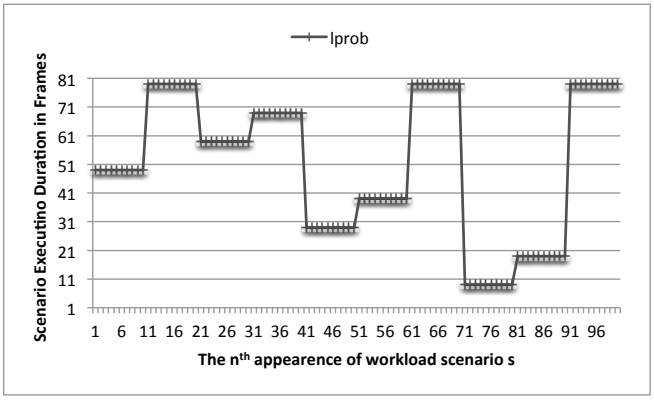


Figure 4.18: The scenario duration distribution used for generating *lprob*

4.3.2.4 Reconfiguration Cost Prediction

Similar to that has been discussed in Section 4.2.2.2, the reconfiguration cost on our target MPSoC system includes two parts: the overhead of our SARA framework and the task migration cost during system reconfiguration. The overhead of SARA is determined by means of measurements and the task migration cost is calculated by the linear analytic model as introduced in Section 4.2.2.2 (see Equation 4.4). Here, we label the overhead of SARA as $CSara$ and the task migration cost as $CMig$. Consequently, for a certain workload scenario s_i with an original task mapping tm_i^j and a newly generated mapping $tm_i^{j'}$, the reconfiguration cost can be derived by the following equation. Notice that, the overhead of this specific SARA instance ($CSara$) includes the time of deriving a new mapping in the RMG, estimating mapping performance for both the old mapping and the new mapping, calculating task migration cost $CMig$ and updating the system run-time information (e.g., actual scenario execution duration) in SARA.

$$c_i^{jj'} = CSara_i^{jj'} + CMig_i^{jj'} \tag{4.7}$$

4.3.2.5 Reconfiguration Decision Prediction

Different with the description of the ARS in Section 4.3.1.2, in our SARA instance of the target MPSoC system, we do not try to predict the exact value of the scenario execution duration n which is a dynamic parameter that relates to the user/system behaviour. In most MPSoC systems (except for systems with periodic workload scenarios), it is hard to accurately predict the exact value of this parameter based on history information. However, we can avoid this problem by approaching our goal slightly differently. Our purpose is to derive a reconfiguration decision based on the reconfiguration benefit B. If the execution duration of the detected workload scenario is higher than a certain value (boundary), then the system can be reconfigured. Consequently, there is no need to directly predict a concrete value for the scenario execution duration but, instead, it is sufficient to predict the probability that the execution duration is higher than the given boundary. Therefore, the ASMM prediction model from Section 4.2.2.2 has been adopted for this purpose.

4.3.3 Experiments

In this subsection, we present the experimental results in which we investigate various aspects of our proposed SARA framework. The simulation environment is similar to that was adopted in Section 4.2.3.1. We have implemented our SARA framework on the migration enabled Sesame simulator. In this simulation framework, the synthetic applications and the heterogeneous MPSoC system as described in the experiments of previous section are used again for the experiments of this section.

4.3.3.1 Evaluating Adaptivity

In this experiment, the main goal is to evaluate the adaptivity of our approach. In the experiment of the previous section, we have shown the results of using the general hybrid task mapping approach with adaptivity throttling for different workload scenario sequences of five synthetic applications on the target heterogeneous MPSoC. In this experiment, besides the three workload scenario sequences *longterm*, *shortterm* and *prob*, we also consider another scenario sequence: *lprob*. As shown in Figure 4.18, the *lprob* sequence is generated by combining the approach of *prob* (considering a different probability distribution) and *longterm* described in Section 4.2.3.1. The *prob* scenario sequence of Section 4.2.3.1 is relabelled to *sprob* as the scenarios in the sequence frequently change, similar to the *shortterm* sequence. Our *lprob* scenario sequence is generated under the following probability distribution: 9→10%, 19→10%, 29→10%, 39→10%, 49→10%, 59→10%, 69→10%, 79→30%. Whereas the *longterm* and *shortterm* scenario sequences more or less reflect extreme cases in workload scenario behaviour, the two *prob* sequences possibly exhibit a more realistic view on dynamic behaviour in application workloads.

For the purpose of comparison, we compared our SARA framework with three alternative approaches. First, an approach (*STATIC*) in which all applications are statically mapped (i.e., no run-time mapping takes place) using a mapping which has shown to be optimal *on average* for all possible workload scenarios. Second, an approach (*MIGRATE-OPT*) that always reconfigures the system whenever a new workload scenario has been detected according to a corresponding pre-optimized mapping derived at design time. Note that this approach, which is similar to how many state-of-the-art run-time mapping techniques operate, stores 31 mappings (one mapping for each workload scenario) in total on the system. Finally, we also compare to SARA without adaptivity throttling (*SARA-NOTH*).

The results of the total execution time (including system reconfiguration time) of all workload scenarios in each generated scenario sequence are shown in Figure 4.19. In this figure, we can clearly see that our *SARA* approach outperforms the three alternative approaches in almost all scenario cases except the *STATIC* approach in the *sprob* case, whereas *SARA-NOTH* performs the worst. According to the above-mentioned distribution of each scenario sequence, the average scenario duration of *sprob* is less than 30 frames and *lprob* is higher than 60 frames, where *longterm* and *shortterm* both have a mean execution duration of around 40 frames. As mentioned before, the workload scenarios considered in our test cases need around 40 frames on average to neutralize the reconfiguration cost. From the perspective of average scenario duration, one can easily understand the reason

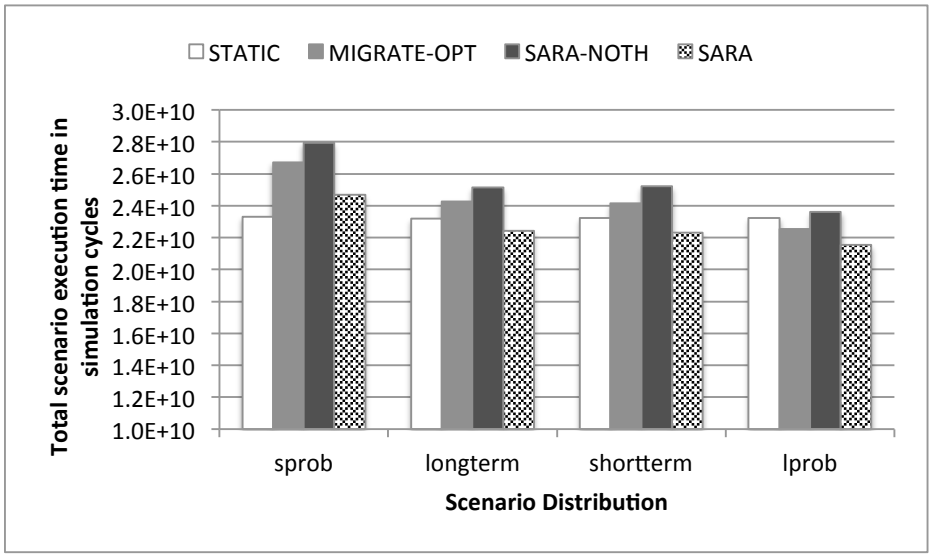
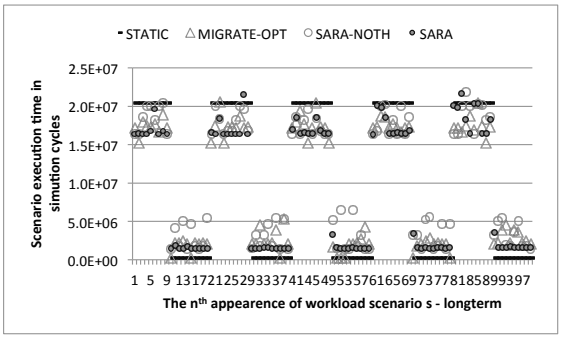


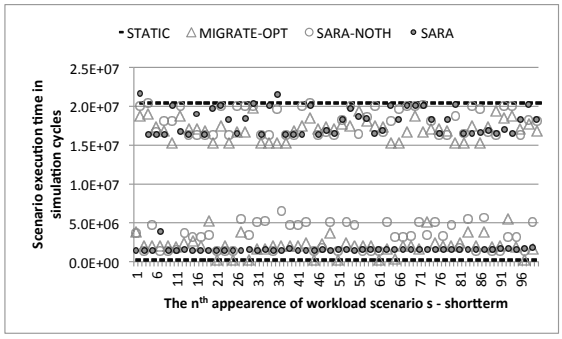
Figure 4.19: Performance comparison of different resource scheduling approaches

of why the approaches that always reconfigure the system for a newly detected workload scenario, i.e. *MIGRATE-OPT* and *SARA-NOTH*, perform worse than *STATIC* in the scenario sequences of *sprob*, *longterm* and *shortterm*. However, when the average scenario duration is large enough like in *lprob*, this kind of approaches start to perform better than *STATIC*. Notice that with even larger average scenario durations the performance difference of the approaches that (always) migrate and *STATIC* wil only become larger. Comparing *MIGRATE-OPT* and *SARA-NOTH*, the former one always performs better as it uses the pre-optimized mappings for entire workload scenarios which are better in mapping quality compared to the mappings found by the EIM algorithm in *SARA-NOTH*. With regard to our SARA instance, it shows good behaviour for each test case because of its ability to throttle adaptivity. It even performs relatively well in the case of *sprob* where 70% of the workload scenarios have a short duration and for which reconfiguring the system is not beneficial. In this particular case, the static mapping approach works best as this approach does not suffer from any run-time overheads.

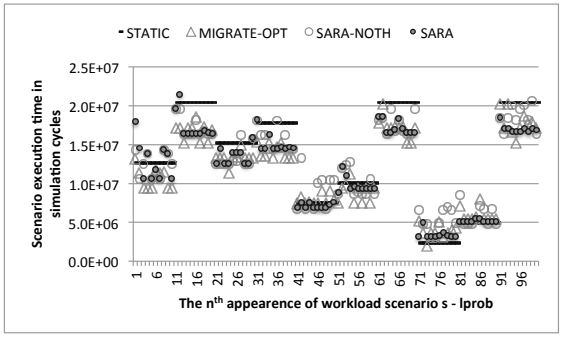
To better understand how these adaptive techniques work, we zoom into the run-time executing behaviour of a certain workload scenario in the scenario sequences for these techniques. Figure 4.20 shows the results of the scenario execution time (including system reconfiguration time) of the first 100 appearances of a single selected workload scenario in our four workload scenario sequences. Clearly, the *MIGRATE-OPT* approach performs best and *STATIC* performs worst in those cases where the system should have been reconfigured (i.e., scenario execution duration is large enough such that reconfiguration is beneficial): see the top parts of the four figures. On the other hand, in the cases where the system should not be reconfigured (bottom parts of the four figures), *STATIC* is the best and *SARA-NOTH* is the worst. We can also see that the scenario execution time is influenced by the reconfiguration cost. For example, consider the bottom parts of the upper two graphs (*longterm* and shortterm), i.e., small scenario execution durations. Here, one can clearly see that *SARA* consistently outperforms *MIGRATE-OPT* and *SARA-NOTH* since the latter two approaches are negatively affected by the



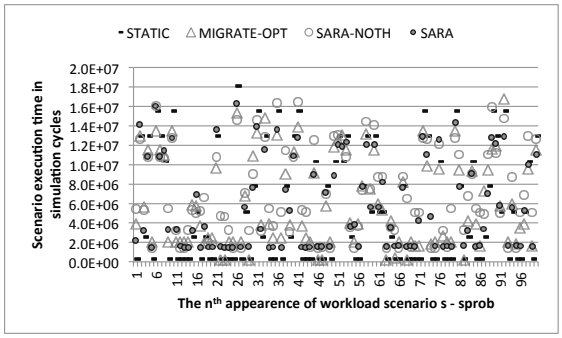
(a) Execution pattern under *longterm* distribution



(b) Execution pattern under *shortterm* distribution



(c) Execution pattern under *lprob* distribution



(d) Execution pattern under *sprob* distribution

Figure 4.20: Execution pattern of different resource scheduling approaches

reconfiguration overhead whereas *SARA* is not because it accurately predicted that reconfiguration is not beneficial. However, compared with *STATIC*, *SARA* still suffers from its computational overhead. Moreover, erroneous predictions inducing unnecessary system reconfigurations also affect the performance of *SARA*. In the upper two graphs (*longterm* and *shortterm*), there is a clear pattern for the duration of workload scenarios where *SARA* can accurately predict. In these two cases, the performance is mainly affected by its computational overhead. However, in the case of *sprob*, a high prediction error is the main factor that influences the performance of *SARA* as it is hard to predict such a random scenario behaviour with frequent changes. This implies that the prediction error mainly comes from the scenario duration predictor (ASMM) in the ARS of our SARA instance. A similar situation can be seen in the first 10 appearances of scenario *s* in the graph of *lprob* where *SARA* shows a poor behaviour (filled dots randomly distributed around the straight line of *STATIC*) whereas SARA's performance in the other parts of this graph is much better. This is also caused by a high prediction error in SARA. However, in this case, the prediction error is caused by the accuracy of the mapping performance and reconfiguration cost predictors in the ARS of SARA. As the scenario duration of these first 10 appearances of scenario *s* is very close to the average frame number needed to neutralize the reconfiguration cost in our test cases, a small prediction error of these two parameters can easily lead to an erroneous system reconfiguration prediction.

From this experiment, we can see that our SARA framework is able to handle complex and dynamic workload scenarios and further improves the system

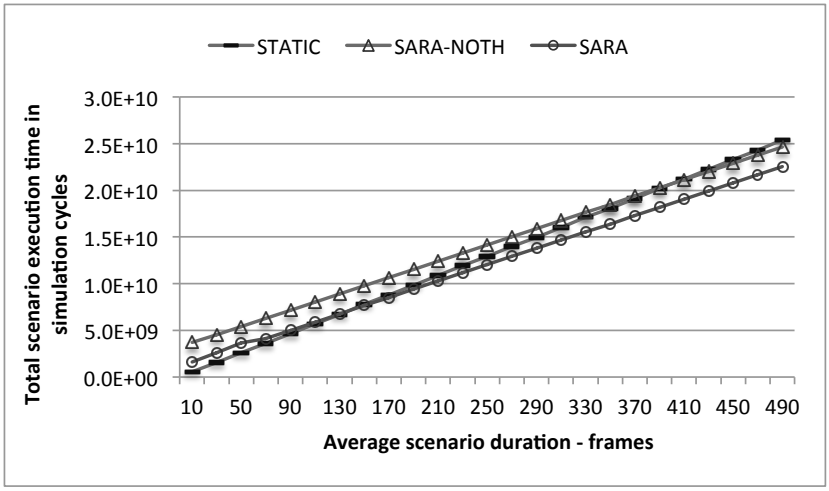


Figure 4.21: Performance comparison of different resource scheduling approaches in complex workload scenarios

efficiency by run-time task remapping and adaptivity throttling.

4.3.3.2 Evaluating Scalability

To evaluate the scalability of our approach, we consider ten synthetic streaming applications in this experiment, where each application has the same properties as the applications used in the previous experiment. Consequently, there are 1023 workload scenarios in total. For the purpose of simulating the dynamic application behaviour over time, we use a scenario sequence which is similar to *lprob* from the first experiment but without limiting the scenario execution duration. In this experiment, the impact of an increasing scenario execution duration will be studied. Furthermore, the parameters of the ASMM are the same as in the first experiment except for the maximal depth of the history tables which is set based on the actual scenario duration *bins*.

Figure 4.21 shows the total execution time (including system reconfiguration time) of 100 appearances of the workload scenario with all the 10 applications active. The x-axis in Figure 4.21 represents the average scenario duration (in scenario frames) of the 100 appearances of the workload scenario. For better visibility of the results, we have discarded the results of *MIGRATE-OPT* which is a line between *SARA-NOTH* and *SARA*, but converges earlier with *SARA* than *SARA-NOTH*. From this figure we can clearly see that our SARA framework still works well when the number and the complexity of workload scenarios increases. Note that in this experiment, as the workload scenario contains a large number of tasks and communication channels, the system reconfiguration cost is also larger compared with the first experiment. Consequently, the number of average scenario frames needed for neutralizing the reconfiguration cost is also bigger than in the first experiment. Taking our *SARA* instance for example, the computational overhead of SARA can be neutralized when the average number of scenario frames is larger than 150 (the crosspoint of *SARA* and *STATIC* is around 150 frames).

To further improve the scalability of our SARA framework, we use a caching mechanism that aims at avoiding the run-time mapping optimization heuristic

Table 4.3: Run-time system storage demands (*bytes*) of the different techniques

Experiment1	**Mapping**	**App&Arch Inf.**	**ASMM**
STATIC	x	x	x
MIGRATE-OPT	880	x	x
SARA-NOTH	55	920	x
SARA	55	920	dynamic
Experiment2	**Mapping**	**App&Arch Inf.**	**ASMM**
STATIC	x	x	x
MIGRATE-OPT	56320	x	x
SARA-NOTH	110	1840	x
SARA	110	1840	dynamic

becoming a performance bottleneck of the target system when frequent scenario changes occur. To this end, the SARA framework uses a small amount of system memory like a scratchpad to cache the mappings optimized by the run-time heuristic for the workload scenarios that undergo the most frequent changes. Consequently, it is able to further save considerable computational overhead with regard to run-time mapping optimization, especially for complex workload scenarios.

4.3.3.3 System Storage Overhead

Regarding the run-time system storage consumption of the four studied approaches, several assumptions should be mentioned. On our target MPSoC system, we store all the design-time prepared information in the shared memory. For storing the pre-optimized mappings, we assume that the mapping information of each task and each communication channel between tasks can be stored in one *byte*. In the first experiment, there are 30 tasks and 25 communication channels in total for all the five synthetic streaming applications. Consequently, to store a pre-optimized mapping, the maximal memory usage is 55 *bytes* (all tasks and communication channels are active). The total number of tasks and communication channels in the second experiment is 60 and 50 respectively. Beside the pre-optimized mappings, in our SARA framework, we also need to store the application/system information and the scenario execution history information if the ARS predicts scenario duration based on history information like in the ASMM-based method of our SARA instance. Here, we assume that each piece of application/system information needs one *word* of system memory and each history scenario duration is encoded using one *byte*.

Based on these assumptions, Table 4.3 gives the run-time system storage demands of the four approaches in the above two experiments. From this table, we can see that our SARA instance consumes the largest system memory usage in the first experiment. However, in our SARA framework, the memory usage only linearly increases with the number of applications for storing the *Pre-optimized Mappings* and *App&Arch Information* on a certain target MPSoC system. It does

not have the scalability problem of *MIGRATE-OPT*. Consequently, in the second experiment, the memory usage of *MIGRATE-OPT* is much larger than *SARA-NOTH* and *SARA*. With regard to the memory usage of the ASMM in our SARA instance, it is dynamic at run-time and depends on the application behaviour as discussed in Section 4.2.3.2. The memory usage of the ASMM will increase exponentially with the number of applications. However, this scenario execution duration predictor can also be implemented by other techniques like Neural Networks [62, 37] if the memory usage becomes an issue.

4.3.4 Conclusion

To increase the efficiency of MPSoC systems, in this section, we have proposed a scalable, run-time adaptive resource scheduler that reconfigures the system based on the system workload and user behaviour. At design time, we explore performance (near) optimal task mappings for different workload scenarios. These pre-optimized mappings will then be used at run time by the resource scheduler to reconfigure the system resources. The decision of whether or not the system should be reconfigured is made explicitly based on the scenario execution history pattern. By using the proposed approach, the system is able to effectively adapt its behaviour according to user behaviour, as demonstrated by our experimental results.

4.4 Summary

In this chapter, we focused on the *blind adaptivity* problem of general hybrid task mapping approaches for MPSoC system with fine-grained workload scenarios. To investigate this problem, the system reconfiguration cost should be carefully considered for dynamic task remapping at run time. For this purpose, we extended our Sesame simulator with the ability of modeling, simulation and exploration of different system reconfiguration mechanisms and policies in MPSoCs. This extended Sesame simulator was used as the evaluating tool for the work of this chapter. To solve the blind adaptivity problem, we proposed an *adaptivity throttling* technique that tries to avoid unnecessary system reconfigurations by predicting whether or not reconfiguration of the system actually is beneficial based on the active workload scenario and the status of the hardware platform. This proposed technique was evaluated combining with a general hybrid task mapping approach where the pre-optimised mapping of a detected workload scenario is considered for such reconfiguration prediction and consequently is applied (or not) on the system according to the prediction. After evaluating the effectiveness of this adaptivity throttling technique, it was further adopted in our SARA framework that was proposed to solve the previous discussed issues like scalability, flexibility and also blind adaptivity of most general hybrid task mapping techniques for MPSoC systems. In the SARA framework, the scalability and flexibility problem is solved by applying the divide-and-conquer method that was used in the novel hybrid task mapping approach of Section 3.3.

CHAPTER 5

Toward the design of future large scale adaptive MPSoC systems

The scalability with regard to the number of target workload scenarios in general hybrid task mapping techniques of MPSoC systems has been investigated in the previous chapters. However, as mentioned in Section 1.4, this scalability problem is also related to the size of the target hardware platform (i.e. the number of hardware components in the system). With the technological advancements, the number of processing elements in a MPSoC system will increase to tens or maybe even hundreds. This imposes a big challenge to manage the hardware resources at run-time in a scalable manner. In this chapter, we focus on providing a scalable resource allocation approach for large-scale MPSoC systems with complex and dynamic workload behaviour.

The architecture of future MPSoCs is still an open research problem in the domain of embedded systems. A popular solution are tile-based architectures like Tilera's 64-core TILEPro64 processor [134] and Intel's 48-core Single-Chip Cloud Computer (SCC) [74]. In the first section, such a tile-based MPSoC architecture is presented as the prototype of our target large-scale heterogeneous MPSoC systems. After that, a hierarchical resource management mechanism is proposed for such a tile-based MPSoC system. Based on this mechanism, a Scenario-based Hierarchical run-time Adaptive Resource Allocation (SHARA) framework is implemented and evaluated for the target large-scale MPSoC system. With this framework, we want to demonstrate that the adaptivity techniques studied thus far can also be applied to future, large-scale MPSoCs. At the end of this chapter, the related research and a short conclusion are presented.

5.1 Tile-based MPSoC architecture

With the scaling of technology, future MPSoCs will feature tens up to hundreds of (heterogeneous) processing elements that are all integrated on a single chip.

This chapter is based on:
- W. Quan and A. D. Pimentel, "Adaptive Large-scale MPSoC Systems," *Submitted*.

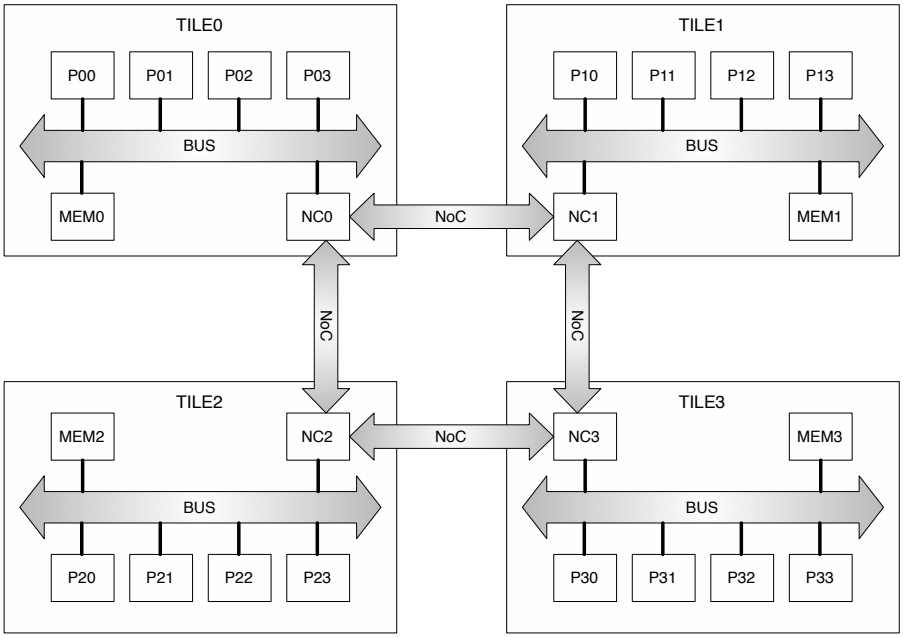


Figure 5.1: The architecture of the target MPSoC system

Traditional system design methods [73] of MPSoCs are not anymore appropriate to the design of future large-scale MPSoC systems. The design of future large-scale MPSoC systems is still an open research question. A very popular prototype is the tile based scalable system [134, 129, 74, 47, 76, 105]. In this chapter, we consider the tile-based heterogeneous MPSoC system illustrated in Figure 5.1 as our target system. From this figure, we can see that the system is composed of four identical tiles. In each tile, there are four heterogeneous processing elements connected to a shared memory by bus. Note that the communication in our target system has multiple levels like the intra-processor communication by buffers, intra-tile communication by a shared bus and inter-tile communication by a Network-on-Chip (NoC) similar to [105]. The reason for considering this type of MPSoC architecture is that the architecture of a tile in our target system can be designed by current state-of-the-art MPSoC design approaches like the work from [95]. Also this kind of layered architecture could reduce the communication congestion that might happen in a large-scale NoC.

Our objective in this chapter is to improve the system performance by adaptively reconfiguring the target large-scale MPSoC system based on dynamically derived mappings for each detected workload scenario. It includes: firstly deriving a spatial and temporal optimised task mapping for each newly detected workload scenario on the target MPSoC system and secondly reconfiguring the system according to the newly derived mapping when the reconfiguration is predicted to be beneficial.

5.2 Hierarchical Control Mechanism

From the perspective of the control mechanism for system resource management, it can be divided into three categories [115, 105]: 1) centralised resource manage-

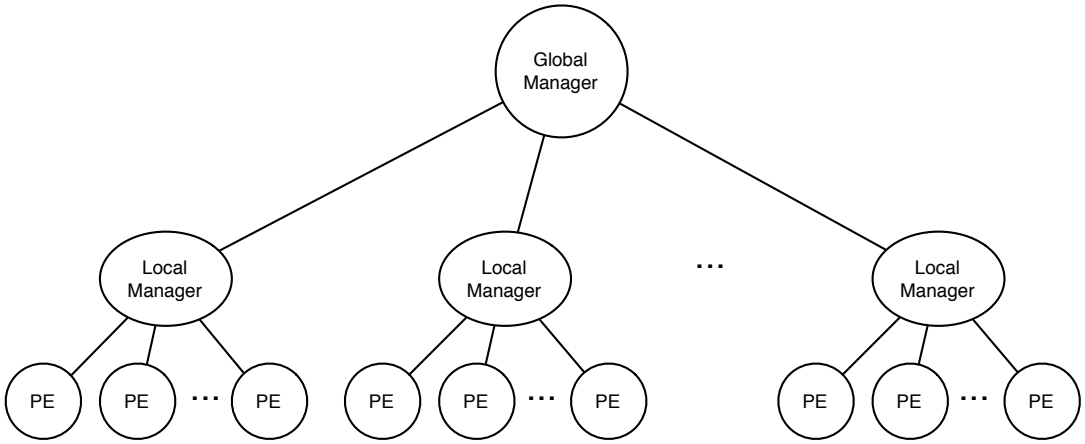


Figure 5.2: The hierarchical resource management on the target MPSoC system

ment, 2) distributed resource management and 3) the combination of two previous methods. On modern MPSoC systems where a limited number of processing elements are present, centralised approaches are usually considered because of their effectiveness and simplicity [71]. However, when the system scales, a centralised approach often suffers from its performance bottleneck as heavy communication might happen during resource reallocation when the number of processing elements is very large. To avoid this problem, distributed resource management approaches have been proposed [59, 108, 5]. However, these distributed approaches are usually complex and not easy to implement. Most importantly, these kind of approaches can only find a local optimal resource allocation solution. Consequently, a trade-off solution that combines the centralised and distributed resource management is commonly considered in multi-/many core systems [105]. For our target large-scale MPSoC system, we also adopt this hybrid approach where the control structure is hierarchically organised as in Figure 5.2. The Global Manager (GM) takes charge of workload partition among tiles and each Local Manager (LM) optimises the resource allocation inside a tile for the assigned applications. This control mechanism can be implemented on the target system by either dedicated hardware (controller) or software.

According to the above-mentioned control mechanism, a scenario-based hierarchical run-time adaptive resource allocation framework is proposed to adaptively reallocate the hardware resources on large-scale heterogeneous MPSoC systems. Note that this framework is not limited to the architecture we considered in this chapter, as tiles can be virtually divided on a target system. Taking a general NoC-based MPSoC system as illustrated in Figure 5.3 as an example, the entire system can be firstly divided into identical tiles and then controlled by our approach. The details of our proposed SHARA framework will be explained in the following section.

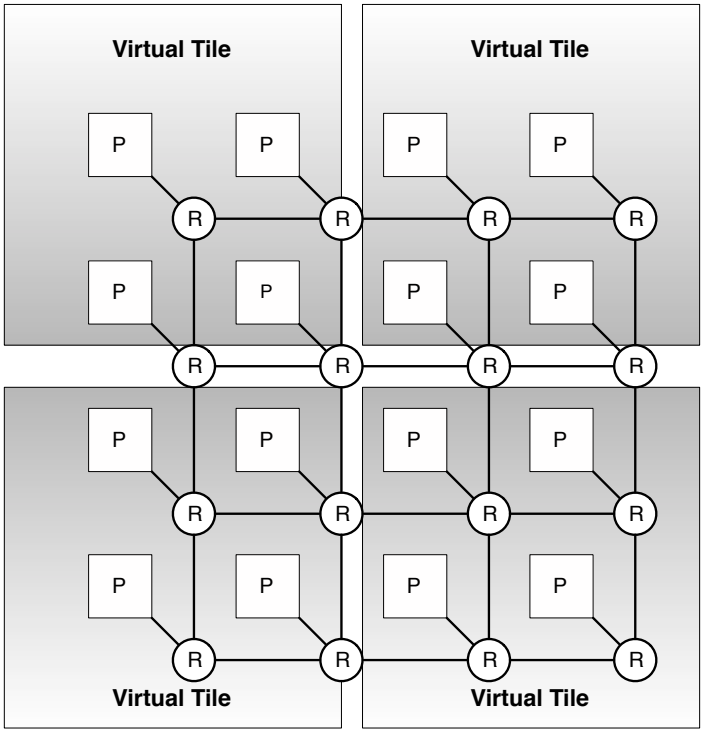


Figure 5.3: An example of dividing a MPSoC into virtual tiles

5.3 Scenario-based Hierarchical Adaptive Resource Allocation

When considering dynamic resource reallocation on a MPSoC system, three steps are needed. The first step is to decide when the resource reallocation should be triggered. For example, it could be a scenario change, a different QoS requirement, a system fault and so on. The second step is to derive a new resource scheme based on the detected trigger. After that, the third step is the actual system reconfiguration. Figure 5.4 shows a high-level workflow of our SHARA framework. In this chapter, we assume that the trigger of the resource allocation events is the change of workload scenario on the target MPSoC system. At run time, the GM will continuously monitor the execution of workload scenarios on the target MPSoC system. When a new workload scenario is detected, the system will enter the resource reallocation stage. In this stage, the GM will try to redistribute the applications in the detected workload scenario according to the utilisation of each tile and the resource consumption of each application in the system. Based on the new workload distribution and the potential reconfiguration benefit, the global scheduler in the GM will start a global workload scheduling. In each tile, if the workload (applications) allocated by the GM is different with its previous workload, the local scheduler of the LM will adaptively reconfigure the hardware resources based on the mapping optimised in the LM and the corresponding reconfiguration benefit.

In this chapter, we mainly focus on the last two steps of dynamic resource reallocation on our target MPSoC system as mentioned above. To derive a new

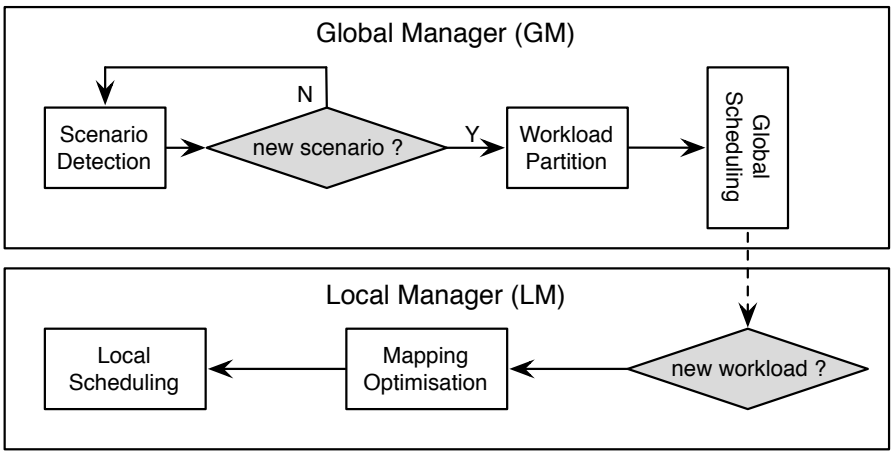


Figure 5.4: A high-level workflow of SHARA framework

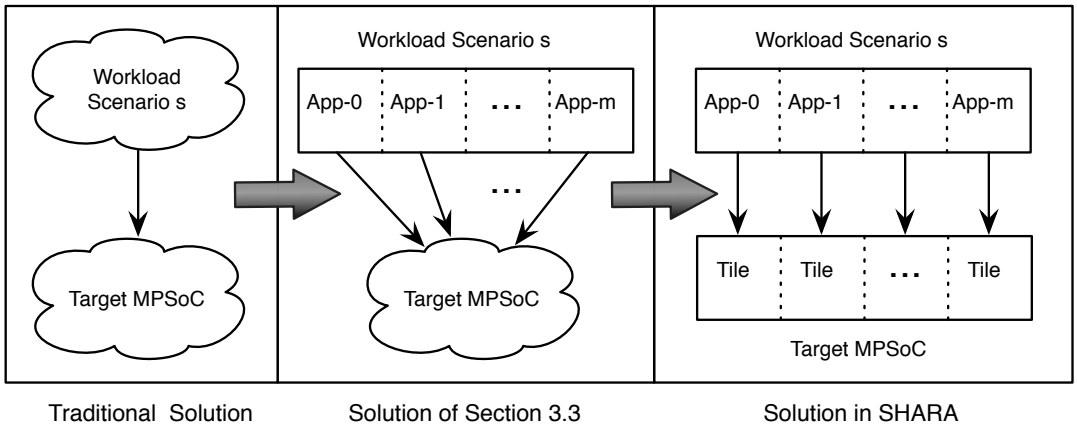


Figure 5.5: Divide-and-conquer solution for complex task mapping problem

mapping for a workload scenario on the target MPSoC system, we propose a scalable run-time task mapping approach which hierarchically maps the applications onto the tile-based MPSoC system. Normally, after deriving a new mapping for the new workload scenario, system reconfiguration should be done by the resource schedulers under the new generated mapping scheme. However, this is not always beneficial as the cost may overweight the benefit of such a system reconfiguration. If the schedulers still reconfigure the target system when it is not beneficial, this will lead to the problem of blind adaptivity as discussed in the previous chapter. To solve this problem on our large-scale MPSoC system, the *adaptivity throttling* technique proposed from the previous chapter is also adopted in the resource schedulers of the system which hierarchically allocate the system resources based on the reconfiguration decision predicted at different architecture levels (tile/processor level).

5.3.1 Scalable Run-time Task Mapping in SHARA

In our SHARA framework, we address the complex task mapping problem on our tile-based heterogeneous MPSoC system using the idea of the hybrid task mapping technique described in Section 3.3 which prepares *partial task mappings* for workload scenarios at design time and completes the mappings for the entire scenario at run time. This task mapping approach was proposed for a small-scale heterogeneous MPSoC system. It solves the scalability problem with regard to the number of tasks in the mapping problem. However, as mentioned before, the complexity of a task mapping problem not only depends on the number of application tasks but also on the number of target processing elements. In this chapter, we solve this problem by taking advantage of both the MPSoC architecture and its hierarchical control mechanism as shown in Figure 5.5. As the tiles in our target MPSoC system have the same architecture, compared with the approach used in Section 3.3, we can further simplify the design-time mapping optimising problem by considering only a partial target architecture (i.e., a tile) to limit the number of processing elements in the mapping problem. The problem of how to further optimise the entire mapping for the target workload scenario and the target MPSoC system will be solved at run time by light-weight heuristics. The details of our proposed task mapping approach will be explained in the following subsections.

5.3.2 Design-time Mapping Optimisation

At design time, the DSE approach of Section 4.2.2.1 is deployed for exploring performance optimal mappings at application level. Note that the target architecture in this mapping exploration process is the tile architecture inside the target MPSoC system. This greatly reduces the complexity of each single task mapping problem compared with exploring task mappings targeting the complete large MPSoC system. In this tile-based MPSoC architecture, there is no need to redo the design time mapping optimisation with the scaling of the target MPSoC system (i.e. a larger number of same tiles in the target MPSoC).

Figure 5.6 shows the mapping of an application on the tile architecture we considered in our MPSoC system. At design time, the mappings that need to be explored are expressed as what is shown in the mid-part of Figure 5.6. These mappings will be stored in the local memory of the GM. Beside the performance optimised mapping for each execution mode of each isolated application, the execution time of each task on each processor in a tile, the communication time between tasks on different communication channels of the target system and the migrating data size between processors for each task should also be analysed at design time and stored on the target system for mapping optimisation and adaptivity throttling.

5.3.3 Scalable Run-time Task Mapping

5.3.3.1 Tile-level Workload Partition

The mappings prepared at design time are optimised targeting the hardware resources inside a tile. To fully utilise the resources of our target MPSoC system where multiple identical tiles are present, the application level parallelism in a

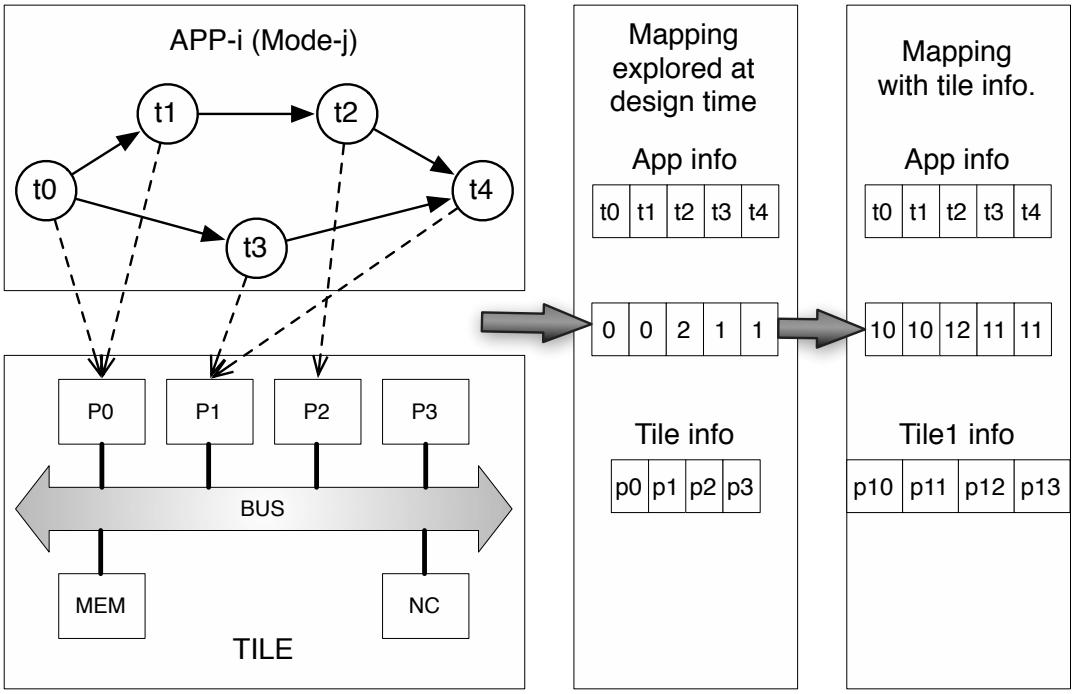


Figure 5.6: A simple example of encoded task mapping for the tile-based MPSoC

workload scenario will be addressed in the GM by means of workload partition. When the workload scenario on the target system changes, the mapping of applications (application to tile) in this new scenario may be adjusted by using a load balancing heuristic as shown in Algorithm 7. It means that the workload partition is triggered by the change of the workload scenario on the target MPSoC system. For a newly detected workload scenario, the utilisation of each tile will be calculated using the application/system information in the function of line 2 based on the current/old mapping on the system. The actual workload partition process starts from line 5 to line 14 in Algorithm 7. In each iteration of this process, if the maximal resource usage among tiles can be reduced, the application with smallest resource consumption (line 8) on the tile with maximal resource utilisation will be reallocated onto the tile with minimal resource utilisation. It means that the algorithm tries to gradually balance the system by migrating applications from overloaded tiles to lightly-loaded tiles. When an application is reallocated to a different tile, its pre-optimised mapping will be used on the new tile as shown in line 9 of Algorithm 7. This process will continue until the workloads on the system are well balanced. As the task migration overhead will greatly influence the system performance as can be seen in the experiment section, this algorithm tries to balance the system workload with a minimal number of task migrations among tiles to reduce the tile-level task migration overhead.

5.3.3.2 Processor-level Task Mapping Optimisation

After the entire new workload scenario is reallocated by the GM, each tile on the target MPSoC system might need to execute a new tile-level scenario. In the workload partition process, it only focuses on the total resource consumption

Algorithm 7 The heuristic of workload partition in the GM

Input: (μ_{old},η_{old}), $MPSoC$
Output: (μ_{new},η_{new})
1: $(\mu_{new},\eta_{new}) = (\mu_{old},\eta_{old})$
2: U = tileUsage((μ_{new},η_{new}), $MPSoC$);
3: $U_{max'} = \max(U)$; $U_{max} = +\infty$;
4: $U_{min'} = U_{min} = \min(U)$;
5: while $U_{max'} < U_{max}$:
6: $U_{max} = U_{max'}$
7: $U_{min} = U_{min'}$
8: app_{umin} = getMinApp(tile with U_{max})
9: (μ^*,η^*) = getOptMapping(app_{umin});
10: (μ_{app},η_{app}) = app2Tile((μ^*,η^*), tile with U_{min});
11: (μ_{new},η_{new}) = change((μ_{app},η_{app}), (μ_{new},η_{new}));
12: U = tileUsage((μ_{new},η_{new}), $MPSoC$);
13: $U_{max'} = \max(U)$;
14: $U_{min'} = \min(U)$;
* recover to the mapping that satisfies the condition in line 5 *\
15: recover2Prev((μ_{new},η_{new}));
16: return (μ_{new},η_{new});

of a complete application. The task mapping of an application on the resources inside a tile is either generated from the pre-optimised mappings stored on the system or the mapping preserved from the previous workload scenario. However, simply merging per-application mappings might not good enough with regard to the optimising goal like the performance objective considered in this chapter. After the GM finished the workload scheduling, the LM in the tile where the new workload has to be executed will further optimise the task mapping derived from the workload partition heuristic in the GM. In each LM, we again adopt the EIM algorithm proposed in Section 3.3 without considering the energy constraint which generates mappings with good quality in system throughput to further optimise the mapping for the new workload.

By using the scalable run-time task mapping approach in SHARA framework, Figure 5.7 shows a simple example of mapping a workload scenario onto our target MPSoC system. In this example, a workload scenario with five applications is mapped on the empty tile-based MPSoC system. When the GM in SHARA detects the new workload scenario, it will allocate these new applications onto tiles available in the target system by the tile-level workload partition algorithm. After that, each LM starts to further optimise the mapping of applications that are allocated on the corresponding tile. As only one application is active on $tile0$ to $tile2$, the LM on these three tiles will not further optimise the default mapping as it has already been optimised at design time. However, the mapping of applications on $tile3$ is further optimised by the EIM algorithm to improve the mapping quality.

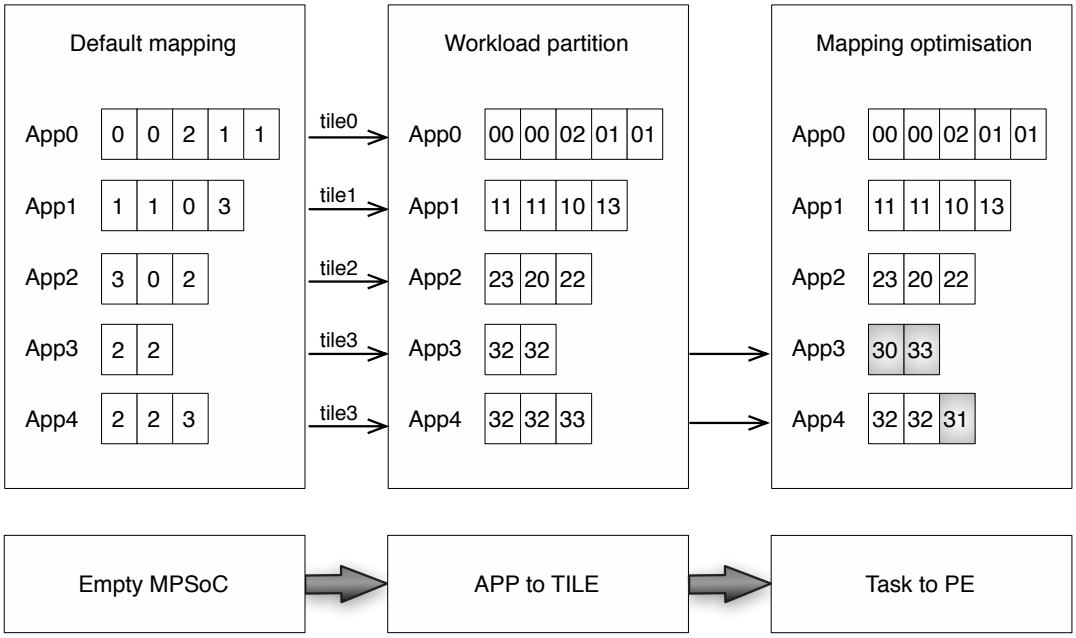


Figure 5.7: A simple example of mapping a workload scenario on the target MPSoC system

5.3.4 Adaptivity Throttling for System Reconfiguration

To avoid the blind adaptivity problem discussed at the beginning of this chapter for our tile-based MPSoC system, the adaptivity throttling technique from the previous chapter is adopted in the schedulers of our target system. However, different with the adaptivity throttling of previous chapter, the system reconfiguration could happen at two different levels (tile level and processor level) on the target MPSoC of this chapter. Consequently, the three parameters: the performance improvement $p - p'$, the reconfiguration cost c and the workload execution duration n for adaptivity throttling at each architecture level should also be predicted from the corresponding architecture level.

5.3.4.1 Mapping Performance Prediction

Our target MPSoC system consists of several identical tiles. Each tile has a typical heterogeneous MPSoC architecture. For this kind of system, we can apply the same performance analytic model to the different tiles in our target MPSoC. The LM of each tile has an instance of the performance model. In this work, we use the simple linear analytic model adopted in the SARA framework of Section 4.3.2.3 to predict the performance of different task mappings for workload scenarios of each separate tile. According to our hierarchical task mapping approach, the tasks of an application will be mapped to the same tile to reduce the communication overhead. It means that each tile can separately process its workload (no data dependence between tiles). Consequently, the performance (in term of scenario frame execution time) of the complete tile-based MPSoC system can be calculated by the GM using the following equation.

$$p_{sys} = max(p_{tl_k}) \tag{5.1}$$

where p_{tl_k} is the predicted execution time of the LM inside the tile tl_k.

5.3.4.2 Reconfiguration Cost Prediction

To predict the reconfiguration cost, a similar prediction model of the SARA framework as introduced in Section 4.3.2.4 can be applied in the GM and each LM of our tile-based MPSoC system. Notice that, in our MPSoC system, there are two levels of system reconfiguration: the tile-level system reconfiguration and processor-level system reconfiguration. Consequently, to calculate the cost of tile-level system reconfiguration, the computational overhead in the GM (derived by measurements), the total data size of inter-tile task migration and the data transferring speed via the NoC are required (the last two parameters are prepared at design time and stored on the system as mentioned in Section 5.3.2). Similarly, for processor-level system reconfiguration cost prediction, the computational overhead of the LM, the intra-tile task migration data size and the memory access speed inside a tile are required.

5.3.4.3 Scenario Duration Prediction

For the purpose of scenario duration prediction, we use the scenario execution history information to predict the future execution behaviour of workload scenarios. However, in this chapter, different with the approach used in the SARA framework of Section 4.3.2.5, we directly use the average scenario duration of previous executions of a workload scenario as the future execution duration value. The reason of considering such a simple prediction model is based on the following facts. In this chapter, we assume a large number workload scenarios will be executed on the target system. If the previously discussed predictors like table-based predictors and (A)SMM predictors are adopted, the memory usage will be a big concern on our MPSoC system as mentioned in Section 4.2.3.2. Besides that, from the adaptivity throttling experiments of the previous chapter, we observed that the performance of our adaptive resource scheduler is highly dependent on the average scenario execution duration. Therefore, such a simple prediction model is a proper choice for our initial study on large-scale MPSoC systems with a large number of workload scenarios. This average scenario duration information will be updated by the GM after the workload scenario actually finished its execution. This simple scenario duration predictor is initialised in the GM. When a new workload scenario is detected, the GM will predict a scenario duration for tile-level adaptivity throttling and send this predicted value to each LM for processor-level adaptivity throttling.

5.3.4.4 Hierarchical Adaptivity Throttling in SHARA

Using the introduced adaptivity throttling mechanism, a hierarchical scheduling policy is implemented in the SHARA framework where the GM actually schedules the system resources at tile level for new workload scenarios based on the tile-level reconfiguration decision and each LM schedules the resources inside the tile

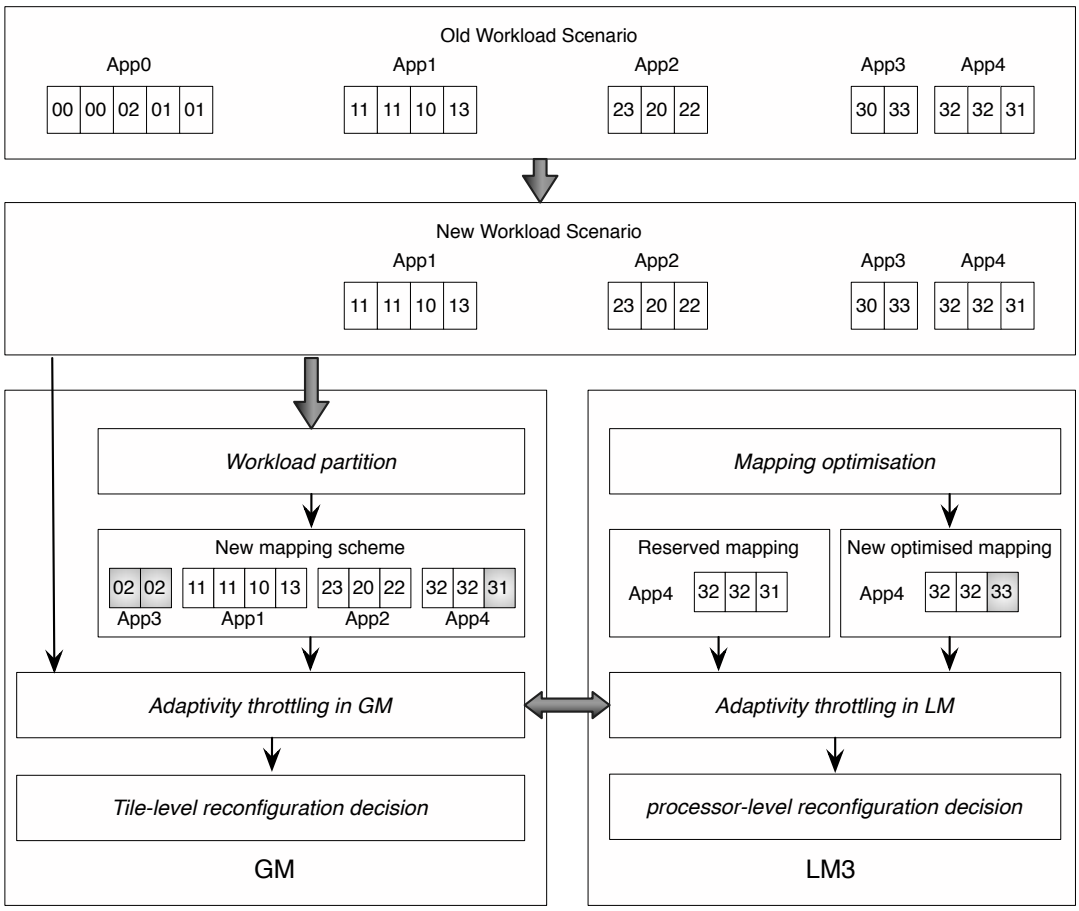


Figure 5.8: Hierarchical adaptivity throttling in SHARA

according to processor-level reconfiguration decision. Figure 5.8 shows how this hierarchical adaptivity throttling approach works in our SHARA framework. To derive a tile-level reconfiguration decision using the adaptivity throttling mechanism in the GM, those parameters needed for predicting the reconfiguration benefits should target the whole complete workload scenario on the target system. The performance improvement prediction in the GM happens after a new mapping is generated by the workload partition algorithm. The GM will firstly send the corresponding mapping information to the LM in each tile. After each LM predicted the performance in that tile, it will send back the performance information to the GM. The system performance of a whole workload scenario is then determined by the GM using Equation 5.1. The tile-level reconfiguration cost depends on both the computational overhead in the GM and the task migration cost between tiles via the NoC. In the example of Figure 5.8, the possible task migration cost concerns migrating application $App3$ from $tile3$ to $tile0$. In each LM, the performance prediction only focuses on the workloads that are allocated to the tile by the GM and the reconfiguration cost includes the computational overhead in itself and the possible task migrations (like the third task of $App4$) inside a tile via the local shared bus. With regard to the scenario execution duration prediction, the GM and each LM use the same prediction.

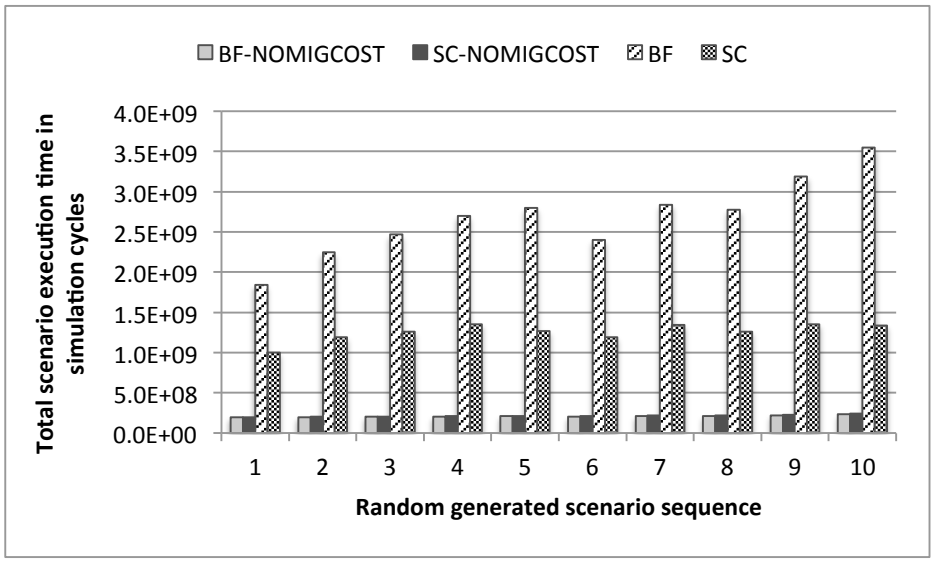


Figure 5.9: Comparison of two load balancing algorithm for tile-level workload partition

5.4 Experiments

In our experiments, we would like to show how our SHARA framework improves the system performance by applying the hierarchical task mapping and adaptivity throttling. We simulate the tile-based MPSoC integrated with our SHARA framework by the migration enabled Sesame simulator (see Section 4.1) similar to the experiments of the previous chapter. With regard to the target applications, we use 16 synthetic streaming applications with each application containing only 1 execution mode. In this case, the total number of workload scenarios is 65535 ($2^{16}-1$). The number of tasks in each application ranges from 4 to 8. We assume that each task can be executed on each processor of the target MPSoC using the corresponding pre-compiled code. The task execution time and migration data size of each task on each processor have been randomly generated and range between 1,000 and 100,000 time units (simulation cycles) and between 5K and 50K Bytes respectively. Communications between tasks range from 100 to 10,000 Bytes in size. In our experiments, we assume that all target applications are firstly loaded onto the same tile ($tile0$) with their pre-optimised mappings as an initial state of the system.

As introduced in the Section 5.3.3.1, the algorithm used in our GM (we refer to it as SC in this experiment) for tile-level workload partition tries to balance the workload among tiles with minimal inter-tile task migration. In the first experiment, we compare it with a load balancing algorithm (noted as BF) similar to the FFBP algorithm introduced in Section 3.2.5.2 to show the effect of our algorithm for reducing the task migration cost while achieving a well balanced system at tile level. In this BF algorithm, the active applications of the new workload scenario will be sorted in resource consumption descending order. And then the applications under this descending order will be gradually allocated to the tile with minimal resource utilisation (the resource utilisation of tiles will be recalculated after each allocation) under the pre-optimised mapping. In this experiment, we

do not consider a further mapping optimisation in the LM of each tile and also the adaptivity throttling ability of our SHARA has been deactivated. It means that the system will only optimise the mapping of a newly detected workload scenario by workload partition in the GM. After that, the system will be reconfigured based on the new mapping to execute the new workload scenario. We randomly generate 10000 workload scenarios and each workload scenario only execute for 1 $scenarioframe$ as a scenario sequence. Figure 5.9 gives the results of executing 10 such scenario sequences under these two algorithms where $BF - NOMIGCOST$ and $SC - NOMIGCOST$ represent the results without considering the tile-level system reconfiguration cost by using BF and SC respectively. From the results, we can clearly see that our algorithm has similar performance compared with BF if we ignore the reconfiguration cost. However, when the cost of system reconfiguration is taken into consideration, our algorithm performs much better than BF as the BF algorithm does not take the previous position of each application into account for workload distribution.

After investigating the mapping quality and tile-level reconfiguration cost by applying the global task mapping optimisation in the GM, we further study the hierarchical mapping optimisation approach of our SHARA framework in the second experiment. In this experiment, we still ignore the adaptivity throttling ability of our framework (task migration happens when the newly derived mapping is different with the old mapping). We select two workload scenarios $S16$ and $S4$ as our target scenarios to show how the scenario execution time is influenced by system reconfiguration. The scenario $S16$, in which all the target applications are active, is the most complex workload scenario of all possible scenarios. And $S4$ is a scenario where only four applications are active. In this experiment, our hierarchical mapping optimisation approach (as it contains two steps of mapping optimisation in the GM and LMs, here we label it as $GM - LM$) is compared to three other approaches $NGM - NLM$, $NGM - LM$ and $GM - NLM$. The $NGM - NLM$ approach does not contain any mapping optimisation process. It means that, in this approach, the mapping in the initial state of the target system is directly used for executing the target two workload scenarios. The $GM - NLM$ and $NGM - LM$ only considers the tile-level and process-level mapping optimisation for the target scenarios from the initial system state respectively. In this experiment, we assume that the system will be triggered for reconfiguration when the first workload scenario is detected on the system. For the target two workload scenarios $S16$ and $S4$, they are separately executed for a single *scenario frame* directly from the initial system state (all target applications are loaded onto $tile0$). The results of this experiment are illustrated in Figure 5.10. In this figure, the x-axis represents different states of the two target scenarios where for example $S16 - NOCOST$ and $S16 - COST$ are executing the scenario $S16$ without and with considering all system reconfiguration cost (both tile-level and process-level) respectively. Here, the results of $* - NOCOST$ is derived by directly executing the corresponding scenario for a single frame under the mapping optimised by the various approaches.

If we only consider the quality of the mapping ($* - NOCOST$) derived from different approaches, from the experimental results, we notice that the mapping optimisation in the GM is more important compared with the optimisation in the LMs. Compared to the approach of $NGM - NLM$, the other three approaches

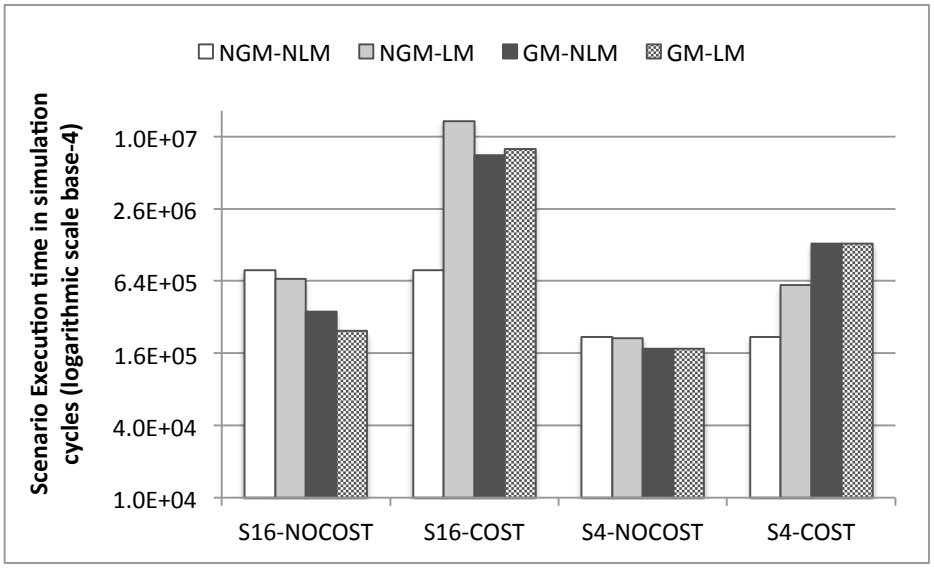


Figure 5.10: Performance comparison of different task optmisation approach

$NGM - LM$, $GM - NLM$ and $GM - LM$ improve the scenario performance by 17%, 121% and 220% respectively in $S16 - NOCOST$ and 3%, 26% and 26% in $S4 - NOCOST$. In the complex scenario case $S16$, the GM and LMs are able to greatly improve the mapping quality. However, when the scenario is relatively simple like $S4$ where the resource contention is not critical, the performance improvement is not that apparent anymore especially the improvement from the optimisation by the LMs. When taking the system reconfiguration cost into consideration, we can see from the results shown in Figure 5.10 that the system reconfiguration cost which contains both the task migration cost and the computational overhead in the GM and LMs will dominate the execution time of scenarios if the scenario duration (number of $scenarioframes$) is very short. In our test cases, as we set the execution duration of each scenario to one $scenarioframe$, consequently the final performance (reconfiguration cost included) of $NGM - NLM$ is much better than the other three approaches. To further understand where the system reconfiguration cost comes from, we zoom into the scenario execution time of $S16 - COST$ and $S4 - COST$ in Figure 5.11. In this figure, the symbols of EXE, OGM, OGC, OLM and OLC respectively represent the actual execution time of the target scenario under the mapping optimised by the corresponding approach, the overhead of inter-tile (global) task migration, the computational overhead of the GM, the overhead of intra-tile (local) task migration and the computational overhead of LMs. Note that, as the LMs in our system work in parallel, the OLM and the OLC come from the tile where the intra-tile task migration cost and the computational overhead in the LM in total is the maximal among tiles. From Figure 5.11, we clearly see that when the GM takes part in the mapping optimisation process, the system reconfiguration cost mainly comes from the task migration between tiles. Considering the processor-level system reconfiguration, the overhead is dominated by the computational cost in LMs especially when the number of tasks that are allocated to the tile is very large. The reason behind that can be explained as follows. In our experiment, the mapping used for further optimisation in each tile is merged from the pre-optimised mapping of each

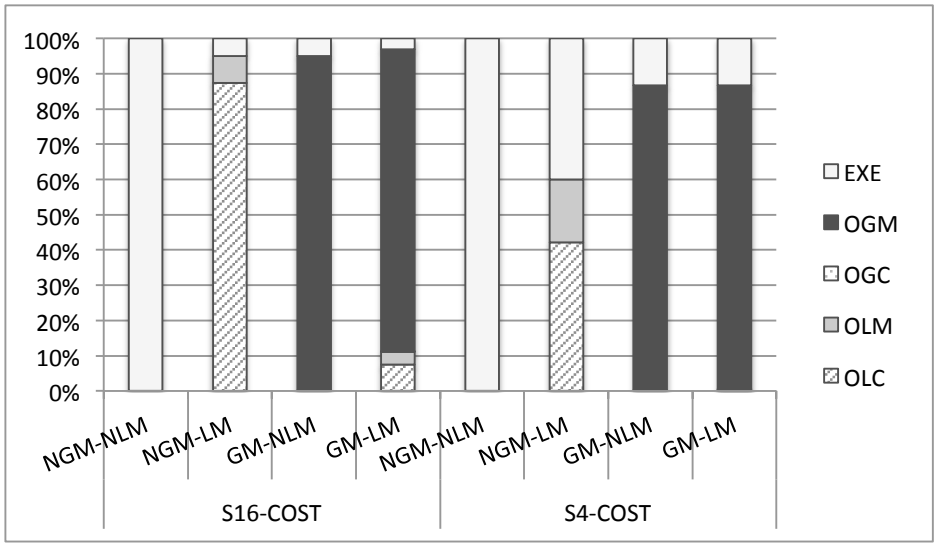


Figure 5.11: System reconfiguration cost in $S16-COST$ and $S4-COST$

application. This original mapping normally is already well balanced among processors in each tile. The algorithm used in each LM will take a relatively large time to further improve the mapping quality with only a few tasks that need to be migrated among processors.

From the second experiment, we can see that if the scenario execution duration is very short, the system should not be reconfigured as the large system reconfiguration cost will neutralize the performance improvement by run-time task mapping optimisation. Consequently, in the third experiment, we would like to show how the system performance is influenced by the scenario execution duration in our target large-scale MPSoC system. For this purpose, we investigate the workload scenario $S16$ of the second experiment with a gradually increasing scenario execution duration. In this experiment, we use the complete SHARA framework (the adaptivity throttling is enabled) for run-time resource allocation and compare it with the three approaches $NGM-NLM$, $GM-NLM$ and $GM-LM$ considered in the second experiment. Figure 5.12 shows the total execution time including the system reconfiguration cost of different scenario durations under different resource allocation approaches. Clearly, as the $NGM-NLM$ does not need application remapping, the total execution time increases linearly with the scenario duration (in $scenarioframes$). Similar behaviour can be found in $GM-NLM$ and $GM-LM$. However, as the system is reconfigured according to the corresponding optimised mapping at the beginning of the scenario $S16$, the total execution time has a slower increase with the scenario duration under these two approaches compared to $NGM-NLM$. As the mapping quality derived by $GM-LM$ is better than the one derived by $GM-NLM$, the total execution time of the former approach has an even slower increase with scenario duration. From these three approaches, we can see that the $NGM-NLM$ has the best performance when the scenario duration is small (for example, under 15 scenario frames in our test case) as it avoids the system reconfiguration cost. However, with the increase of scenario execution duration, it is increasingly outperformed by $GM-NLM$ and $GM-LM$.

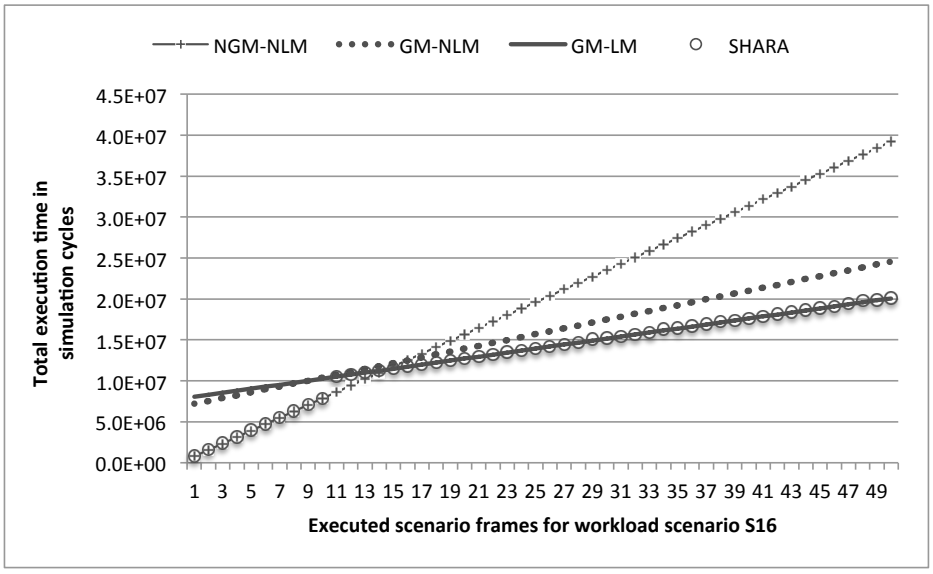


Figure 5.12: The influence of scenario duration to final system performance using different approaches

By using the adaptivity throttling ability of our SHARA framework, we are able to solve the drawback of the other approaches. When the scenario execution duration is small, the system will be kept unchanged to avoid unnecessary system reconfigurations. On the other hand, when the scenario execution duration is large, the system will be reconfigured to the mapping optimised by SHARA. In this experiment, the scenario duration predictor has been deactivated[1] to exclude its influence for deriving a reconfiguration decision which will be further studied in the next experiment. The results of using our complete SHARA framework shown in Figure 5.12 verify the ability of improving the system performance by our hierarchical adaptivity throttling approach on the target large-scale MPSoC system. When the scenario duration of scenario $S16$ is lower than 11, $SHARA$ is very close to $NGM - NLM$. After that, it is very close to $GM - LM$. This also reflects the fact that the overhead of adaptivity throttling is small enough to be ignored. However, notice that, from 11 to 15 in the x-axis of Figure 5.12, the results of $SHARA$ are close to $GM - LM$. If the prediction models used for adaptivity throttling in SHARA are absolutely accurate, these points should have been close to $NGM - NLM$. This problem is mainly caused by the accuracy of the mapping performance predictor and the migration cost predictor.

In the fourth experiment, we apply our proposed SHARA framework in more complex scenario cases on the target MPSoC system to test its performance when scenario duration prediction is considered and compare the results with two approaches $STATIC$ and again $GMLM$. In the $STATIC$ approach, all applications are statically mapped (i.e., no run-time mapping takes place) using a mapping which has shown to be optimal *on average* for all possible workload scenarios. The $GMLM$ is similar to a normally used approach in small scale MPSoCs where the system will always be reconfigured based on the optimised mapping when a

[1] We directly use the actual execution duration of the target workload scenario for reconfiguration prediction.

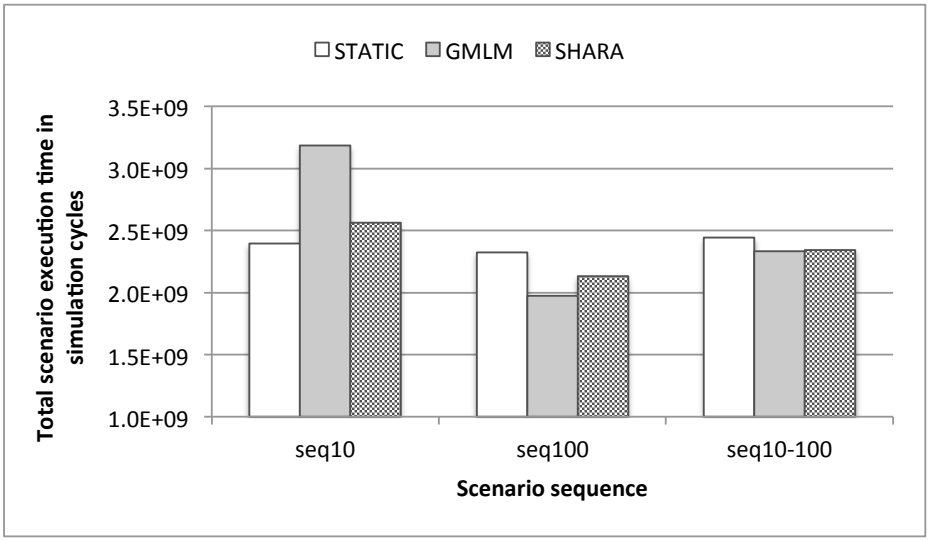


Figure 5.13: Performance of SHARA in more complex scenario cases

new workload scenario is detected. To model dynamic application behaviour over time (e.g. due to user behaviour), we generate three kinds of workload scenario sequences. Each sequence is generated in two steps. The first step is to randomly choose a workload scenario from all the possible workload scenarios. For each selected workload scenario, it will appear in the scenario sequence for multiple times. The second step is to generate the duration in *scenario frames* for each appearance of the selected workload scenario. In this experiment, we model totally random user behaviour to show the performance of our approach in extreme cases. For this purpose, the scenario duration is generated by a random generator with a certain average scenario duration (number of $scenario frames$). This process iterates until a pre-defined total frame number (10,000 frames in our case) has been achieved for each scenario sequence. Our three target scenario sequences $seq10$, $seq100$ and $seq10-100$ in Figure 5.13 are distinguished by the average number of scenario frames set for the random scenario duration generator where an average frame number of 10, 100 and 10 to 100 (the average frame number set for the generator in each iteration is also randomly derived from 10 to 100) are used for generating the three kinds of sequence respectively.

The results of each kind of scenario sequence shown in Figure 5.13 are averaged over five randomly generated different sequences. From this figure, we can see that our $SHARA$ approach has a good trade off between $STATIC$ and $GMLM$. When the average number of scenario frames for each workload scenario is small like $seq10$, the $GMLM$ approach where a system reconfiguration alway happens when a new scenario is detected has the worst performance. However, it has the best performance when the average scenario frame is large like $seq100$ as in this case the system reconfiguration cost is covered by the performance improvement because of system reconfiguration. If the average number of scenario frames becomes even larger, the gap between $STATIC$ and $GMLM$, $SHARA$ will also increase. Comparing $SHARA$ with $GMLM$, in the case of $seq100$, our $SHARA$ approach suffers from both the reconfiguration prediction overhead and error system reconfiguration prediction which mainly caused by the prediction error in

mapping performance and task migration cost as mentioned in the third experiment. In the case of $seq10-100$, the scenario duration predictor also influences the prediction of system reconfiguration. However, as the system performance degradation caused by errors in the system reconfiguration prediction in $SHARA$ is almost equal to the unnecessary reconfiguration overhead in $GMLM$ when the average scenario duration is short, our $SHARA$ shows a similar performance with $GMLM$. The problem of how to improve the system performance by optimising the prediction accuracy of our predictors used for adaptivity throttling will be further studied in future work.

Regarding to the run-time system storage consumption of our SHARA framework, several assumptions should be mentioned. On our target MPSoC system, we store all the design-time prepared information in the local memory of the GM. For storing the pre-optimised mappings, we assume that the mapping information of each task and each communication channel between tasks is stored in one *byte*. In our target synthetic streaming applications, there are 88 tasks and 67 communication channels in total. Consequently, to store the pre-optimised mappings, the memory usage is 155 *bytes*. Beside the pre-optimised mappings, in our $SHARA$ framework, we also need to store the application/system information and the average scenario execution history information. Here, we assume that each piece of this information needs one *word* of memory. Consequently, for storing the application/system information, 1408 *bytes* of memory are required. With regard to the average scenario execution history information, as each workload scenario needs one *word* to store the information, the total memory usage is 256 KB.

5.5 Related Research

There are a lot of task mapping approaches for improving the adaptivity of small scale MPSoC systems with only a certain number of scenarios or applications that need to be supported. But most of them still lack scalability when the task mapping problem becomes complex in large-scale systems with a large number of applications and processing elements. To solve this problem, several distributed resource management approach for large-scale MPSoC systems or many-core systems have been proposed like the work in [44, 5]. In [44], the authors proposed a new concept - invasive computing - for resource management on a heterogeneous, tile-based manycore system. This invasive computing technique uses a multi-agent management layer underpinned by distributed runtime and OS services to support a flexible resource management. The agent of each application executing in the system tries to increase the speedup of its application by acquiring additional cores from the nearby regions. [5] presents a scheme for run- time application mapping in a distributed manner using agents targeting adaptive NoC-based heterogeneous multi-processor systems. Compared to these approaches, our approach uses a hierarchical resource management approach and explicitly studies the influence of system reconfiguration for run-time resource allocation. Recently, [105] also proposed a scenario-based run-time mapping approach for many-core systems which is very similar to our work. In their approach, the execution scenarios are combined into a finite state machine and the transitions between scenarios are limited in the pre-determined states. However, we do not have such kind of limitations and consequently more complex run-time situations can be considered in our work.

5.6 Conclusion

In this chapter, we proposed a scenario-based hierarchical run-time adaptive resource allocation framework to increase the adaptivity of large-scale heterogeneous MPSoC systems where a large number of scenarios or applications need to be supported. The SHARA framework adopts a hierarchical resource allocation mechanism to reduce the complexity of the task mapping problem at run time. In this framework, the system resources are allocated as tiles which could be either real tiles in a tiled system or virtual tiles virtually divided in a system by the global manager. Inside each tile, the hardware resources will be allocated to the workload active on it by the corresponding local manager. For a new workload scenario, after deriving a new mapping by the hierarchical task mapping approach, the hierarchical adaptivity throttling technique will be applied for actual system reconfiguration based on the scenario execution history behaviour. It is helpful to avoid unnecessary system reconfiguration in the case when the reconfiguration is not beneficial. By applying our SHARA framework on the target tile-based MPSoC system, the problem of scalability considering both the number of target workload scenarios and the number of processing elements in the target system, but also the flexibility and blind adaptivity problems of general hybrid task mapping approaches as mentioned in previous chapters are addressed.

CHAPTER 6

Conclusion and Future Work

In this chapter, we will conclude the work of this thesis and give a brief introduction of possible future work in the domain of dynamic resource management of MPSoC systems.

6.1 Conclusion

The technology improvement and the adoption of more and more complex applications in the embedded domain are causing a rapid increase in the complexity of embedded systems. The major solution for today's embedded systems, namely heterogeneous multiprocessor system-on-chip (MPSoC), is fuelled by the increasing demand of high-performance and low-power solutions in the embedded devices. These MPSoC systems are increasingly required to be adaptive at run time to support complex and dynamic application workloads, dynamic QoS, fault tolerance and so on. To achieve the ability of system adaptivity for MPSoCs, two kinds of techniques, namely dynamic hardware reconfiguration and dynamic software reconfiguration as introduced in the first chapter, have been proposed in recent years. Among those solutions, the approaches with dynamic application task remapping provide a good trade off between the hardware overhead and design complexity. In this thesis, we have focused our research on improving the adaptivity of MPSoC systems with complex workload behaviour by means of dynamic application task remapping.

The process of application task mapping plays a crucial role in exploiting the system properties such that applications can meet their, often diverse, demands on performance and energy efficiency. To cope with dynamic application behaviour, hybrid task mapping approaches has been widely studied recently. These hybrid strategies combine the design-time static task mapping with the run-time management in order to select mapping configurations that are best suited for the current workload scenario on the target system. However, most of these approaches still suffer from the issues that have been stated in Section 1.4 as the research questions of this thesis.

With regard to the first research question: *How to improve the efficiency of static task mapping exploration*, we solved this problem by pruning the search space of the target mapping problem with domain-specific heuristics. When the mapping solution space is very large in a static mapping optimisation problem, it is impossible to explore each solution at design time. In this case, a possible solution is using heuristics to guide the search process such that it only concentrates on parts of the solution space that may contain the optimal solution. By using this approach, in Chapter 2, we presented a GA-based mapping DSE algorithm (BEG) for design-time task mapping exploration with a single optimisation goal of maximising mapping performance. This algorithm is guided by a domain-specific heuristic to find the optimal mapping solution with maximal performance. With the help of such a heuristic, the proposed DSE algorithm only needs to evaluate the mapping candidates that have a higher chance to be the optimal solution. Consequently, it can solve large-scale static task mapping problems more efficiently. For multi-objective static mapping optimisation problems, we have deployed the NSGA-II-based DSE approach as introduced in Section 3.2.3. Similar to the BEG algorithm, this multi-objective DSE approach allows for effectively pruning the design space by only evaluating a representative subset of the target problem (chosen by the genetic operators). However, as the NSGA-II-based approach does not have a similar heuristic adopted in the BEG algorithm to derive good mapping candidates in the search process, the efficiency of NSGA-II is lower than BEG.

For the second research question: *How to achieve scalability with regard to the number of workload scenarios as well as flexibility in hybrid task mapping techniques*, two different hybrid scenario-based task mapping approaches have been presented in Chapter 3. The scenario-clustering based task mapping approach proposed in Section 3.2 solves the scalability issues of general hybrid task mapping techniques when large number of workload scenarios need to be supported on the target system. In this approach, the target workload scenarios are firstly divided into different scenario clusters. For each separate scenario cluster, a mapping that on average performs best for all scenarios inside the scenario cluster is explored at design-time and stored on the target system for run-time usage. More specifically, in Section 3.2, we distinguished each inter-application scenario as a separate scenario cluster. Inside each inter-application scenario, there are multiple intra-application scenarios caused by the change of execution modes of each active application. At run time, a light-weight system resource scheduler applies the proposed STM algorithm to perform mapping initialisation and customisation according to the pre-stored cluster-level mapping information. The mapping initialisation process is triggered by the change of inter-application scenarios on the target system and the mapping customisation process is triggered by a violation of application-specific performance objectives.

This scenario-clustering based task mapping approach still cannot solve the flexibility issue for supporting new applications on the target system. In this approach, when a new application needs to be supported, design-time analysis needs to be redone for each possible new inter-application scenario. In Section 3.3, we therefore presented a novel hybrid task mapping approach that uses a divide-and-conquer method to solve both the scalability and flexibility issues of general hybrid task mapping techniques. In this approach, the scenario-level task mapping problem is broken down into application-level task mapping problems at design

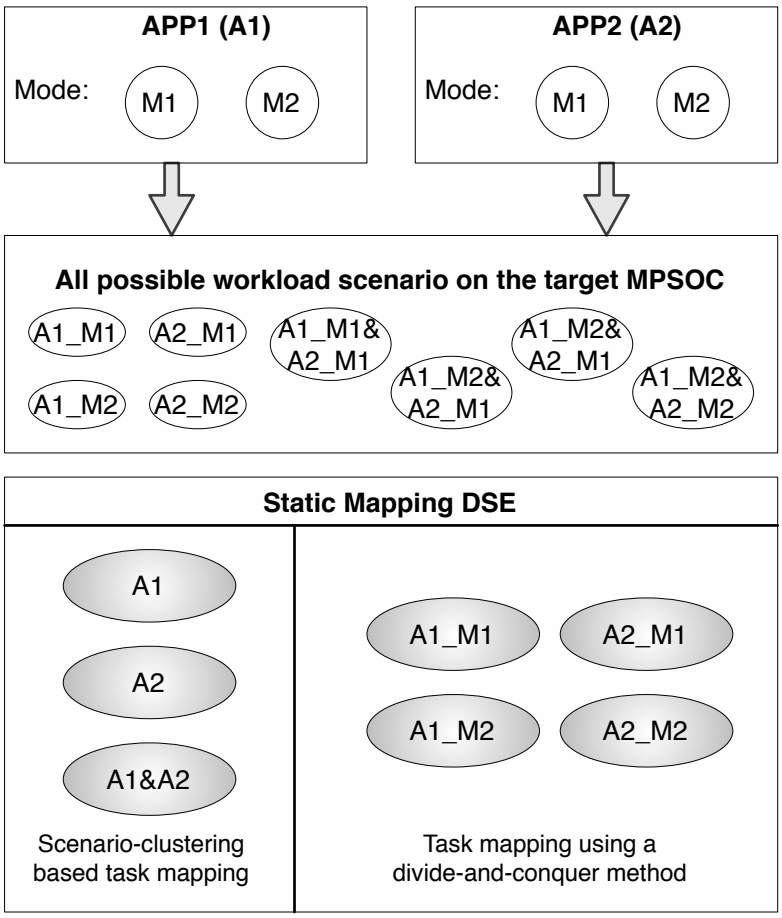


Figure 6.1: Design-time target mapping problems of the two proposed hybrid task mapping approaches

time, and the application-level mapping solutions are then dynamically combined and further optimised to give a complete solution for a workload scenario at run time. By applying this method, we only need to do static mapping optimisation for each application in isolation at design time. Consequently, when a new application needs to be added on the target system, only a small effort of static mapping DSE for this single application needs to be done. At run-time, the EIM algorithm takes charge of the mapping optimisation for the active workload scenario. The mapping optimisation process is triggered by the change of workload scenarios.

Figure 6.1 illustrates the target mapping optimisation problems that need to be solved at design time for the target applications using these two hybrid task mapping approaches. In both cases, the number of mappings that needs to be explored and consequently the memory usage for storing the pre-optimised mappings on the target system can clearly be reduced compared to general hybrid task mapping techniques where design-time analysis should be done for each possible workload scenario. To support n applications where each application has m execution modes, the total number of possible workload scenarios on the target system is $(m+1)^n - 1$. By using our scenario-clustering task mapping approach and the one with a divide-and-conquer method, the total number of mappings that needs to be explored and stored is $2^n - 1$ and $m * n$ respectively. This means that the

scenario-clustering task mapping approach has a better scalability in the case of a small number of applications but where each application contains a large number of execution modes. Otherwise, the divide-and-conquer task mapping technique will be better. These two proposed hybrid task mapping approaches not only solve the scalability and flexibility problems of general hybrid task mapping approaches that explore task mappings for all the target workload scenarios at design time but also produce similar mapping results like those from general hybrid task mapping approaches.

To solve the third research question: *How to deal with blind adaptivity at run time for an adaptive MPSoC system*, an adaptivity throttling technique has been proposed in Chapter 4. The blind adaptivity problem of an adaptive MPSoC system is caused by run-time system reconfiguration costs. When a reconfiguration is triggered on a MPSoC system, the system reconfiguration should actually not happen if the reconfiguration overhead outweighs its benefit (performance and/or energy consumption). However, in state-of-the-art hybrid task mapping techniques, this problem has not been explicitly studied. In Chapter 4, we presented our solution for the blind adaptivity problem by applying adaptivity throttling in the resource scheduler of the target MPSoC system to improve the system efficiency. By using this technique, the system scheduler can predict (using several analytic prediction models) whether or not reconfiguration of the system actually is beneficial based on the active workload scenario and the status of the hardware platform. According to this prediction, unnecessary system reconfigurations can be avoided, and consequently the system efficiency can be improved.

This adaptivity throttling technique has been combined with a general hybrid task mapping approach where the pre-optimised mappings (derived by design-time DSE) for all the target workload scenarios can be directly used at run time and the divide-and-conquer task mapping approach of Section 3.3 to improve performance of a heterogeneous MPSoC system. Comparing the MPSoC system that uses a normal hybrid task mapping approach with adaptivity throttling to the MPSoC system using our divide-and-conquer task mapping approach with adaptivity throttling, the former one has a slightly better performance as its system reconfiguration is based on the design-time optimised mapping. However, the advantage of the latter system is that it solves not only the blind adaptivity problem but also the scalability and flexibility problem of such a general hybrid task mapping approach.

About the last research question stated in the first chapter: *Are hybrid task mapping techniques still applicable on future large-scale MPSoCs*, we presented our initial research in Chapter 5. For a task mapping problem, its complexity is related to both the number of target application tasks and processing elements in the target MPSoC system. When adopting traditional hybrid task mapping approaches on a large-scale MPSoC, two issues are obvious. Firstly, the design-time mapping exploration will become intractable even when our previously proposed techniques are considered. This is because the complexity of each single mapping problem is now dominated by the number of processing elements in the target system. Secondly, the mechanism for run time resource management is also problematic because centralised resource management approaches deployed in general hybrid task mapping approaches are not suitable for large-scale systems anymore. To solve these two issues, we proposed a solution that uses a similar divide-and-conquer

approach as considered in the hybrid task mapping techniques of Chapter 4. More specifically, we presented a tile based MPSoC architecture as the prototype of future large-scale heterogeneous MPSoC systems that contains multiple identical tiles where each tile contains several heterogeneous processing elements. For this MPSoC system, we made an assumption that an entire application can only be mapped to a single tile to reduce the communication overhead between tasks inside an application. Under this assumption, the task mapping problem is greatly simplified on our target large-scale MPSoC system. At design-time, we only need to solve the mapping problem targeting a single tile architecture. Consequently, the design-time DSE approaches presented in this thesis can directly be used to solve the task mapping problem of each tile.

At run time, for the target MPSoC platform, we studied a hierarchical resource management mechanism to overcome the performance bottleneck of centralised approaches and the complexity of distributed approaches. In this mechanism, the run-time task mapping process is divided into two levels where the tile-level resource allocation (across tiles) is handled by the global manager of the system and the processor-level resource allocation (inside each tile) is solved by a local manager. The tile-level resource management explores the application-level parallelism for the target applications by a load balancing heuristic. The processor-level resource allocation is done by the hybrid task mapping approach presented in Section 3.3 to explore the task-level parallelism. The actual resource reconfiguration on the target MPSoC system is controlled by the manager of each level using the adaptivity throttling technique. By applying our scenario-based hybrid task mapping approach and the adaptivity throttling technique on the target tile-based MPSoC system, the problem of scalability (with regard to the size of both the number of target applications and the target platform), flexibility and blind adaptivity can be solved. This hierarchical solution for our target tile-based MPSoC system is also applicable to a general large-scale MPSoC system. For example, considering a general NoC-based large-scale MPSoC, one can virtually divide the MPSoC into smaller blocks and then apply our approach to improve the system adaptivity. Therefore, for large-scale MPSoC systems with complex and dynamic application behaviour, our hybrid task mapping approaches are still useful for generating better mapping solutions compared to pure dynamic task mapping approaches normally adopted for large-scale systems and consequently improving the system efficiency.

To evaluate the proposed techniques described in this thesis, we deployed the Sesame simulation framework (see Section 2.1). It is a system-level modeling and simulation environment which allows for flexible evaluation of different applications, different underlying architectures, and different application-to-architecture mappings for MPSoC systems. This Sesame simulator was directly used for design-time mapping DSE in our work. However, the original Sesame is unable to support the simulation of dynamic application behaviour and run-time mapping optimisation/customisation. To enable this property, we have extended it with a run-time resource scheduling framework as introduced in Section 3.1. Using the extended Sesame simulator, we are able to study different mapping optimising techniques for MPSoC systems with complex and dynamic workload behaviour. The proposed approaches of Chapter 3 have been evaluated by this extended Sesame simulator. In the research work of Chapter 4 and 5, the impact of system reconfiguration

was taken into consideration for dynamic task remapping at run time. For this purpose, the Sesame simulation framework was further extended with the ability of run-time system reconfiguration cost evaluation to study system reconfiguration mechanisms and policies for adaptive MPSoCs.

Overall, this thesis presented our research for improving the adaptivity of MPSoC systems with dynamic and complex application behaviour using dynamic application task remapping. It can help embedded system designers to build an efficient MPSoC system that can adaptively cope with complex run-time behaviour of target applications. More specifically, our two extended Sesame simulators provide a highly flexible and efficient simulation environment for MPSoC designers to evaluate their possible adaptivity and run-time mapping solutions at the very early stage of design. The techniques for solving the scalability, flexibility and blind adaptivity issues of general hybrid task mapping approaches proposed in this thesis can be integrated in MPSoC products to improve the system adaptivity. Furthermore, our initial research on large-scale adaptive MPSoC system shows a possible solution for the design of further MPSoC systems.

6.2 Future Work

There are many potential directions for future research based on the work presented in this thesis. For example, considering the static multi-objective task mapping problem of Section 3.2.3, the NSGA-II-based approach still has an efficiency problem in complex mapping optimisation problems as general crossover and mutation operators were directly used without any optimisation like the genetic algorithm proposed in Section 2.3. To improve the efficiency of this DSE approach, further efforts such as providing domain-specific heuristics to optimise the genetic operators can be done. In our proposed scenario-clustering based task mapping approach, currently, only the performance constraints of target applications are considered as the trigger for run-time mapping customisation during the execution of a certain inter-application scenario. In the future, we may also consider some other triggers such as the violation of system power consumption or chip overheating for mapping customisation to satisfy different run-time execution requirements of the target MPSoC system. As mentioned in Section 4.2.3.2, the memory consumption of our history based scenario duration predictor (ASMM) will be an issue in the case a large number of workload scenarios need to be considered on the target system. To solve this problem, we could consider to implement this scenario execution duration predictor by other techniques like Neural Networks. In Section 5.4, we also mentioned the accuracy problem of predictors used for adaptivity throttling. This accuracy problem can lead to an erroneous system reconfiguration that apparently degrades the system performance. Therefore, how to improve the prediction accuracy for the predictors in our adaptivity throttling technique is also another future research direction.

Besides the above mentioned future research directions for optimising the work presented in this thesis, some other possible extension of our work are:

- Prototyping a real adaptive MPSoC system with our proposed dynamic task remapping techniques on e.g. an FPGA to further optimise and increase the practicability for our approaches. On a real system, more concrete problems

such as how the run-time scheduler should be integrated into the system, how to implement different task migration mechanism for task remapping and so on need to be carefully considered.

- In this thesis, only approaches for software reconfiguration (i.e., application task remapping) have been considered to improve the system adaptivity on MPSoC systems. As mentioned in the first chapter, dynamic hardware reconfiguration is also an option for adaptive MPSoC systems. With a hardware reconfigurable MPSoC, hardware reconfiguration approaches like dynamic voltage and frequency scaling can be combined with our software reconfiguration approaches to derive an even more efficient MPSoC system.

General bibliography

[1] A. Acquaviva, A. Alimonda, S. Carta, and M. Pittau. Assessing task migration impact on embedded soft real-time streaming multimedia applications. *EURASIP J. Embedded Syst.*, 2008:9:1–9:15, January 2008.

[2] Alexandra Aguiar, Sérgio J Filho, Tatiana G dos Santos, César Marcon, and Fabiano Hessel. Architectural support for task migration concerning mpsoc. *SBC*, page 169, 2008.

[3] W. Ahmed, M. Shafique, L. Bauer, and J. Henkel. mrts: Run-time system for reconfigurable processors with multi-grained instruction-set extensions. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[4] Yongjin Ahn, Keesung Han, Ganghee Lee, Hyunjik Song, Junhee Yoo, Kiyoung Choi, and Xingguang Feng. Socdal: System-on-chip design accelerator. *ACM Trans. Des. Autom. Electron. Syst.*, 13(1):17:1–17:38, February 2008.

[5] Mohammad Abdullah Al Faruque, Rudolf Krist, and Jörg Henkel. Adam: Run-time agent-based distributed application mapping for on-chip communication. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 760–765, New York, NY, USA, 2008. ACM.

[6] A. Alexandrescu, I. Agavriloaei, and M. Craus. A genetic algorithm for mapping tasks in heterogeneous computing systems. In *System Theory, Control, and Computing (ICSTCC), 2011 15th International Conference on*, pages 1–6, 2011.

[7] F. Angiolini, Jianjiang Ceng, R. Leupers, F. Ferrari, C. Ferri, and L. Benini. An integrated open framework for heterogeneous mpsoc design space exploration. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1–6, March 2006.

[8] Robert Armstrong, Debra Hensgen, and Taylor Kidd. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. In *In 7th IEEE Heterogeneous Computing Workshop (HCW '98*, pages 79–87, 1998.

[9] Yeshayahu Artsy and Raphael Finkel. Designing a process migration facility: The charlotte experience. *Computer*, 22(9):47–56, September 1989.

[10] Giuseppe Ascia, Vincenzo Catania, Alessandro G. Di Nuovo, Maurizio Palesi, and Davide Patti. Efficient design space exploration for application specific systems-on-a-chip. *Journal of Systems Architecture*, 53(10):733 – 750, 2007. Embedded Computer Systems: Architectures, Modeling, and Simulation.

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

[12] Daniel Barcelos, Eduardo Wenzel Brião, and Flávio Rech Wagner. A hybrid memory organization to enhance task migration and dynamic task allocation in noc-based mpsocs. In *Proceedings of the 20th Annual Conference on Integrated Circuits and Systems Design*, SBCCI '07, pages 282–287, New York, NY, USA, 2007. ACM.

[13] L. Bauer, M. Shafique, and J. Henkel. Rispp: A run-time adaptive reconfigurable embedded processor. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 725–726, Aug 2009.

[14] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali. Supporting task migration in multi-processor systems-on-chip: A feasibility study. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1–6, March 2006.

[15] A. Bonfietti, L. Benini, M. Lombardi, and M. Milano. An efficient and complete approach for throughput-maximal sdf allocation and scheduling on multi-core platforms. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 897–902, March 2010.

[16] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Bölöni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, and Richard F Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810 – 837, 2001.

[17] Eduardo Wenzel Brião, Daniel Barcelos, and Flávio Rech Wagner. Dynamic task allocation strategies in mpsoc for soft real-time applications. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 1386–1389, New York, NY, USA, 2008. ACM.

[18] Eduardo Wenzel Briao, D. Barcelos, Fabio Wronski, and F.R. Wagner. Impact of task migration in noc-based mpsocs for soft real-time applications. In *Very Large Scale Integration, 2007. VLSI - SoC 2007. IFIP International Conference on*, pages 296–299, Oct 2007.

[19] P.P. Bungale, S. Sridhar, and V. Krishnamurthy. An approach to heterogeneous process state capture/recovery to achieve minimum performance overhead during normal execution. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 9 pp.–, April 2003.

[20] E. Cannella, L. Di Gregorio, L. Fiorin, M. Lindwer, P. Meloni, O. Neugebauer, and A. Pimentel. Towards an esl design framework for adaptive and fault-tolerant mpsocs: Madness or not? In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2011 9th IEEE Symposium on*, pages 120–129, Oct 2011.

[21] Emanuele Cannella, Onur Derin, Paolo Meloni, Giuseppe Tuveri, and Todor Stefanov. Adaptivity support for mpsocs based on process migration in polyhedral process networks. *VLSI Des.*, 2012:2:2–2:2, January 2012.

[22] J. Castrillon, R. Leupers, and G. Ascheid. Maps: Mapping concurrent dataflow applications to heterogeneous mpsocs. *IEEE Trans.on Industrial Informatics*, PP(99):1, 2011.

[23] Jeronimo Castrillon, Andreas Tretter, Rainer Leupers, and Gerd Ascheid. Communication-aware mapping of kpn applications onto heterogeneous mpsocs. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1266–1271, New York, NY, USA, 2012. ACM.

[24] H.W.D. Chang and W. J B Oldham. Dynamic task allocation models for large distributed computing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 6(12):1301–1315, Dec 1995.

[25] Weijia Che and Karam S. Chatha. Unrolling and retiming of stream applications onto embedded multicore processors. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1272–1277, New York, NY, USA, 2012. ACM.

[26] Junchul Choi, Hyunok Oh, Sungchan Kim, and Soonhoi Ha. Executing synchronous dataflow graphs on a spm-based multicore architecture. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 664–671, New York, NY, USA, 2012. ACM.

[27] Chen-Ling Chou and R. Marculescu. User-aware dynamic task allocation in networks-on-chip. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 1232–1237, 2008.

[28] Chen-Ling Chou and R. Marculescu. Farm: Fault-aware resource management in noc-based multiprocessor platforms. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[29] Chen-Ling Chou and Radu Marculescu. Run-time task allocation considering user behavior in embedded multiprocessor networks-on-chip. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 29(1):78–91, January 2010.

[30] Joseph E. Coffland and Andy D. Pimentel. A software framework for efficient system-level performance evaluation of embedded systems. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, SAC '03, pages 666–671, New York, NY, USA, 2003. ACM.

[31] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., 1997.

[32] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, June 2002.

[33] D. Cuesta, J.L. Ayala, J.I. Hidalgo, D. Atienza, A. Acquaviva, and E. Macii. Adaptive task migration policies for thermal control in mpsocs. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pages 110–115, July 2010.

[34] E.L. de Souza Carvalho, N.L.V. Calazans, and F.G. Moraes. Dynamic task mapping for mpsocs. *Design Test of Computers, IEEE*, 27(5):26–35, Sept 2010.

[35] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197, Apr 2002.

[36] Soumen Dey and Subhodip Majumder. Task allocation in heterogeneous computing environment by genetic algorithm. In SajalK. Das and Swapan Bhattacharya, editors, *Distributed Computing*, volume 2571 of *Lecture Notes in Computer Science*, pages 348–352. Springer Berlin Heidelberg, 2002.

[37] N. Doulamis, A. Doulamis, A. Panagakis, K. Dolkas, T.A. Varvarigou, and E. Varvarigos. A combined fuzzy-neural network model for non-linear prediction of 3-d rendering workload in grid computing. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 34(2):1235–1247, April 2004.

[38] A. Elantably and F. Rousseau. Task migration in multi-tiled mpsoc: Challenges, state-of-the-art and preliminary solutions. Technical report, Marseille, France, June 2012.

[39] C. Erbas. *System-level modelling and design space exploration for multiprocessor embedded system-on-chip architectures.* PhD thesis, University of Amsterdam, The Netherlands, 2006.

[40] C. Erbas, S. Cerav-Erbas, and A.D. Pimentel. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *Evolutionary Computation, IEEE Transactions on*, 10(3):358–374, 2006.

[41] Yang Ge, P. Malani, and Qinru Qiu. Distributed task migration for thermal management in many-core systems. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 579–584, June 2010.

[42] Stefan Valentin Gheorghita, Twan Basten, and Henk Corporaal. Application scenarios in streaming-oriented embedded-system design. *IEEE Design & Test of Computers*, 25(6):581–589, 2008.

[43] Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stelios Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, and Koen De Bosschere. System-scenario-based design of dynamic embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):3:1–3:45, January 2009.

[44] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hubner, R.K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, and S. Kobbe. Invasive manycore architectures. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, pages 193–200, Jan 2012.

[45] Philip K. F. Hölzenspies, Johann L. Hurink, Jan Kuper, and Gerard J. M. Smit. Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (mpsoc). In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 212–217, New York, NY, USA, 2008. ACM.

[46] S. Hong, S.H.K. Narayanan, M. Kandemir, and O. Ozturk. Process variation aware thread mapping for chip multiprocessors. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 821–826, April 2009.

[47] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, Feb 2010.

[48] Jia Huang, A. Raabe, C. Buckl, and A. Knoll. A workflow for runtime adaptive task allocation on heterogeneous mpsocs. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[49] A.M.A. Hussien, A.M. Eltawil, R. Amin, and J. Martin. Energy aware task mapping algorithm for heterogeneous mpsoc based architectures. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pages 449–450, 2011.

[50] Kai Hwang. *Advanced Computer Architecture*. McGraw-Hill Education (India) Pvt Limited, 2003.

[51] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 359–370, Dec 2006.

[52] J. Jahn, M.A.A. Faruque, and J. Henkel. Carat: Context-aware runtime adaptive task migration for multi core architectures. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[53] Haris Javaid and Sri Parameswaran. A design flow for application specific heterogeneous pipelined multiprocessor systems. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 250–253, New York, NY, USA, 2009. ACM.

[54] Zai Jian Jia, Tomás Bautista, Antonio Núñez, Andy D. Pimentel, and Mark Thompson. A system-level infrastructure for multidimensional mp-soc design space co-exploration. *ACM Trans. Embed. Comput. Syst.*, 13(1s):27:1–27:26, December 2013.

[55] Zai Jian Jia, A.D. Pimentel, M. Thompson, T. Bautista, and A. Nunez. Nasa: A generic infrastructure for system-level mp-soc design space exploration. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*, pages 41–50, 2010.

[56] G. Kahn. The semantics of a simple language for parallel programming. In *Information processing*, pages 471–475. North Holland, Amsterdam, Aug 1974.

[57] Joachim Keinert, Martin Streubühr, Thomas Schlichter, Joachim Falk, Jens Gladigau, Christian Haubelt, Jürgen Teich, and Michael Meredith. Systemcodesigner - an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Trans. Design Autom. Electr. Syst.*, 14(1), 2009.

[58] Joonsoo Kim and Michael Orshansky. Towards formal probabilistic power-performance design space exploration. In *Proceedings of the 16th ACM Great Lakes symposium on VLSI*, GLSVLSI '06, pages 229–234, New York, NY, USA, 2006. ACM.

[59] Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. Distrm: Distributed resource management for on-chip many-core systems. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*, CODES+ISSS '11, pages 119–128, New York, NY, USA, 2011. ACM.

[60] A. Kumar, S. Fernando, Yajun Ha, B. Mesman, and H. Corporaal. Multi-processor system-level synthesis for multiple applications on platform fpga. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 92–97, Aug 2007.

[61] Yu-Kwong Kwok, Anthony A. Maciejewski, Howard Jay Siegel, Ishfaq Ahmad, and Arif Ghafoor. A semi-static approach to mapping dynamic iterative tasks onto heterogeneous computing systems. *J. Parallel Distrib. Comput.*, 66(1):77–98, January 2006.

[62] K.Y. Lee, Y.T. Cha, and J.H. Park. Short-term load forecasting using an artificial neural network. *Power Systems, IEEE Transactions on*, 7(1):124–132, Feb 1992.

[63] Lianq-Yu Lin, Cheng-Yeh Wang, Pao-Jui Huang, Chih-Chieh Chou, and Jing-Yang Jou. Communication-driven task binding for multiprocessor with latency insensitive network-on-chip. In *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, volume 1, pages 39–44 Vol. 1, 2005.

[64] Weichen Liu, Mingxuan Yuan, Xiuqiang He, Zonghua Gu, and Xue Liu. Efficient sat-based mapping and scheduling of homogeneous synchronous dataflow graphs for throughput optimization. In *Real-Time Systems Symposium, 2008*, pages 492–504, Nov 2008.

[65] Chin Lu and Sau-Ming Lau. A performance study on load balancing algorithms with task migration. In *TENCON '94. IEEE Region 10's Ninth Annual International Conference. Theme: Frontiers of Computer Technology. Proceedings of 1994*, pages 357–364 vol.1, Aug 1994.

[66] S. Mahadevan, M. Storgaard, J. Madsen, and K. Virk. Arts: a system-level framework for modeling mpsoc components and analysis of their causality. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*, pages 480–483, Sept 2005.

[67] Nakul Manchanda and Karan Anand. Non-Uniform Memory Access (NUMA). *New York*, 1, 2012.

[68] J.L. Manferdelli, N.K. Govindaraju, and C. Crall. Challenges and opportunities in many-core computing. *Proceedings of the IEEE*, 96(5):808–815, May 2008.

[69] Sorin Manolache, Petru Eles, and Zebo Peng. Task mapping and priority assignment for soft real-time applications under deadline miss ratio constraints. *ACM Trans. Embed. Comput. Syst.*, 7(2):19:1–19:35, January 2008.

[70] Gabriel Marchesan Almeida, Gilles Sassatelli, Pascal Benoit, Nicolas Saint-Jean, Sameer Varyani, Lionel Torres, and Michel Robert. An adaptive message passing mpsoc framework. *International Journal of Reconfigurable Computing*, 2009, 2009.

[71] G. Mariani, P. Avasare, G. Vanmeerbeeck, C. Ykman-Couvreur, G. Palermo, C. Silvano, and V. Zaccaria. An industrial design space exploration framework for supporting run-time resource management on multi-core systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 196–201, March 2010.

[72] Giovanni Mariani, Aleksandar Brankovic, Gianluca Palermo, Jovana Jovic, Vittorio Zaccaria, and Cristina Silvano. A correlation-based design space exploration methodology for multi-processor systems-on-chip. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 120–125, New York, NY, USA, 2010. ACM.

[73] G. Martin. Overview of the mpsoc design challenge. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 274–279, 2006.

[74] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core scc processor: The programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[75] Paolo Meloni, Giuseppe Tuveri, Luigi Raffo, Emanuele Cannella, Todor Stefanov, Onur Derin, Leandro Fiorin, and Mariagiovanna Sami. System adaptivity and fault-tolerance in noc-based mpsocs: the madness project approach. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 517–524. IEEE, 2012.

[76] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. Platform 2012, a many-core computing accelerator for embedded socs: Performance evaluation of visual analytics applications. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1137–1142, New York, NY, USA, 2012. ACM.

[77] J-Y Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 986–991, 2003.

[78] S. Mohanty and V.K. Prasanna. Rapid system-level performance evaluation and optimization for application mapping onto soc architectures. In *ASIC/SOC Conference, 2002. 15th Annual IEEE International*, pages 160–167, Sept 2002.

[79] Orlando Moreira, Jacob Jan-David Mol, and Marco Bekooij. Online resource management in a multiprocessor with a network-on-chip. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 1557–1564, New York, NY, USA, 2007. ACM.

[80] Orlando Moreira, Frederico Valente, and Marco Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '07, pages 57–66, New York, NY, USA, 2007. ACM.

[81] F. Mulas, M. Pittau, M. Buttu, Salvatore Carta, A. Acquaviva, L. Benini, D. Atienza, and G. De Micheli. Thermal balancing policy for streaming computing on multiprocessor architectures. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 734–739, March 2008.

[82] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli. A methodology for mapping multiple use-cases onto networks on chips. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1–6, March 2006.

[83] V. Nollet, P. Avasare, H. Eeckhaut, D. Verkest, and H. Corporaal. Run-time management of a mpsoc containing fpga fabric tiles. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(1):24–33, Jan 2008.

[84] V. Nollet, T. Marescaux, P. Avasare, D. Verkest, and J-Y Mignolet. Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 234–239 Vol. 1, March 2005.

[85] R. O. Oladele and J. S. Sadiku. Article: Genetic algorithm performance with different selection methods in solving multi-objective network design problem. *International Journal of Computer Applications*, 70(12):5–9, May 2013. Published by Foundation of Computer Science, New York, USA.

[86] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the traveling salesman problem. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 224–230, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[87] H. Orsila. *Optimizing Algorithms for Task Graph Mapping on Multiprocessor System on Chip*. PhD thesis, Tampere University of Technology, Finland, 2011.

[88] Heikki Orsila, Tero Kangas, Erno Salminen, Timo D. Hämäläinen, and Marko Hännikäinen. Automated memory-aware application distribution for multi-processor system-on-chips. *J. Syst. Archit.*, 53(11):795–815, November 2007.

[89] Luciano Ost, Sameer Varyani, Leandro Soares Indrusiak, Marcelo Mandelli, Gabriel Marchesan Almeida, Eduardo Wachter, Fernando Moraes, and Gilles Sassatelli. Enabling adaptive techniques in heterogeneous mpsocs based on virtualization. *ACM Trans. Reconfigurable Technol. Syst.*, 5(3):17:1–17:11, October 2012.

[90] Andrew J Page, Thomas M Keane, and Thomas J Naughton. Multi-heuristic dynamic task allocation using genetic algorithms in a heterogeneous distributed system. *Journal of parallel and distributed computing*, 70(7):758–766, July 2010.

[91] G. Palermo, C. Silvano, and V. Zaccaria. Robust optimization of soc architectures: A multi-scenario approach. In *Embedded Systems for Real-Time Multimedia, 2008. ESTIMedia 2008. IEEE/ACM/IFIP Workshop on*, pages 7–12, Oct 2008.

[92] G. Palermo, C. Silvano, and V. Zaccaria. Respir: A response surface-based pareto iterative refinement for application-specific design space exploration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(12):1816–1829, 2009.

[93] JoAnn M. Paul, Donald E. Thomas, and Alex Bobrek. Scenario-oriented design for single-chip heterogeneous multiprocessors. *IEEE Trans. VLSI Syst.*, 14(8):868–880, 2006.

[94] A.D. Pimentel, S. Polstra, F. Terpstra, A.W. van Halderen, J.E. Coffland, and L.O. Hertzberger. Towards efficient design space exploration of heterogeneous embedded media systems. In EdF. Deprettere, Jürgen Teich, and Stamatis Vassiliadis, editors, *Embedded Processor Design Challenges*, volume 2268 of *Lecture Notes in Computer Science*, pages 57–73. Springer Berlin Heidelberg, 2002.

[95] Andy D. Pimentel, Cagkan Erbas, and Simon Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Computers*, 55(2):99–112, 2006.

[96] R. Piscitelli. *Pruning techniques for multi-objective system-level design space exploration.* PhD thesis, University of Amsterdam, The Netherlands, 2014.

[97] R. Piscitelli and A.D. Pimentel. Interleaving methods for hybrid system-level mpsoc design space exploration. In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages 7–14, July 2012.

[98] Roberta Piscitelli and Andy D. Pimentel. Design space pruning through hybrid analysis in system-level design space exploration. In *Proceedings of the Int. Conference on Design, Automation, and Test in Europe (DATE '12)*, pages 781–786, 2012.

[99] Roberta Piscitelli and Andy D. Pimentel. A signature-based power model for mpsoc on fpga. *VLSI Des.*, 2012:6:6–6:6, January 2012.

[100] M. Pittau, A. Alimonda, Salvatore Carta, and A. Acquaviva. Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation. In *Embedded Systems for Real-Time Multimedia, 2007. ESTIMedia 2007. IEEE/ACM/IFIP Workshop on*, pages 59–64, Oct 2007.

[101] Michael Richmond and Michael Hitchens. A new process migration algorithm. *SIGOPS Oper. Syst. Rev.*, 31(1):31–42, January 1997.

[102] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti, and M. Milano. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 6 pp.–, March 2006.

[103] R. Sarikaya, C. Isci, and A. Buyuktosunoglu. Runtime application behavior prediction using a statistical metric model. *Computers, IEEE Transactions on*, 62(3):575–588, March 2013.

[104] N. Satish, K. Ravindran, and K. Keutzer. A decomposition-based constraint optimization approach for statically scheduling task graphs with communication delays to multiprocessors. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1–6, April 2007.

[105] Lars Schor, Iuliana Bacivarov, Devendra Rai, Hoeseok Yang, Shin-Haeng Kang, and Lothar Thiele. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, pages 71–80, New York, NY, USA, 2012. ACM.

[106] A. Schranzhofer, Jian-Jian Chen, and L. Thiele. Dynamic power-aware mapping of applications onto heterogeneous mpsoc platforms. *Industrial Informatics, IEEE Transactions on*, 6(4):692–707, Nov 2010.

[107] Martin Serpell and James E. Smith. Self-adaptation of mutation operator and probability for permutation representations in genetic algorithms. *Evol. Comput.*, 18(3):491–514, September 2010.

[108] A. Shabbir, A. Kumar, B. Mesman, and H. Corporaal. Distributed resource management for concurrent execution of multimedia applications on mpsoc platforms. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 132–139, July 2011.

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

[110] H. Shojaei, A.-H. Ghamarian, T. Basten, M. Geilen, S. Stuijk, and R. Hoes. A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for cmp run-time management. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 917–922, July 2009.

[111] K. Sigdel, M. Thompson, C. Galuzzi, A.D. Pimentel, and K. Bertels. rsesame - a generic system-level runtime simulation framework for reconfigurable architectures. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 460–464, Dec 2009.

[112] Kamana Sigdel, Carlo Galuzzi, Koen Bertels, Mark Thompson, and Andy D. Pimentel. Evaluation of runtime task mapping using the rsesame framework. *Int. J. Reconfig. Comp.*, 2012, 2012.

[113] Kamana Sigdel, Mark Thompson, Andy D. Pimentel, Carlo Galuzzi, and Koen Bertels. System-level runtime mapping exploration of reconfigurable architectures. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[114] A.K. Singh, A. Kumar, T. Srikanthan, and Yajun Ha. Mapping real-life applications on run-time reconfigurable noc-based mpsoc on fpga. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 365–368, Dec 2010.

[115] A.K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: Survey of current and emerging trends. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–10, May 2013.

[116] Amit Kumar Singh, Akash Kumar, and Thambipillai Srikanthan. Accelerating throughput-aware runtime mapping for heterogeneous mpsocs. *ACM Trans. Des. Autom. Electron. Syst.*, 18(1):9:1–9:29, January 2013.

[117] T.T.Y. Suen and J.S.K. Wong. Efficient task migration algorithm for distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 3(4):488–499, Jul 1992.

[118] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. Invasive computing: An overview. In *Multiprocessor System-on-Chip*, pages 241–268. Springer, 2011.

[119] Theocharis Theocharides, Maria K. Michael, Marios Polycarpou, and Ajit Dingankar. Towards embedded runtime system level optimization for mpsocs: On-chip task allocation. In *Proceedings of the 19th ACM Great Lakes Symposium on VLSI*, GLSVLSI '09, pages 121–124, New York, NY, USA, 2009. ACM.

[120] L. Thiele, I. Bacivarov, W. Haid, and Kai Huang. Mapping applications to tiled multiprocessor embedded systems. In *Application of Concurrency to System Design, 2007. ACSD 2007. Seventh International Conference on*, pages 29–40, July 2007.

[121] D. Thierens. Adaptive mutation rate control schemes in genetic algorithms. In *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on*, volume 1, pages 980–985, 2002.

[122] M. Thompson. *Tools and techniques for efficient system-level design space exploration.* PhD thesis, University of Amsterdam, The Netherlands, 2012.

[123] Mark Thompson and Andy D. Pimentel. Exploiting domain knowledge in system-level mpsoc design space exploration. *J. Syst. Archit.*, 59(7):351–360, August 2013.

[124] P. van Stralen. *Applications of scenarios in early embedded system design space exploration.* PhD thesis, University of Amsterdam, The Netherlands, 2014.

[125] P. van Stralen and A. Pimentel. Fitness prediction techniques for scenario-based design space exploration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(8):1240–1253, Aug 2013.

[126] P. van Stralen and A.D. Pimentel. Signature-based microprocessor power modeling for rapid system-level design space exploration. In *Embedded Systems for Real-Time Multimedia, 2007. ESTIMedia 2007. IEEE/ACM/IFIP Workshop on*, pages 33–38, Oct 2007.

[127] P. van Stralen and A.D. Pimentel. A trace-based scenario database for high-level simulation of multimedia mp-socs. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 11–19, July 2010.

[128] P. van Stralen and Andy D. Pimentel. Scenario-based design space exploration of mpsocs. In *Proc. of IEEE ICCD'10*, pages 305–312, October 2010.

[129] S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, Jan 2008.

[130] S. Vassiliadis and I. Sourdis. Flux networks: Interconnects on demand. In *Embedded Computer Systems: Architectures, Modeling and Simulation, 2006. IC-SAMOS 2006. International Conference on*, pages 160–167, July 2006.

[131] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena Moscu Panainte. The molen polymorphic processor. *IEEE Trans. Comput.*, 53(11):1363–1375, November 2004.

[132] Feng Wang, Yibo Chen, C. Nicopoulos, Xiaoxia Wu, Yuan Xie, and N. Vijaykrishnan. Variation-aware task and communication mapping for mpsoc architecture. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(2):295–307, Feb 2011.

[133] Yun Wen, Hua Xu, and Jiadong Yang. A heuristic-based hybrid genetic-variable neighborhood search algorithm for task scheduling in heterogeneous multiprocessor system. *Information Sciences*, 181(3):567 – 581, 2011.

[134] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, September 2007.

[135] Wulf Werum and Hans Windauer. *Introduction to PEARL: Process and Experiment Automation Realtime Language (2Nd Ed.).* Heyden & Sons, Inc., Philadelphia, PA, USA, 1983.

[136] Stefan Wildermann, Felix Reimann, Daniel Ziener, and Jürgen Teich. Symbolic design space exploration for multi-mode reconfigurable systems. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, pages 129–138, New York, NY, USA, 2011. ACM.

[137] W. Wolf, A.A. Jerraya, and G. Martin. Multiprocessor system-on-chip (mpsoc) technology. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10):1701–1713, Oct 2008.

[138] C. Ykman-Couvreur, P. Avasare, G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria. Linking run-time resource management of embedded multi-core platforms with automated design-time exploration. *Computers Digital Techniques, IET*, 5(2):123–135, 2011.

[139] Jia Yu, Jingnan Yao, L. Bhuyan, and Jun Yang. Program mapping onto network processors by recursive bipartitioning and refining. In *Proc. of DAC'07*, pages 805 –810, june 2007.

[140] Nicholas H. Zamora, Xiaoping Hu, and Radu Marculescu. System-level performance/power analysis for platform-based design of multimedia applications. *ACM Trans. Des. Autom. Electron. Syst.*, 12(1):2:1–2:29, February 2007.

[141] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In *Evolutionary Methods for Design, Optimisation, and Control*, pages 95–100. CIMNE, Barcelona, Spain, 2002.

Publications

[1] W. Quan and A. Pimentel, "Towards exploring vast mpsoc mapping design spaces using a bias-elitist evolutionary approach," in *Digital System Design (DSD), 2014 17th Euromicro Conference on,* Aug. 2014, pp. 655–658.

[2] W. Quan and A. D. Pimentel, "Exploring task mappings on heterogeneous mpsocs using a bias-elitist genetic algorithm," *CoRR, vol. abs/1406.7539,* 2014.

[3] W. Quan and A. Pimentel, "A scenario-based run-time task mapping algorithm for mpsocs," in *Proceedings of the 50th Annual Design Automation Conference (DAC '13),* New York, NY, USA: ACM, 2013, pp. 131:1–131:6.

[4] W. Quan and A. D. Pimentel, "An iterative multi-application mapping algorithm for heterogeneous mpsocs," in *Embedded Systems for Real- time Multimedia (ESTIMedia), 2013 IEEE 11th Symposium on,* Oct. 2013, pp. 115–124.

[5] W. Quan and A. D. Pimentel, "A Hybrid Task Mapping Algorithm for Heterogeneous MPSoCs," in *ACM Trans. Embed. Comput. Syst.,* vol. 14, no. 1, Jan. 2015, pp. 14:1–14:25.

[6] W. Quan and A. Pimentel, "A system-level simulation framework for evaluating task migration in mpsocs," in *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '14),* New York, NY, USA: ACM, 2014, pp. 13:1–13:9.

[7] W. Quan and A. D. Pimentel, "Towards Self-adaptive MPSoC Systems with Adaptivity Throttling," in *Proceedings of the 15th Int. Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS '15),* Samos, Greece, July, 2015. (to appear)

[8] W. Quan and A. D. Pimentel, "A Run-time Self-Adaptive Resource Allocation Framework for MPSoC Systems," in *Proceedings of the 22nd European Conference on Circuit Theory and Design (ECCTD '15),* Norway, August, 2015. (to appear)

[9] W. Quan and A. D. Pimentel, "Towards Scalable Scenario-based Run-time Adaptive MPSoC Systems," *Submitted.*

[10] W. Quan and A. D. Pimentel, "Adaptive Large-scale MPSoC Systems," *Submitted.*

Summary

Modern embedded systems, which are more and more based on Multi-Processor System-on-Chip (MPSoC) architectures, increasingly require to be adaptive at run time to support complex and dynamic application workloads, dynamic Quality-of-Service management, etc. As one of the approaches for improving the adaptivity of MPSoC systems, dynamic application task (re-)mapping plays a crucial role in exploiting the system properties such that applications can meet their, often diverse, demands on performance and energy efficiency. The research of this thesis aims at improving these dynamic application mapping techniques to increase the efficiency of modern MPSoC systems by adaptively reconfiguring the system according to the dynamic behaviour of application workloads and the status of the target system.

The application task (re-)mapping methods presented in this thesis belong to the class of hybrid task mapping approaches. They overcome the drawback of static task mapping techniques traditionally considered in embedded systems that are unable to support dynamic application behaviour as well as the drawback of pure dynamic (on-the-fly) task mapping techniques that typically only produce mappings of relatively low quality. Our hybrid task mapping approaches have two mapping optimisation stages: design-time static mapping exploration and run-time mapping optimisation/customisation. At design time, two static Genetic-Algorithm based mapping DSE approaches have been proposed to explore partial task mappings (at the level of inter- or intra-application scenarios) for workload scenarios that might appear on the target MPSoC system under optimisation objectives such as performance and/or energy consumption. At run time, a light-weight resource scheduler – integrated with our proposed mapping optimisation algorithms and a technique for so-called adaptivity throttling – has been deployed for dynamic system reconfiguration. According to the reason that has triggered the system reconfiguration, the scheduler on the target system is able to dynamically derive near optimal application mappings for the purpose of system reconfiguration based on the mapping information explored at design time. However, applying an adaptivity throttling technique, the actual system reconfiguration will only take place when it has been predicted to be beneficial.

By using our proposed hybrid task mapping techniques, which benefit from both static and dynamic task mapping approaches, the efficiency of MPSoC systems can be considerably improved. On top of this, our techniques also provide solutions for the issues of scalability, flexibility and blind adaptivity that general hybrid task mapping approaches suffer from. Moreover, using a hierarchical con-

trol mechanism, we also show that our techniques can perform well on future, large-scale MPSoC systems.

Samenvatting

Moderne embedded systemen zijn steeds vaker gebaseerd op Multi-Processor System-on-Chip (MPSoC) architecturen en moeten zich kunnen aanpassen aan de veranderende eisen van de complexe applicaties tijdens run-time en ondersteuning bieden voor gegarandeerde service niveaus (QoS). Een van de methodes om de adaptiviteit van MPSoC systemen te verbeteren is het dynamisch toewijzen van applicatietaken aan resources zodat aan de snelheidseisen en energiebudgetten voldaan kan worden. Het onderzoek gepresenteerd in dit proefschrift heeft als doel om de efficiëntie van moderne MPSoC systemen te verbeteren door de systeemconfiguratie aan te passen aan het dynamische gedrag van de applicaties en de status van het onderliggende hardware platform.

Dit proefschrift presenteert een methode voor het toewijzen van applicatietaken die behoort tot de klasse van hybride methodes. Deze hybride klasse heeft niet de nadelen van een statische toewijzing van taken, die gebruikelijk is bij embedded systemen, zoals het ontbreken van ondersteuning voor dynamisch applicatiegedrag. Aan de andere kant van het spectrum vinden we de puur dynamische methodes, die configuraties van relatief lage kwaliteit opleveren. Onze hybride methode voor het toewijzen van taken heeft twee fases: Een statische exploratie in de ontwerpfase en een optimalisatie- en aanpassingsfase tijdens run-time. Tijdens de ontwerpfase wordt de ontwerpruimte van partiële taaktoewijzingen (op het niveau van inter- of intra-applicatie scenarios) doorzocht door twee statische methodes die zijn gebaseerd op genetische algoritmes. Hierbij wordt gezocht naar mogelijke applicatiescenarios met werkverdelingen op de MPSoC, geoptimaliseerd op snelheid en/of energiegebruik. Tijdens run-time zorgt een lichtgewicht resource scheduler – die geïntegreerd is met onze algoritmes voor het toewijzen van taken en tevens een methode bevat om de mate van adaptiviteit te begrenzen – voor de dynamische herconfiguratie van het systeem. Aan de hand van de reden die geleid heeft tot een herconfiguratie van het systeem kan de scheduler, door gebruik te maken van de statische exploratie in de designfase, dynamisch een vrijwel optimale toewijzing van taken genereren. Doordat er gebruik gemaakt wordt van de methode die de mate van adaptiviteit begrenst zal een herconfiguratie alleen plaatsvinden als er bepaald is dat deze voordelig is voor het systeem.

Met de door ons voorgestelde hybride methode voor het toewijzen van taken aan resources, die de voordelen van de statische methodes van toewijzing combineert met de dynamische methodes, wordt de efficiëntie van de MPSoC systemen significant verbeterd. Onze methode biedt daarnaast ook oplossingen voor de problemen met schaalbaarheid, flexibiliteit en adaptiviteit waar generieke hybride methodes mee kampen. Bovendien laten we zien dat onze methodes, door gebruik

te maken van een hiërarchisch controlemechanisme, ook goed kunnen presteren op de grootschalige MPSoC systemen van de toekomst.

Acknowledgements

The last near four years of my PhD life leave me an indelible memories. With ups and downs, all the past days are now still as fresh as yesterday. As the proverb says: "No man is an island". In the process of pursuing my PhD degree, I was not alone. Quite a lot of bright people practically or mentally supported and helped me. Without their support and guidance, I would not be able to have this moment. Here I would like to extend my gratitude to all these people.

First, I would like to thank my daily supervisor and co-promotor: Andy Pimentel. I feel very lucky that I have Andy to supervise my PhD research. He provided me an excellent working environment that is full of freedom and trust. Under this environment, I was able to pursue and explore my own direction of research. Apart from providing excellent scientific support and guidance, he was always willing to make time for problem discussion, paper revision and so on. During my PhD life, I have greatly benefited from his professional knowledge and inspirational attitude. I can never thank him enough for his selfless dedication and effort to me.

I would like to express my sincere gratitude to my previous promoter, Prof. Chris Jesshope, for offering me the position and the opportunity to do a PhD in the University of Amsterdam. As Chris has already retired, when I started to write this thesis, Prof. Cees de Laat took over him as my promoter. I am grateful to Cees for his supervision and help at the end of my PhD life. I also want to thank Prof. Chunyuan Zhang from National University of Defence Technology in China. He enlightened me on the research road of computer architecture. His support and encouragement gave me the power to go ahead on my research road.

A very special thanks goes out to Simon Polstra and Peter van Stralen. From the start of my PhD study, I worked together with them in the same room. I really appreciate their countless help on both my research and my daily PhD life. Furthermore, I would like to thank all the other colleagues of in our group, with whom I spent most of my working days. They are Clemens Grelck, Raphael Poss, Roy Bakker, Roeland Douma, Qiang Yang, Jian Fu, Fangyong Tang, Sebastian Altmeyer, Roberta Piscitelli, Irfan Uddin, Michiel van Tol, Merijn Verstraaten. Thanks for all of you! Everything we did together is a nice memory to me.

I would also like to thank my committee members (Prof. Pieter Adriaans, Prof. Henk Corporaal, Prof. Chunyuan Zhang, Dr. Clemens Grelck and Dr. Ana Varbanescu) for serving as my committee members and taking time to review my thesis.

Meanwhile, I want to appreciate the friendship with many great Chinese friends during my stay in the Netherlands. Without them, I would never have had such a wonderful life in the past few years.

Finally, I would like to thank my parents for always supporting me to do what I want and encouraging me when I was upset. I owe my deepest gratitude to my girl, Lingxue, who accompanied me through my PhD life. She brightened my life with her wisdom. Without her, this thesis would not have been possible.

Wei Quan
Amsterdam, juli 2015