Mälardalen University Press Dissertations
No. 192

TASK ALLOCATION OPTIMIZATION FOR MULTICORE EMBEDDED SYSTEMS

Juraj Feljan

2015

School of Innovation, Design and Engineering

Copyright © Juraj Feljan, 2015

ISBN 978-91-7485-241-7

ISSN 1651-4238

Printed by Arkitektkopia, Västerås, Sweden

Mälardalen University Press Dissertations
No. 192

TASK ALLOCATION OPTIMIZATION FOR MULTICORE EMBEDDED SYSTEMS

Juraj Feljan

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid
Akademin för innovation, design och teknik kommer att offentligen försvaras
fredagen den 18 december 2015, 14.15 i Kappa, Mälardalens högskola, Västerås.

Fakultetsopponent: Juniorprofessorin Anne Koziolk, Karlsruhe Institute of Technology

Akademin för innovation, design och teknik

Abstract

Modern embedded systems are becoming increasingly performance intensive, since, on the one hand, they include more complex functionality than before, and on the other hand, the functionality that was typically realized with hardware is often moved to software. Multicore technology, previously successfully used for general-purpose systems, is penetrating into the domain of embedded systems. While it does increase the performance capacity, it also introduces the problem of how to allocate software tasks to the cores of the hardware platform, as different allocations exhibit different extra-functional properties. An intuitive example is allocating too many tasks to a core --- the core will be overloaded and tasks will miss their deadlines.

This thesis addresses the issue of task allocation in multicore embedded systems. The overall goal of the thesis is to advance the way soft real-time multicore systems are developed, by providing new methods and tools that enable deciding already at design-time which task to run on which core, with respect to a number of timing-related extra-functional properties. To achieve this goal, we developed a model-based framework for task allocation optimization. The framework uses model simulation in order to obtain performance predictions for particular task allocations. This in turn enables testing a large number of allocation candidates in search for one that exhibits good timing-related performance. Apart from defining and implementing the framework, three additional contributions are provided, each tackling a particular aspect of the framework: the influence of task allocation on communication duration is studied and interpreted in the context of design-time model-based analysis; a novel heuristic for guiding task allocation optimization is defined; and finally, a novel optimization method combining performance prediction and performance measurement is defined.

Abstract

Modern embedded systems are becoming increasingly performance intensive, since, on the one hand, they include more complex functionality than before, and on the other hand, the functionality that was typically realized with hardware is often moved to software. Multicore technology, previously successfully used for general-purpose systems, is penetrating into the domain of embedded systems. While it does increase the performance capacity, it also introduces the problem of how to allocate software tasks to the cores of the hardware platform, as different allocations exhibit different extra-functional properties. An intuitive example is allocating too many tasks to a core — the core will be overloaded and tasks will miss their deadlines.

This thesis addresses the issue of task allocation in multicore embedded systems. The overall goal of the thesis is to advance the way soft real-time multicore systems are developed, by providing new methods and tools that enable deciding already at design-time which task to run on which core, with respect to a number of timing-related extra-functional properties. To achieve this goal, we developed a model-based framework for task allocation optimization. The framework uses model simulation in order to obtain performance predictions for particular task allocations. This in turn enables testing a large number of allocation candidates in search for one that exhibits good timing-related performance. Apart from defining and implementing the framework, three additional contributions are provided, each tackling a particular aspect of the framework: the influence of task allocation on communication duration is studied and interpreted in the context of design-time model-based analysis; a novel heuristic for guiding task allocation optimization is defined; and finally, a novel optimization method combining performance prediction and performance measurement is defined.

Acknowledgements

Reading my colleagues' acknowledgements (usually at their defenses) I would always find myself imagining how it would feel to be there at the end of the road, writing an acknowledgement of my own. And here I am now, worried that my words will not suffice in describing to you, my dear reader, the amazing period of my life that is now behind me. In your hands (or on your screen) you have the most expected result of a PhD, the thesis, but there is more to it than that. I learned a lot, traveled a lot, experienced a lot... I changed a lot in the process. Thanks to the PhD studies, I get to say I have two home countries, and I cannot imagine anything so intimidating, but at the same time so enriching and rewarding as moving to another country. But, probably the best part of a PhD is how it awards you with getting to know many wonderful people. This is to you, it is my privilege and pleasure to acknowledge your role in this, at times difficult, but mostly remarkable journey.

My deepest thanks goes to my advisors Ivica Crnković and Mario Žagar for giving me the opportunity to become a PhD student. Thank you for believing in me, thank you for all the guidance and support, both professional and personal. Mostly, thank you for your patience. And having mentioned patience — to my co-advisor Jan Carlson, thank you for enduring through the countless excursions I had both to your office and to your e-mail inbox. At some point (or several points) you must have regretted for having me as your PhD student :-). I am amazed by your deep knowledge of many various areas, and the impressive ability to quickly understand and solve detailed technical problems, while at the same time never losing focus from the big picture. I always knew you had my back, even in the face of the most depressing experiment results. I am not exaggerating when I say that without you three this thesis never would have come into existence.

Next, I would like to thank two additional persons who had a special role in my PhD studies. Thank you Tiberiu Secceanu from ABB for inspiring the topic of this thesis and for giving me the opportunity to do an internship in ABB. Thank you Federico Ciccozzi for pushing me forward in the finishing stages of the studies with your great research ideas and help with implementation. It has been a pleasure working with you.

A big thank you to all my office-mates, all my co-authors, all the people I had the pleasure of working with on various courses, all the people that shared the joy and despair of developing an autonomous underwater robot with me, and to all the administrative staff that helped me with visas, apartments, tickets and similar issues. I spent a great deal of time in the first phases of my PhD studies at the Faculty of electrical engineering and computing at the University of Zagreb, so thank you everybody who made this possible and enjoyable.

My friends and colleagues, thank you for the all the fun (and often extremely lively) lunches and coffee breaks, and for making the university feel like home. Whenever I had to work late, at least one of you would be there in the same shoes, which always made me feel much less lonely.

Thank you to my parents Ljerka and Juraj, and my brother Andrija. Without your love and support, I would not be where I am today. Also, thanks for not turning my old room into a storage room.

To my wife Aneta and my daughter Emili — thank you for making my life complete. Aneta, I admire your unselfish love, patience and support. Thank you for always being there for me. Emili, you came into this world just before I started writing the thesis — thank you for always putting a smile on my face and making the thesis writing much less stressful than I feared it would be.

And lastly, thank you dear reader for devoting your time to the thesis, I hope you find what you are looking for.

Juraj Feljan
Stockholm, November 2015

This work was supported by the Unity Through Knowledge Found through the DICES project and the Swedish Foundation for Strategic Research through the Ralf3 project (IIS11-0060).

Contents

1	Introduction	1
1.1	Research goal and questions	2
1.2	Research methodology	5
1.3	Research contributions	6
1.4	Publications	8
1.5	Thesis outline	11
2	Background	13
2.1	Model-based analysis and architecture optimization . . .	13
2.2	Real-time multicore embedded systems	15
3	Impact of allocation on task communication	19
3.1	Memory in a multicore platform	20
3.2	Investigating task communication	21
3.3	Experiment setup	22
3.4	Experiment results	25
3.5	Discussion	28
4	Task allocation framework	31
4.1	Framework overview	32
4.1.1	Key design decisions	32
4.1.2	Structure of the framework	33
4.2	Framework implementation	35
4.2.1	Input models	36
4.2.2	Stop criteria	38
4.2.3	Simulation mechanism	39
4.2.4	Simulation model	44

4.2.5	Optimization mechanism	47
4.3	Validation of the simulation mechanism	52
4.3.1	Experiment setup	52
4.3.2	Experiment results	57
4.3.3	Discussion	64
4.4	Summary	65
5	Delay matrix heuristic	67
5.1	Definition of the heuristic	67
5.2	Evaluation of the heuristic	71
5.2.1	Experiment setup	71
5.2.2	Experiment results	75
5.3	Summary	81
6	Enhancing model-based optimization with monitored system runs	83
6.1	Model-based and execution-based architecture optimization	84
6.2	Model-based and execution-based task allocation optimization	86
6.3	Experiment	90
6.3.1	Experiment setup	90
6.3.2	Experiment results	93
6.4	Summary	97
7	Related work	99
7.1	Model-based performance analysis and architecture optimization	99
7.2	Task allocation in real-time systems	104
8	Conclusion	109
8.1	Contributions	110
8.2	Discussion	114
8.3	Future work	116
	Bibliography	119
A	Impact of allocation on task communication — Experiment results	127

List of Figures

3.1	Task communication in a dual-core system	22
3.2	Stride examples	24
3.3	Experiment results	27
4.1	Task allocation framework	34
4.2	Example software and hardware model	37
4.3	Scenario showing multiple chain instances	41
4.4	Example of simulation visualization: task execution trace	44
4.5	Optimization example: average end-to-end response times of the current candidate and the best candidate	50
4.6	Validation systems	56
4.7	Short-chain system, low load	58
4.8	Short-chain system, high load	59
4.9	Long-chain system, low load	60
4.10	Long-chain system, high load	61
4.11	Mix-chain system, low load	62
4.12	Mix-chain system, high load	63
5.1	Experiment systems	73
5.2	Short-chain system, low load	78
5.3	Long-chain system, low load	78
5.4	Short-chain system, high load	79
5.5	Long-chain system, high load	80
6.1	Combined model-based and execution-based architec- ture optimization	85

6.2	Framework for combined model-based and execution-based task allocation optimization	88
6.3	Software and hardware models of the experiment system	91
6.4	Experiment results	95
6.5	Experiment results over time	96
7.1	Classification of our approach against the architecture optimization taxonomy defined in [1]	103
A.1	128 elements	128
A.2	256 elements	129
A.3	512 elements	130
A.4	4 096 elements	131
A.5	8 192 elements	132
A.6	16 384 elements	133
A.7	262 144 elements	134
A.8	524 288 elements	135
A.9	1 048 576 elements	136
A.10	1 310 720 elements	137

List of Tables

4.1	Optimization example: changes of the best affinity specification	51
5.1	Delay matrix example	70
5.2	Experiment systems and their load	74
5.3	Experiment results, low load	77
5.4	Experiment results, high load	77

List of Listings

4.1	Example of two initial affinity specifications	38
4.2	Example simulation model	45
4.3	Optimization algorithm	47
4.4	Candidate comparison	48
5.1	Delay matrix heuristic	69
5.2	Identifying a problematic task using the delay matrix . .	70

Chapter 1

Introduction

Most computer systems in use today are embedded systems. In fact, more than 98% of all processors produced worldwide work in embedded systems [2]. An embedded system is a microprocessor-based system with a typically single dedicated function (as opposed to general-purpose computer systems), embedded in and interacting with a larger device. Embedded systems range from simple devices (e.g., fitness trackers) to complex systems consisting of multiple nodes communicating over a network (e.g., factory process controllers), and their presence is ubiquitous, as they are used in industry, entertainment, transport, medicine, communication, commerce, etc. An aspect that they share with general-purpose computer systems is a constantly increasing performance demand. They include more complex functionality than before, while having to be reliable, maintainable and robust. At the same time, functionality that had traditionally been realized in hardware is instead being implemented in software (e.g., software defined radio [3]).

There is a trend in embedded systems to cope with the increasing performance demands by increasing the number of processing units, for instance by using multicore technology, which has already been successfully used in general-purpose systems. A multicore processor is a single chip with two or more processing units called cores, that are coupled tightly together in order to keep power consumption reasonable. While enabling a higher performance capacity, increasing the number of processing units also opens up an issue of how to best allocate software modules (in the embedded system domain typically referred to

as tasks) to the available cores, in order to best utilize the hardware platform. The allocation can have a substantial impact on particular performance aspects. An intuitive example is timeliness — if too many tasks are allocated to a core, it will be overloaded, and tasks will miss their deadlines.

Performance is a broad term and there are many characteristics that influence whether an allocation is qualified as good or bad, including for example response time of some critical functionality, energy consumption or memory consumption, but also concerns such as safety, availability, scalability and robustness. This work addresses soft real-time systems — systems where timing is crucial to the correctness of the system but occasional deadline misses can be tolerated — and we focus on timing-related performance aspects¹ such as average timeliness, end-to-end response times and core load. We target modern embedded systems whose hardware architecture resembles the one in today's personal computers — systems with a multicore processor where the cores typically have small amounts of local memory (cache) and share a larger amount of RAM. We believe that this is also the direction the hardware architecture for a majority of embedded systems is heading in the near future.

1.1 Research goal and questions

The overall goal of this thesis is to advance the way multicore embedded systems are developed, through an automatic mechanism for deciding early in the development process which software task to run on which processing core of the hardware platform. More specifically, we aim to develop a model-based framework for design-time optimization of task allocation in soft real-time multicore embedded systems, with respect to a number of timing-related extra-functional properties. Based on the research goal, we formulated four research questions — the first one corresponds to the overall goal, while the remaining ones address specific aspects of the goal. The questions are presented and discussed in the remainder of this section.

¹The terms performance aspects, extra-functional properties and quality attributes are used as equivalents throughout the thesis.

Research question 1: *How can a good task allocation with respect to performance be found automatically at design-time?*

A possible approach for finding out whether a particular allocation gives satisfactory performance could be to implement, deploy and run the system in order to collect performance measurements. However, to avoid redeployment, which can be time-consuming and therefore costly, a preferred approach would be to predict the performance early in the development process, in line with what model-driven engineering [4] and software performance engineering [5] advocate. The idea is to use models of the system under development to obtain performance predictions with sufficient accuracy, already prior to the implementation, and thus get an indication whether a particular allocation is good or bad in terms of performance. Also by using models, we can test the performance of a large number of candidate allocations in less time than we could by performing measurements on a running system. Our first research question deals with the following issues: what kind of models are needed and what information must they contain in order to be able to get design-time performance predictions for different task allocations, and how to use these models and the obtained performance predictions in a mechanism that automatically looks for a good allocation.

Research question 2: *How does allocation of communicating tasks influence the communication duration in the context of design-time model-based performance prediction?*

The duration of communication between tasks is a parameter that has a significant impact on system-level timing properties. In order for performance prediction to be accurate, we need to identify how allocation influences communication duration. In the case of two communicating tasks, the duration of communication can be different depending on whether the tasks are executed on the same core or on separate cores. In the former case, the communication can happen through the fast local memory, while in the latter case it must go through a slower memory shared between the cores. Once we have identified the extent of the difference between intra-core and inter-core task communication, we can discuss the relevance of this difference in the context of design-time model-based performance prediction.

Research question 3: *How to guide local search to find a good allocation of tasks to cores in as few iterations as possible?*

Allocating tasks to cores is a bin packing like problem, and bin packing is NP-hard, i.e., no algorithm is known that can find the optimal solution in polynomial time [6]. Furthermore, an inherent property of design-time model-based analysis — when detailed property values valid for the running system are typically unknown — is that analysis methods use estimates and approximations. Having this in mind, rather than finding the optimal solution, our goal for optimizing task allocation is to find a good allocation quickly (in as few iterations as possible). We have thus opted for local search as the optimization strategy: in each iteration of the optimization, a new candidate is proposed by making a small modification to the best candidate found so far. In its basic form, local search performs random modifications, so it can take many iterations to find a candidate that represents an improvement over the best one. The search can be made more efficient by guiding it to generate better candidates more often — we therefore pair local search with a domain-specific heuristic. This research question deals with defining such a heuristic.

Research question 4: *How can performance measurement and performance prediction be combined in order to improve task allocation optimization?*

Design-time optimization uses performance predictions obtained through model-based analysis for comparing different solution candidates. Model-based analysis is typically fast, but also limited in its accuracy, as it inherently uses estimates and approximations. Other than for analysis, model-driven engineering promotes using models as a specification from which the system implementation can be generated automatically. Having access to code, we can run the system in order to obtain accurate performance measurements, and utilize this when searching for a good task allocation. However, optimization based purely on measurement is typically too slow to be feasible. We would therefore like to leverage both the speed of performance predictions and the accuracy of performance measurements. This research question addresses developing an improved optimization method that combines model-based performance prediction and execution-based performance measurement.

1.2 Research methodology

The research was done within the area of software engineering. It started as a collaboration with an industrial partner, which needed a solution to the problem of quickly finding a good allocation of tasks to the available cores, prior to the implementation. The fact that a practical industrial problem motivated the research classifies it as applied research, but the achieved results are general and applicable in a broader context. The industrial problem was refined and narrowed down into a research setting — this resulted in the overall research goal defined in Section 1.1 and in Research question 1. The remaining research questions originate from this original question, and each one tackles a specific aspect of the overall research goal. For Research questions 1, 3 and 4 we developed a theoretical solution, implemented it and validated it by an experiment. Research question 2 was answered by performing an experiment and interpreting the experiment results.

The framework that was defined as an answer to Research question 1 was first implemented as a proof of concept prototype, to be later extended with support for the following performance metrics: chain end-to-end response time, task and chain deadline misses, and core load. The framework was validated by comparing performance predictions produced by the framework through model simulation to performance measurements obtained from a running system.

In order for our performance prediction to be sufficiently accurate, we needed to identify how allocation influences the duration of communication between tasks, which in turn influences the performance metrics of interest — this defined Research question 2. To answer the question, a series of experiments were defined and executed. The experiments were performed on a system of two tasks that were communicating in different scenarios, while the communication duration was measured. The experiment results were then analyzed and interpreted in the context of design-time model-based analysis.

With the framework in place, we wanted to improve a key part of the search process for a good allocation. This led to Research question 3, which was answered by developing a custom heuristic for proposing a new allocation candidate to be tested in the next iteration of the search process. In order to evaluate the heuristic, we set up an experiment in which the performance of our heuristic was compared against two reference heuristics.

Finally, we wanted to improve the optimization process by complementing performance prediction with performance measurement, which led to Research question 4. In order to answer the question we defined and implemented a novel optimization method that leverages the speed of model-based optimization and the accuracy of execution-based optimization. The feasibility of the approach was demonstrated by an experiment.

1.3 Research contributions

Here we present the scientific contributions of the thesis that address the listed research questions.

Research contribution 1: *A model-based framework for task allocation optimization in soft real-time multicore embedded systems*

We defined an optimization framework for automatically finding a good allocation of software tasks to the processing cores of the hardware platform. The framework uses two models as input — one specifies the software architecture of the system under development, in terms of tasks and the connections between them, while the other specifies the hardware platform. Via an automatic model-to-model transformation, these are translated into an executable model. Since the performance metrics of interest depend on the dynamic interplay between tasks, and since we are interested in average-case performance rather than the worst-case scenario, we cannot obtain performance-related data analytically. Rather, this is done by simulating the aforementioned executable model. Having obtained simulation data, concrete performance metrics can be derived, and used to compare allocation candidates to each other. This in turn enables the optimization mechanism to look for good allocations.

The framework can be implemented for different performance metrics. We provided an implementation supporting end-to-end response times for task chains, task and chain deadline misses, and core load.

Research contribution 2: *The impact of task allocation on communication duration in the context of design-time model-based performance prediction*

We ran a series of experiments to identify the difference between intra-core and inter-core task communication duration. Due to the effect that allocation can have on the duration of task communication,

and thus on system-wide timing properties, this was needed to obtain accurate performance predictions. The intuitive assumption that communication between two tasks on the same core would be faster than communication between two tasks on separate cores held true only in several corner-cases, but identifying such corner-cases is typically not possible at design-time, due to a lack of detailed information of the patterns in which the data shared between tasks is accessed. Thus, in the context of model-based performance analysis, this difference of communication duration can be ignored without sacrificing the accuracy of performance prediction.

Research contribution 3: *A novel heuristic for task allocation optimization with respect to end-to-end response times*

In order to find a good allocation in as few iterations as possible, we developed a novel heuristic for proposing a new allocation candidate to be tested in the next iteration of the optimization. It uses information about how tasks delayed each other during simulation when identifying a problematic task to be moved to a less loaded core. The heuristic both finds better allocations and finds them quicker than the reference heuristics we used for comparison.

Research contribution 4: *A novel task allocation optimization method that combines performance prediction and performance measurement*

We extended the optimization framework with support for monitored system runs. The main idea behind the extended framework was to complement task allocation optimization based on model simulation with optimization based on execution. Running the generated code that implements the system enables us to extract performance metrics by measurement. By also getting access to more accurate metrics than the ones obtained purely by model simulation, the speed of model-based optimization is combined with the accuracy of execution-based optimization: model-based optimization is used to quickly converge towards a good allocation candidate, which is then used as the starting point for the slower, but more accurate execution-based optimization. The combined model-based and execution-based optimization method also represents a general contribution that can be used independently of our framework.

1.4 Publications

In this section we list the main publications the thesis is based on. I was the main author of the text and contributions in papers A, B, C and E, with the coauthors contributing with valuable discussions and comments and smaller amounts of text. For paper D, the coauthor was the driver of the idea, while I contributed with roughly one third of the text. As such, this paper does not take a crucial role in the thesis, rather it represents a possible extension of the optimization framework. It was superseded by paper E, which elaborates further and implements the idea presented in paper D.

Paper A

Towards a model-based approach for allocating tasks to multicore processors, Juraj Feljan, Jan Carlson, Tiberiu Seceleanu, 38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2012

The paper introduces our model-based framework for task allocation optimization with respect to timing-related extra-functional properties. A prototype implementation based on core load is also presented. This corresponds to a part of Research contribution 1.

Abstract: Multicore technology provides a way to improve the performance of embedded systems in response to the demand in many domains for more and more complex functionality. However, increasing the number of processing units also introduces the problem of deciding which task to execute on which core in order to best utilize the platform. In this paper we present a model-based approach for automatic allocation of software tasks to the cores of a soft real-time embedded system, based on design-time performance predictions. We describe a general iterative method for finding an allocation that maximizes key performance aspects while satisfying given allocation constraints, and present an instance of this method, focusing on the particular performance aspects of timeliness and balanced computational load over time and over the cores.

Paper B

The impact of intra-core and inter-core task communication on architectural analysis of multicore embedded systems, Juraj Feljan, Jan Carlson, 8th International Conference on Software Engineering Advances (ICSEA), 2013

The paper presents the experiments performed to identify the difference in duration between intra-core and inter-core communication, and discusses the significance of the difference in the context of design-time model-based performance prediction. This corresponds to Research contribution 2.

Abstract: In order to get accurate performance predictions, design-time architectural analysis of multicore embedded systems has to consider communication overhead. When communicating tasks execute on the same core, the communication typically happens through the local cache. On the other hand, when they run on separate cores, the communication has to go through the shared memory. As the shared memory has a significantly larger latency than the local cache, we expect a significant difference between intra-core and inter-core task communication. In this paper, we present a series of experiments we ran to identify the size of this difference, and discuss its impact on architectural analysis of multicore embedded systems. In particular, we show that the impact of the difference is much lower than anticipated.

Paper C

Task allocation optimization for multicore embedded systems, Juraj Feljan, Jan Carlson, 40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2014

The paper presents an implementation of the framework with respect to end-to-end chain response times, deadline misses and core load, which forms a part of Research contribution 1. The main focus of the paper is our novel heuristic that guides the optimization mechanism, and this corresponds to Research contribution 3.

Abstract: In many domains of embedded systems, the increasing performance demands are tackled by increasing performance capacity through the use of multicore technology. However, adding more processing units also introduces the issue of task allocation — decisions have to be made which software task to run on which core in order to

best utilize the hardware platform. In this paper, we present an optimization mechanism for allocating tasks to cores of a soft real-time embedded system, that aims to minimize end-to-end response times of task chains, while keeping the number of deadline misses below the desired limit. The optimization relies on a novel heuristic that proposes new allocation candidates based on information how tasks delay each other. The heuristic was evaluated in a series of experiments, which showed that it both finds better allocations, and does it in fewer iterations than two heuristics that we used for comparison.

Paper D

Model-driven deployment optimization for multicore embedded real-time systems: the OptimAll approach, Federico Ciccuzzi, Juraj Feljan, 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), 2014

The paper presents an idea of a framework that encompasses our optimization framework and extends it with support for automatic complete code generation as well as back-propagation features for optimization based on monitored system runs. As such it represents a first step towards answering Research question 4.

Abstract: The power of modern embedded systems is continuously increasing together with their complexity, thereby making their development more challenging. In the specific case of the adoption of multicore solutions, while processing power is heavily increased, the issue of allocating software tasks to specific cores on the target platform arises. In this paper we introduce *OptimAll*, an automated model-driven approach that aims at providing support in the delicate phase of task allocation at design time. Besides introducing the entire approach, in this work we focus on the automatic generation of a suitable input to the task allocation optimization mechanism from a UML–MARTE system design model, as well as on the actual optimization mechanism and its outcomes in relation to the design model elements.

Paper E

Enhancing model-based architecture optimization with monitored system runs, Juraj Feljan, Federico Ciccuzzi, Jan Carlson and Ivica Crnković, 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2015

The paper describes an extended version of the task allocation framework which combines model-based optimization with execution-based optimization, leveraging the speed of the former and the accuracy of the latter. This corresponds to Research contribution 4.

Abstract: Typically, architecture optimization searches for good architecture candidates based on analyzing a model of the system. Model-based analysis inherently relies on abstractions and estimates, and as such produces approximations which are used to compare architecture candidates. However, approximations are often not sufficient due to the difficulty of accurately estimating certain extra-functional properties. In this paper, we present an architecture optimization approach where the speed of model-based optimization is combined with the accuracy of monitored system runs. Model-based optimization is used to quickly find a good architecture candidate, while optimization based on monitored system runs further refines this candidate. Using measurements assures a higher accuracy of the metrics used for optimization compared to using performance predictions. We demonstrate the feasibility of the approach by implementing it in our framework for optimizing the allocation of software tasks to the processing cores of a multicore embedded system.

1.5 Thesis outline

In this section we give the outline of the thesis by briefly listing the contents of each chapter.

Chapter 1 — Introduction

The chapter presents the overall research goal and the concrete research questions tackled in the thesis, the research methodology used to guide the research, the research contributions that were achieved, and the publications that the thesis is based on.

Chapter 2 — Background

The chapter gives the preliminaries — it presents model-based analysis and architecture optimization, and real-time multicore embedded systems.

Chapter 3 — Impact of allocation on task communication

The chapter presents the experiment performed in order to identify how task allocation influences the duration of task communication. The identified difference between intra-core and inter-core communication duration is then interpreted in the context of design-time model-based analysis.

Chapter 4 — Task allocation framework

The chapter presents our framework for optimizing task allocation in soft real-time multicore embedded systems. It introduces the structure of the framework and describes the provided implementation, focusing on the model inputs to the framework and on the simulation mechanism used to obtain performance predictions.

Chapter 5 — Delay matrix heuristic

The chapter focuses on the optimization mechanism of the framework, it presents the domain-specific heuristic that the framework uses for guiding the optimization process.

Chapter 6 — Enhancing model-based optimization with monitored system runs

The chapter presents our combined model-based and execution-based architecture optimization method and its application in the task allocation framework.

Chapter 7 — Related work

The chapter presents approaches that are related to the contributions of the thesis.

Chapter 8 — Conclusion

The chapter concludes the thesis by summarizing the contributions, discussing whether they fulfill the overall goal of the thesis, and by presenting possible directions for future work.

Appendix A — Impact of allocation on task communication — Experiment results

The appendix provides the complete results of the experiment presented in Chapter 3.

Chapter 2

Background

In this chapter we give the preliminaries by presenting the two areas that the thesis tackles: model-based analysis and architecture optimization on the one hand, and real-time multicore embedded systems on the other.

2.1 Model-based analysis and architecture optimization

Architecture design is one of the most important activities when developing a non-trivial software system, since decisions made during architecture design have a considerable impact on the extra-functional properties of the system (among other aspects such as the cost of development). Typically, architecture-level decisions include selecting software and hardware components, allocating software components to the available hardware nodes, deciding on system topology, etc. Due to the ever increasing system complexity, software architects face today a search space where manual exploration is not sufficient. This has made automated architecture optimization a prominent research topic over the recent years [1]. Research on architecture optimization has been done for various system domains (e.g., enterprise systems, embedded systems), various system representations (e.g., mathematical models, architecture description languages) and extra-functional properties (e.g., reliability, timing), with varying dimensionality (optimizing for a sin-

gle or for multiple extra-functional properties), degrees of freedom (e.g., component selection, allocation, scheduling) and optimization methods (e.g., local search, genetic algorithms).

In order to compare different architecture candidates to each other, architecture optimization needs concrete values for the extra-functional properties of interest. These could either be performance predictions obtained via model-based analysis or performance measurements obtained by executing the system. The former is typically less accurate than the latter, but often much faster. Due to its speed, model-based analysis enables the optimization to process a larger portion of the search space than what would be possible with system execution. Furthermore, model-based analysis is the only possible choice when performing architecture optimization in an early stage of system development, before the system is implemented.

Despite the diversity of architecture optimization approaches mentioned above, they are typically structured in a similar way: they use model-based analysis to predict extra-functional properties, and pair the analysis with an optimization method (search mechanism). Optimization methods can either be general-purpose (e.g., genetic algorithms) or problem-specific (e.g., domain-specific heuristics). The latter try to leverage domain specific knowledge about the optimization problem to guide the search process to a good solution in less iterations than the former, but at the expense of being applicable to a smaller set of problems.

In the thesis as the optimization method we use local search paired with a domain-specific heuristic. In its basic form, local search iteratively looks for a good solution by doing small random changes to the best candidate found thus far. This means that it can take many iterations to find a candidate that represents an improvement over the currently best one. The search can be made more efficient by employing a custom heuristic to guide it. Such a heuristic uses domain-specific knowledge when proposing a new candidate, with the aim to generate better candidates more often compared to using random changes. A domain-specific heuristic has to have low overhead in order not to slow the search process down. Furthermore, it should be sufficiently intelligent to guide the search process towards a good solution quickly, but without being overly restrictive when generating new candidates (otherwise good solutions might be missed).

2.2 Real-time multicore embedded systems

In order to describe real-time multicore embedded systems, we discuss each aspect of the term separately.

Embedded systems

Computer systems can be divided into two main groups: general-purpose systems and embedded systems. The former are designed to support a variety of different functions, for instance browsing the Web, editing text, manipulating data, tracking inventory and so on. Supporting additional functionality is done by simply installing additional software (applications). General-purpose computer systems include personal computers, servers, smartphones and tablets. On the other hand we have embedded systems — systems designed for a single dedicated function (or a few dedicated functions). An embedded system is embedded in a larger device, often a mechanical or electrical system, and interacts with its environment through sensors and actuators. Embedded systems range from simple devices to complex systems consisting of multiple nodes communicating over a network. They are used in industry, transport, medicine, communication, commerce, entertainment etc. In fact, they make up the majority of today's computer systems — more than 98% of all processors produced worldwide are used in embedded systems [2].

Embedded systems are typically characterized by small size, low power consumption, low per-unit cost and limited processing resources. At the same time they have to be reliable, maintainable and robust. This means that when developing their software, equal focus must be put on the extra-functional properties as on the functionality of the system. Explicit verification must be done to confirm that a system satisfies the requirements set on its extra-functional properties. This can be done after implementation, by measurement, but the preferred approach is to do it earlier in the development process, by performing analysis based on models of the system (see Section 2.1). In general, extra-functional properties cover a variety of aspects such as robustness, reliability, security, safety and performance. The performance category includes properties such as power consumption and memory consumption, but also timing-related extra-functional properties (e.g., response time, deadline misses), as embedded systems often have real-time demands (see the following subsection).

The software of embedded systems is typically built from modules called tasks. A software task is the smallest unit of functionality and it is often implemented with a single thread. An example of a task is a thread that periodically reads the value of a sensor.

Real-time systems

In real-time systems the total correctness of an operation depends not only on its logical correctness, but also on the time when the operation was finished. In other words, a logically correct result which is produced late is equivalent to an incorrect result. Since timing is crucial, it must be explicitly addressed during the development process.

Real-time systems are typically divided into hard real-time and soft-real time systems. For the former, deadline misses are not allowed, and missing a deadline is regarded as a system failure. The absence of deadline misses has to be guaranteed. This is usually done by performing schedulability analysis, which assumes worst-case behavior. For example, a pacemaker or the system controlling the airbags in a car is a hard real-time system.

For soft real-time systems, timing is crucial, but missing deadlines occasionally is tolerated, as it does not represent a system failure, but rather a degradation in system quality. A typical soft-real time example is streaming audio and video — deadlines generally have to be met for the stream to be comprehensible, but intermittent deadline violations that result in stream quality degradation can be recovered from and do not represent a complete system failure. Telecom systems are usually also soft real-time systems. Analysis of soft real-time systems can be done for the worst-case and/or for the average-case behavior.

Real-time systems are often confused with high-performance systems, since meeting deadlines is typically associated with speed. A supercomputer used for doing weather forecast simulations might have impressive performance, but it is not performing real-time calculations. On the other hand, when an engine control system is designed to meet its deadlines, no further gains in performance are necessary. So, real-time systems are not characterized by performance, but by predictability.

Multicore technology

Multicore technology has been used on a large scale for general-purpose computer systems for roughly fifteen years now. In the recent years, it also penetrated the domain of embedded systems. Complex multicore processors are used more and more in embedded systems. For example, processors used in today's microcontroller boards support up to 8 cores at 2 GHz (e.g., the quad-core ARM Cortex-A57 and ARM Cortex-A53 processors [12] in the Dragonboard 810 development board [13]).

A multicore processor is a single chip with two or more processing units called cores, that are coupled tightly together in order to keep power consumption reasonable. The technology developed when the traditional way of increasing processor speed was no longer feasible. Processor frequency had been increased by putting more and more transistors on the same circuit area. The development followed the so called Moore's law — the number of transistors per square centimeter on integrated circuits doubled approximately every two year since the integrated circuit was invented [14]. This enabled single-threaded code to execute faster on newer processors with no modification. However, towards the end of the 1990s, increasing the processor clock rate further led to power consumption and thermal issues. This caused a shift in the way processor speed was increased and gave birth to multicore processors. Multicore technology enabled continuing the overall increase in processor capacity, but it was no longer straightforward for single-threaded programs to benefit from faster execution. Software had to be parallelized in order to fully benefit from the multicore hardware platform. Software parallelization is, however, not within the scope of this work. The thesis tackles how to allocate software modules to the available cores in order to best utilize the multicore platform, and assumes that parallelized software is given.

For general-purpose computer systems, thread allocation is usually not addressed explicitly by the developer of the application software. The developer cannot know what additional applications the system will run, so it is the duty of the operating system to dynamically allocate threads to cores. Embedded systems typically only run a single dedicated application, which makes it possible for the developer to address task allocation already at design-time. Addressing task allocation is necessary since it has a significant influence on extra-functional properties.

In general, executing multi-threaded software on a multicore system can be done according to symmetric multiprocessing (SMP) or bound multiprocessing (BMP) [15]. In SMP all cores are governed by a single operating system and tasks can be dynamically moved between the cores. In BMP all cores are governed by a single operating system, but each core has a local scheduler and task migration is allowed during runtime only according to predefined rules. For example, a task can be allowed to run only on particular cores, but not on all of them. In our approach we use BMP where tasks are not allowed to migrate during runtime at all. The third alternative, called asymmetric multiprocessing (AMP), is not supported by multicore systems. It assumes a separate operating system on each processing unit and no task migration between the units.

Multicore systems can be homogeneous or heterogeneous. The former have identical cores, while the latter have different cores for different specialized purposes (for example for digital signal processing).

Chapter 3

Impact of allocation on task communication

The time tasks spend communicating can have a significant impact on timing-related extra-functional properties, such as response time or throughput. In a multicore system, one of the aspects that affect communication time is the allocation of tasks to the available cores. If two communicating tasks run on the same core, the communication normally happens through the local memory and has thus the potential to be much faster than communication between two tasks running on different cores, which happens through the slower shared memory. In order to be able to give sufficiently precise performance predictions, we need to identify the difference in communication time depending on whether tasks communicate locally with tasks running on the same core, or globally with tasks running on different cores. Due to the significant differences in latencies between the different memories, we intuitively expect this difference to be significant.

In this chapter we discuss the impact that the allocation of tasks to the cores of a multicore system has on communication time. By performing measurements on a running system, we determined the difference between intra-core and inter-core communication duration under varying conditions. We show that in many situations the difference is significantly lower than intuitively expected, and then discuss the reasons and implications of this, namely that the impact of this difference on design-time model-based performance prediction is limited.

The chapter is organized as follows. In Section 3.1 we briefly describe the preliminaries, namely how memory is organized in a multicore platform. Then, in Section 3.2 we discuss in more details about the expected difference between intra-core and inter-core communication in various scenarios. Section 3.3 gives the details of the experiment setup. In Section 3.4 we interpret the experiment results. Finally, in Section 3.5 we discuss the meaning of the experiment results in the context of design-time model-based architectural analysis and in the context of our simulation mechanism used for obtaining performance predictions (presented in Chapter 4).

3.1 Memory in a multicore platform

As mentioned in Chapter 1, the scope of our work are modern and future embedded systems whose hardware architecture resembles the one of today's general purpose computers. Typically in such systems each core of has a small on-chip memory (cache), while a larger off-chip main memory (RAM) is shared between the cores. The cache keeps a copy of a subset of data present in the RAM, in order to make this data available to the CPU at a much lower latency than when accessing data from the RAM. For this, the cache utilizes the fact that the same data is often re-accessed frequently (temporal locality of data), and the fact that data being accessed close in time is often stored in adjacent memory locations (spatial locality of data). Other than cache local to a core (called L1 cache), modern processors typically have additional levels of cache. L2 cache is usually shared between pairs of cores, while L3 cache is shared between all cores. The latency of a particular memory grows in the following order: L1 cache, L2 cache, L3 cache, RAM. Even when having a particular CPU in mind, it is difficult to characterize these values with concrete numbers, but in general L2 cache latency is roughly two to three times larger than L1 cache latency, L3 cache latency is roughly ten times larger than L1 cache latency, and finally RAM latency is two orders of magnitude larger than the latency of L1 cache [16, 17]. When data is transferred between the cache and the RAM, it is done in bigger blocks of fixed size called cache lines. A cache line is usually several tens of bytes long.

When two tasks running on the same core communicate, the communication can happen using only L1 cache, while two tasks running

on different cores have to communicate using one of the shared memories. This means that intra-core communication has the potential to be considerably faster than inter-core communication.

3.2 Investigating task communication

Looking at how memory is organized, we can expect a significant difference in communication duration depending on whether the communicating tasks are allocated to the same core or to different cores. As mentioned before, the goal of this chapter is to identify this difference in practice, through an experiment. Before presenting the experiment, in this section we discuss in more detail about the expected difference in communication duration in different scenarios.

Since many factors other than allocation (for instance, interruptions from other tasks) influence communication time, we start by identifying the case that has the highest potential of exhibiting a significant difference between intra-core and inter-core communication duration. Imagine the following scenario (Figure 3.1): a dual-core system, where each core has L1 cache, and the cores share the RAM. There are two communicating tasks: task $t1$ produces (writes) data which task $t2$ consumes (reads), and task $t2$ runs immediately after task $t1$ completes. The data fits in the L1 cache. If both tasks run on core 0 (scenario depicted in Figure 3.1a), task $t2$ can obtain the data directly from the L1 cache on core 0, where it was written when task $t1$ produced it. On the other hand, if task $t1$ runs on core 0 and task $t2$ on core 1 (scenario depicted in Figure 3.1b), data produced by task $t1$ is stored in the L1 cache of core 0 and not in the L1 cache of core 1. So $t2$ will have to fetch the data from the RAM. Accessing the RAM is around a hundred times slower than accessing L1 cache, so inter-core communication should be significantly slower than intra-core communication. Should the system also have shared L2 cache, the reasoning still applies — since the latency of L2 cache is around two to three times larger than the latency of L1 cache, the difference in communication times should be smaller than in the case when there is no shared cache, but significant nevertheless.

If two communicating tasks do not run immediately after each other, or if they get preempted by a higher priority task, the data they share might be evicted from the cache, due to other data taking its place. The longer the duration between producing and consuming a particular

(a) Intra-core communication

(b) Inter-core communication

Figure 3.1: Task communication in a dual-core system

piece of data, the more likely other data will occupy the cache. In such cases even intra-core communication will have to go through the shared memory, thus reducing the communication time gain from allocating communicating tasks to the same core. Similarly, if the data being communicated does not fit in the local cache, the communication will have to go through the shared memory and the difference between intra-core and inter-core communication is reduced.

3.3 Experiment setup

Here we describe how we set up the experiment performed in order to identify the difference between inter-core communication duration and intra-core communication duration. We specify the hardware and software environments, the general task model and the concrete task setup used in the experiment, and finally the variation points of the experiment.

We used a system with an Intel Core 2 Duo E6700 processor [18]. Each core of this dual-core processor has 32 kB of local L1 cache, while 4 MB of L2 cache is shared between the cores. The cache lines in all caches are 64 bytes long. The system ran a 32-bit version of the Ubuntu 12.04 LTS operating system (kernel version 3.2.29) patched with the PREEMPT RT patch (version 3.2.29-rt44) [19], which turns the stock Linux kernel into a hard real-time kernel. By reducing the overall jitter and enabling the tasks to run at the highest priority, in combination with a high resolution timer of nanosecond granularity, this contributes to reducing unwanted interference in the experiments and increasing the precision of the measurements.

Next we describe the task model used in the experiments. Tasks were implemented as Posix threads [20], and had read-execute-write semantics, meaning that during one instance of execution (i.e., one job) they first read input data, then perform calculations and finally write output data. A task can either be periodic or event-triggered. A periodic task is activated at regular time intervals, while an event-triggered task is activated when the task it receives data from finishes a job. We assume that the tasks exchange data through shared memory, and that each core has access to the whole main memory. Other models (e.g., distributed memory, where each processor has its own local main memory), are possible but since they are not common in embedded systems, they were out of the scope of the experiment.

As identified in Section 3.2, the biggest difference between intra-core and inter-core communication duration should happen in the case where two communicating tasks share data which fits into the L1 cache, and the reader task runs immediately after the writer task finishes a job. We therefore used two tasks in the experiment, a periodic task that wrote data, and an event-triggered task that read the data. The event-triggered task was activated by the periodic task immediately after the data has been written. Data shared between the tasks was an array of integers (integer size in the system was 4 bytes), and each task held a pointer to the shared data. We used bound multiprocessing with no task migration, i.e., each task was allocated to a particular core and could not move to a different core during the execution of a particular experiment. In order to reduce jitter, we ran the tasks at the highest possible priority and prevented memory from being paged to the disk.

In the experiment we measured the time it took the reader task to read the shared data. Between the different experiment runs we varied

the allocation of the tasks to the cores, the pattern of accessing the data, and the size of the data the tasks shared. Regarding the allocation, in the case of intra-core communication both tasks ran on core 0, while in the case of inter-core communication the periodic task ran on core 0 and the event-triggered task ran on core 1.

In order to represent different data access patterns, we varied the stride of accessing the shared data. In other words, the tasks accessed the data array with different increments. See Figure 3.2 for an example of different strides (the grey elements are accessed, while the white ones are skipped). In the experiment runs we used the following strides: 1, 2, 3, 4, 8, 12, 16, 24 and 32. This means that in the different experiment runs, the tasks accessed every element of the shared data array, every second element of the shared data array, every third element of the shared data array, and so on. The point of using different strides was to compare reading times in the following cases: (i) when the data is read sequentially (stride 1), (ii) when the data is read nonsequentially with an increment smaller than the cache line (stride 2, 3, 4, 8 and 12), and finally (iii) when the data is read nonsequentially with an increment larger than the cache line (stride 16, 24 and 32).

In a particular experiment run, the writer and the reader tasks accessed the same amount of data and with the same stride. The amount of data shared between the tasks in different runs was the following number of integers: 128, 256, 512, 4 096, 8 192, 16 384, 262 144, 524 288, 1 048 576, 1 310 720. In order to access N integers with stride S , we allocated a block of data whose size is $N * S * 4$ bytes. This means that the data we allocated in the different runs varied from 512 B (128 integers with stride 1) to 160 MB (1 310 720 integers with stride 32), and thus covered data that fit into the L1 cache, data that was too large for the L1 cache but fit into the L2 cache, and finally data that was too large for the L2 cache but fit into the RAM.

Figure 3.2: Stride examples

3.4 Experiment results

We varied 2 allocations, 9 strides and 10 data sizes, which means that 180 experiment runs were performed in total. In each run we collected 10 000 measurements of the time it took the event-triggered task to read the data written by the periodic task. The complete experiment results are available in Appendix A. Here we illustrate the results by focusing on three representative data sizes: one that fits into L1 cache (256 elements: from 1 kB for stride 1 to 32 kB for stride 32), one that fits into L2 cache (8 192 elements: from 32 kB for stride 1 to 1 MB for stride 32), and finally one that fits into RAM (1 048 576 elements: from 4 MB for stride 1 to 128 MB for stride 32). In Figure 3.3 we show the results as three graphs, one for each data size. As the data size increases, so does the reading time, which is the reason for the difference in the time scales between the graphs. Each graph has two entries for every stride: one for intra-core communication (depicted in black) and one for inter-core communication (depicted in red). Each entry is a boxplot describing the 10 000 measurements. The ends of the boxes show the first and third quartiles, the band inside the box is the second quartile (median), while the whiskers extend to the most extreme data point which is no more than 1.5 times the interquartile range away from the box. For the sake of readability of the graphs, the outliers are omitted.

Comparing the three graphs, we can identify a trend of a relative decrease in the difference between intra-core and inter-core communication when increasing the amount of data shared between the tasks. If we take stride 16 as an example, intra-core communication is 144% faster than inter-core communication when the tasks share 256 integers. When the tasks share 8 192 integers, this difference decreases to 6%, and finally when 1 048 576 elements are shared the difference is 1%. As identified in Section 3.2, this is expected behavior. If the shared data is bigger than the L1 cache, only the end portion of the data will be present in the L1 cache after the writer task has finished writing the data. Since the reader task reads the data from the beginning, it had to be fetched from one of the shared memories (the L2 cache or the RAM, depending on the size of the shared data), regardless of whether the tasks run on the same core or on different cores.

Looking only at the case where the shared data fits into L1 cache (Figure 3.3a), we see the expected significant difference between intra-core and inter-core communication. The inter-core communication takes

(a) 256 elements

(b) 8192 elements

(c) 1 048 576 elements

Figure 3.3: Experiment results

roughly three times as long, which corresponds to the difference in latency between L1 and L2 cache. However, the difference is present only at the higher strides. If the shared data is accessed sequentially (stride 1) there is no significant difference between intra-core and inter-core communication. The reason lies in the way data is transferred between cache and RAM — as mentioned in Section 3.1, this is done at cache line granularity. One cache line of 64 bytes corresponds to 16 integers. So even in the case when the data is not present in the L1 cache, as soon as the reader task reads the first integer from one of the shared memories, one whole cache line is transferred to the L1 cache, containing the currently read element and the 15 subsequent elements. Thus, the next 15 elements will be read from the L1 cache. This continues in the same fashion: after reading one element not present in the L1 cache, the next 15 are read from the L1 cache. In other words, in the case of inter-core communication where we intuitively expected 16 cache misses, we got one cache miss followed by 15 cache hits. Increasing the

stride increases the share of the elements that create cache misses and decrease the share creating cache hits. This explains the increase of the times it takes to read the shared data in Figure 3.3a as we increase the stride. When the stride reaches 16, and thus the difference between two read elements reaches the length of the cache line, then reading every element creates a cache miss. The same happens with the strides higher than 16. Therefore the reading times stay roughly the same even with further increasing the stride. On the other hand, in the case of intra-core communication, the data being read is always present in the L1 cache, regardless of the stride, and the reading times are roughly the same.

It should be noted that the experiment did not cover a memory intensive scenario, with, for example, tasks with high memory usage executing at the same time as other tasks are communicating. In a memory intensive application, due to high activity on the memory bus, tasks might often have to wait before being allowed to access the RAM. This could result in longer inter-core communication times, and thus increase the difference between intra-core and inter-core communication duration. To identify the difference in practice, further experiments would have to be performed in a memory-intensive setting.

3.5 Discussion

The experiment confirmed that when tasks share data that is bigger than the local cache, we do not see a significant difference between intra-core and inter-core communication duration. On the other hand, when the shared data does fit into the local cache, the experiment only partially confirmed the intuitively expected difference in communication times. Inter-core communication took roughly three times as long as intra-core communication (which conforms with the difference between the latencies of the L1 and L2 caches) but only when the shared data was not read sequentially. In the case of sequential data access (stride 1), the difference between intra-core and inter-core communication was marginal, due to the way data is transferred between the RAM and the caches.

When the tasks do not share a set of data elements, but rather a very small amount of data (for instance only one integer), then inter-core communication would be significantly slower than intra-core communication. However, this would likely not have a large impact on the re-

sponse time, since the time it takes to access one data element is typically negligible in comparison with the time that a task spends performing calculations.

In summary, we have seen that in multicore systems with shared memory the difference between intra-core and inter-core communication in most cases is smaller than what could be anticipated from the difference in the latencies of the local and the shared memory. This was shown for the case when the tasks that share data run immediately after each other, which is the most favorable case for exhibiting a significant difference between intra-core and inter-core communication. A typical application would consist of a set of tasks, meaning that tasks that share data would not always run in immediate sequence, and that the difference between intra-core and inter-core communication would be further reduced.

In the context of design-time model-based performance prediction for multicore embedded systems, this has the following consequences. In order to identify whether a particular case exhibits a significant difference between intra-core and inter-core communication, we need detailed information about data access patterns. This information is typically not available prior to the implementation, when we envision the performance predictions to be performed. However, as seen from the experiments, in the typical case, the difference between intra-core and inter-core communication is not significant enough to hinder performing design-time performance prediction. Design-time performance prediction relies on a set of abstractions and estimates, and for a sufficiently precise performance prediction a small difference in a particular input to the analysis (in this case the difference between intra-core and inter-core communication time) can normally be ignored.

The consequence this has for our simulation mechanism used to obtain performance predictions is the following. We do not need to keep track during simulation if tasks communicate within a core or between cores. In other words, we do not need to introduce a special communication cost in the case of intra-core communication. More on this in Chapter 4.

We would like to stress that we do not claim that task allocation has no impact whatsoever on communication duration, and that it can be ignored in the general case. Our context of interest was design-time performance prediction when the code is typically not available. On the other hand, if we have the final implementation code and the

detailed data access patterns are known, and the goal is to minimize communication duration, task allocation is a viable degree of freedom that can result in minimized communication times.

Chapter 4

Task allocation framework

Multicore technology has emerged over the last years as an answer to the increasing performance demands of embedded systems. While increasing the number of processing units does indeed increase the performance capacity, it also introduces the problem of how to best allocate the software tasks to the processing cores of the hardware platform, as the allocation can have a substantial impact on system performance. A possible way to evaluate a particular allocation would be to implement, deploy and run the system in order to get performance measurements. However, rather than employing such a “fix-it-later” approach, we would like to identify a good allocation already prior to implementing and deploying the system, in line with model-driven engineering (MDE) [4] and software performance engineering (SPE) [5], which promote lifting the abstraction level from code to models. Using models makes it possible to obtain performance predictions early in the development process. This in turn enables architecture optimization, as many candidate system configurations can be assessed quickly.

In this chapter we present our model-based framework for performance prediction and architecture optimization of task allocations to the cores of a soft real-time multicore embedded system. The chapter is organized as follows. In Section 4.1 we give a high-level overview of the framework. In Section 4.2 we dive into the details of the framework and present the provided implementation. In Section 4.3 we present how we validated the simulation mechanism that the framework uses to obtain performance predictions. Finally, in Section 4.4 we summarize the chapter.

4.1 Framework overview

In this section we first discuss the key design decisions behind the framework and then present the activities and artefacts that make up the framework, each in their respective subsection.

4.1.1 Key design decisions

The framework is based on three key aspects: (i) model simulation, (ii) local search paired with a domain-specific heuristic and (iii) flexible start conditions and stop criteria for the optimization mechanism. We briefly describe them here, each in their separate paragraph. They will be discussed in more detail throughout this chapter and in Chapter 5.

Since the targeted domain is soft real-time systems and we are interested in average-case performance, and since the performance metrics of interest depend on the dynamic interplay between tasks, performance predictions cannot be obtained by analytically solving a model of the system. Rather, we get the predictions by *model simulation* — we run an executable model of the system in our simulation mechanism. This model is automatically generated from the models describing the software architecture and hardware platform of the system.

As mentioned in Section 1.1, task allocation is an NP-hard problem, meaning that no algorithm is known that can find the optimal solution in polynomial time. Furthermore, as our model-based performance prediction is done prior to implementing the system, when concrete performance values valid for the running system are unknown, the predictions are based on estimates and abstractions. And finally, our optimization function cannot be expressed in the form $y = f(x)$. Having all this in mind, rather than finding the optimal solution, the goal of the framework is to find a good solution quickly. We have therefore opted for a simple optimization method — *local search paired with a domain-specific heuristic*.

The notion of a “good enough” allocation, the available optimization time and the difficulty of finding an acceptable solution are different in different contexts and systems. We do therefore not offer a generic method to stop the optimization process. Rather, we let the user of the framework (typically the software architect) choose the *stop criteria* (for more details, see Section 4.2.2). The framework also provides *flexible start conditions*. In order to mitigate the inherent limitation of local

search — converging to a local optimum, and thus missing the global one — the search process is restarted for multiple starting allocation candidates. These are a combination of initial allocation candidates specified by the software architect and random allocation candidates generated automatically by the framework.

4.1.2 Structure of the framework

The framework represents an iterative method based on local search, where each iteration makes a small modification to the best allocation found so far, and determines by means of simulation if the modification resulted in an improvement or not. Figure 4.1 shows how the framework is structured. The main inputs to the framework are a software and a hardware model, which come as a result of architectural design — a complex manual activity of defining the structure of the system being developed. Together with these models, the software architect can specify any number of initial allocation candidates (initial affinity specifications) to be tested by the framework. An affinity specification defines the affinity of each task — the parameter which tells which core the task is allocated to.

The software model specifies the software architecture of the system, in terms of tasks and the connections between them. The hardware model defines the execution platform, specifying the number of cores and scheduling options on each core. Via an automatic model-to-model transformation, from the software model and hardware model, a simulation model is generated. This is an executable model that captures the dynamic interaction between the tasks, including task scheduling and transfer of control. Having executed the simulation model for a particular affinity specification, we can process the data collected during simulation, and derive from it concrete performance metrics, by which we can then compare affinity specification candidates against each other. Based on the best affinity specification found thus far, the framework generates a new candidate, to be fed back to the simulation activity. This forms the optimization cycle (marked with the gray rounded rectangle in Figure 4.1) that repeats simulation, performance metrics derivation, candidate comparison and candidate generation in search for a good allocation, until the stop criterion has been met.

The optimization cycle is restarted for each initial affinity specification provided by the software architect. In order to avoid local optima,

Figure 4.1: Task allocation framework

a large number of restarts should be performed. For this purpose, the framework can complement the initial affinity specifications provided by the software architect with a desired number of random starting affinity specifications. Reading the initial affinity specifications and later generating random starting affinity specifications is done in the Starting candidate creation activity shown in Figure 4.1.

When the optimization mechanism stops, it compares the best affinity specifications identified in each restart, and outputs the best one among them as the final affinity specification.

4.2 Framework implementation

The description of the framework presented in Section 4.1 was agnostic to particular extra-functional properties and implementation details. This is because the framework can be implemented for various modeling technologies, extra-functional properties, simulation mechanisms and optimization heuristics. In this section we discuss our provided implementation of the framework. It uses UML [21], MARTE [22] and Xtend [23] as the modeling technologies; end-to-end response times for task chains, task and chain timeliness (deadline misses) and core load as the extra-functional properties of interest; a custom simulation engine implemented in Java; and finally a custom domain-specific optimization heuristic. An additional, proof-of-concept implementation of the framework exists [7]. It is based on Mathworks Matlab and Simulink [24] as the modeling technology and simulation engine; peak core load and task timeliness as the extra-functional properties of interest; and a simple random heuristic. As the proof-of-concept implementation shares many aspects with the more complete implementation, it will not be discussed further. Rather, this section will focus on detailing the complete implementation.

The goal of the implemented framework is to minimize the average end-to-end response time for a particular task chain, while keeping the overall number of chain deadline misses in the system below a desired limit. The software and hardware specifications of the system are defined using UML and MARTE. From these specifications, by means of an automatic model-to-text transformation implemented using Xtend, we generate a simulation model. This is a Java class that describes the structure of the system and is used as input to our simulation mecha-

nism. Having run the simulation, from the simulation data we obtain the metrics of interest: end-to-end response time for the chain being optimized, deadline misses for all chains, deadline misses for all tasks and core load for all cores. The former two extra-functional properties are used to compare different allocation candidates against each other, while the latter is used by the optimization heuristic when proposing a new allocation candidate to be tested in the next step of the optimization. In the following subsections, we discuss these aspects in more detail. A running example is used to connect the subsections.

4.2.1 Input models

This subsection discusses the model inputs to the framework, namely the software model, hardware model and initial affinity specifications. They are defined by the software architect during the architecture design phase. The former two are specified using UML and MARTE. MARTE is a UML profile used for modeling and analyzing real-time embedded systems. We use only a subset of MARTE, as this is sufficient for our modeling needs. An alternative would have been to define our own modeling language, but instead we chose to use a widely-known standard language, even if we only use a small part of it.

The software model specifies the software architecture of the system in terms of tasks and the connections between them. Tasks are modeled as instances of a class stereotyped with MARTE's `<<swSchedulableResource>>` stereotype. Since they can be periodic or event-triggered, we have defined two classifiers: `PeriodicTask` and `EventTriggeredTask`. The semantics of the two types of tasks are as follows: the former is activated periodically, while the latter gets activated when the task preceding it finishes one instance of execution. Tasks have the following parameters: best-case execution time, worst-case execution time, priority and, for periodic tasks, period.

The connections between the tasks model control flow only. Data flow does not need to be modeled explicitly, as it is not needed in order to accurately simulate the aforementioned extra-functional properties of interest (more on this in Section 4.3). A sequence of control flow represents a task chain. In other words, a task chain is started by a periodic task which can be followed by any number of event-triggered tasks triggered in sequence. This means that each chain is activated with the same period as the periodic task at the start of the chain. We

Figure 4.2: Example software and hardware model

support only connections of single dimensionality, meaning that a task can at most trigger one task and be triggered by one task. Currently, we do not support triggering by outside events.

An example software model is shown in the upper part of Figure 4.2. It contains chains of varying length — from single-task chains to 10-task chains.

The hardware model is quite straightforward. Since the approach assumes only homogeneous cores, the following information is sufficient to fully describe a hardware platform: the number of available cores and the type of scheduler each core runs. Cores are modeled as instances of a class stereotyped with MARTE's <<hwProcessor>> stereotype and typed with our Core classifier. Similarly, schedulers are modeled as instances of a class stereotyped with MARTE's <<scheduler>> stereotype. Currently, the supported scheduler options are a preemptive priority-based (PPScheduler classifier) and a nonpreemptive priority-based scheduler (NPPScheduler classifier). With the former type, when multiple tasks are ready for execution on a core, the one with highest priority executes until it is finished or until a task with higher priority becomes ready. With the latter type, when multiple tasks are ready for execution on a core, the one with highest priority executes until it is finished. The lower part of Figure 4.2 shows an example hardware model, with two cores running preemptive priority-based schedulers.

An affinity specification is a mapping between the software and hardware models. It defines on which core each task is executed. As mentioned in Section 4.1.2, the software architect can specify a number of initial affinity specifications for the framework to use as starting points when restarting the optimization cycle. These are specified in a simple textual format, as illustrated in Listing 4.1.

Listing 4.1: Example of two initial affinity specifications

```
t1#0 t2#0 t3#0 t4#1 t5#1 t6#1 t7#0 t8#0 t9#0 t10#1 t11#1 t12#1 t13#0 t14#0 t15#0 t16#1 t17#1 t18#1 t19#0
t20#0 t21#0 t22#1 t23#1 t24#1 t25#0 t26#0 t27#0 t28#1 t29#1 t30#1

t1#1 t2#0 t3#1 t4#1 t5#0 t6#1 t7#1 t8#0 t9#1 t10#1 t11#0 t12#1 t13#1 t14#0 t15#1 t16#1 t17#0 t18#1 t19#0
t20#0 t21#1 t22#1 t23#0 t24#1 t25#0 t26#0 t27#1 t28#1 t29#1 t30#1
```

4.2.2 Stop criteria

In addition to the model inputs described above, the software architect has to specify the stop criteria for the optimization mechanism. Looking

at the structure of the framework (described in Section 4.1.2 and depicted in Figure 4.1), we can see that there are two levels of stop criteria needed:

- i one for the optimization cycle performed for a particular starting affinity specification (the inner loop), and
- ii one for restarting the optimization cycle for multiple starting affinity specifications (the outer loop).

Regarding the inner loop, the software architect can choose between the following stop criteria: a fixed number of optimization iterations, a certain number of iterations that have not found an improvement (a better allocation candidate), a given time limit, or a combination of the latter two (for example, run the optimization cycle for at most 10 minutes or until there is no improvement in 10 consecutive iterations).

Regarding the outer loop, the supported stop criteria are the following: a fixed number of restarts or a given time limit. Restarting the optimization cycle for all the initial affinity specifications defined by the software architect can mean that the optimization mechanism has to be run for more restarts or for longer time than specified by the stop criterion. The software architect can specify if he/she wants to test every initial affinity specification regardless of the stop criterion. In the case that the stop criterion is not reached after having restarted the optimization cycle for every initial affinity specification, the framework generates additional necessary starting affinity specifications randomly.

An interesting additional stop criterion would be reaching a desired value for a particular performance metric. This is however not implemented in the framework.

4.2.3 Simulation mechanism

In this subsection we detail the simulation mechanism used to obtain predicted values for the extra-functional properties of interest: task deadline misses, chain deadline misses, end-to-end response times and core load. As said earlier, since these properties depend heavily on the dynamic interaction between the tasks and since we are interested in average-case behavior, it is not possible to derive the property values analytically from task and platform parameters. Rather, we do it by simulation.

The simulation mechanism is implemented in Java and based on an object-oriented design with separate classes for the important aspects

of the software and hardware models — tasks, chains and schedulers (which also represent cores). Simulation is done on a tick basis — in every tick, each scheduler checks for ready tasks on its core, determines which task gets to run and then executes it. The duration of the simulation (the number of ticks) is chosen by the user of the framework. In order to get representative average performance metrics, the duration should at least cover several hyper-periods (the hyper-period for a set of tasks is the least common multiple of the tasks' periods).

A task can be in one of the following three states: passive, ready or executing. It is in the passive state while it waits for an activation. Upon activation, the task will perform one instance of task execution (i.e. a job). Periodic tasks are activated periodically, while an event-triggered task gets activated when its preceding task finishes a job. Upon being activated, the task transitions to the ready state. In other words, a new job is released for this task, at which point the execution time for the job is generated randomly with a uniform distribution¹ between the corresponding task's best- and worst-case execution times. When the task is granted access to the processor, it transitions to the executing state and reduces its remaining execution by one unit per simulation tick. If another task preempts it, it goes back to the ready state, where it waits for processor access again. When the remaining execution reaches zero, the task returns to the passive state. In other words, this particular job is done. This structure allows for both preemptive and nonpreemptive schedulers to be defined, since the scheduler decides which task to grant execution in each simulation step. As mentioned in Section 4.2.1, the implementation supports a preemptive and a nonpreemptive priority-based scheduler, but it is simple to add other types of schedulers.

In the next few paragraphs we discuss how the simulation mechanism handles the aforementioned extra-functional properties of interest. We start with task and chain deadline misses, followed by end-to-end response times, and end with core load.

When a task gets activated while its current job is still active, a deadline miss occurs. For periodic tasks this implicitly means that the deadline is equal to the period. An explicit specification of deadlines for each task would be straightforward to add to the framework, but this is currently not implemented. In the event of a deadline miss, the current job is allowed to finish, while the next job (the one causing the deadline

¹Other distributions are possible, but not currently implemented.

miss) is dropped. This means that each task can only have one instance (one job) active at a time.

Such a set up, on the other hand, allows multiple active instances of a particular chain. Let's look at the following scenario. We have three tasks that build a chain:

- periodic task $t1$, BCET = 2, WCET = 5, priority = 2, period = 10,
- event-triggered task $t2$, BCET = 2, WCET = 4, priority = 2, and
- event-triggered task $t3$, BCET = 1, WCET = 2, priority = 1.

Task $t1$ is run on core 0, while tasks $t2$ and $t3$ on core 1. Both cores run a preemptive priority-based scheduler. A possible task execution trace is shown in Figure 4.3. Between time steps 0 and 6 there is only one instance of the chain active, likewise between time steps 10 and 20. However, at step 20, the current instance of the chain has not yet finished and periodic task $t1$ becomes ready and starts executing. At this point we have two instances of the chain active, even though all the tasks have only one active job.

At the time when the last task in a chain finishes execution, it is checked if the chain met its deadline. Chain deadlines can be set to a desired value, but by default the deadline of a chain corresponds to the period of the task starting the chain. In the current example this means that the chain missed its deadline. This also illustrates the fact that chain deadline misses can occur without the need for any task to miss its deadline. Furthermore, when a task misses its deadline, it is also considered as a deadline miss of the corresponding chain. This

Figure 4.3: Scenario showing multiple chain instances

is because a job is dropped, which means that one chain instance will never finish executing.

The end-to-end response time for a chain is defined as the time elapsed between the point when the periodic task at the start of the chain gets activated and the point when the last task in the chain finishes the corresponding job. However, since chains can have multiple active instances, calculating end-to-end response times is not as trivial as subtracting the start time from the finishing time. In Figure 4.3 we can see a scenario where such a subtraction would result with an incorrect end-to-end response time. Time step 20 when the periodic task gets activated is the start time of the chain. Task t_3 finishes its job at time step 21, which would then give an end-to-end response time of one time unit. This is incorrect due to the fact that the end time of 21 does not correspond to the start time of 20, but rather to the start time of 0.

In our simulation mechanism, end-to-end response times are calculated in the following way. In order to allow multiple chain instances, the start times of chains are stored in an array of size two. Again using the scenario from Figure 4.3, at time step 0 the start time of the chain is stored (`start_times[0] = 0`). Then at time step 20, the start time of the next chain instance is stored (`start_times[1] = 20`). When task t_3 finishes its job, it subtracts the start time at index 0 from its finishing time, resulting with one instance of its end-to-end response time ($21 - 0 = 21$). The instance is stored and contributes to the average end-to-end response time for the chain. Then the start time from index 1 is moved to index 0.

When a task misses its deadline, due to the fact that the job is dropped, this particular instance of the chain will never finish. In that case the corresponding chain start time (`start_times[1]`) is removed from the array and this instance of the chain does not contribute to its average end-to-end response time. As said earlier, a task deadline miss is also counted as a deadline miss of the chain the task belongs to.

This mechanism does not allow for more than two active instances of a chain at the same time. If such a scenario occurs during simulation, the allocation candidate is marked as invalid, and will as such not be used by the optimization mechanism. This is a design decision taken in order to keep the calculation of response times simple.

Having covered task deadline misses, chain deadline misses and end-to-end response times, we now focus on the last extra-functional property of interest, namely core load. It is used by our custom heuristic

during optimization for proposing the next allocation candidate. More on this in Chapter 5. Core load is the only extra-functional property of interest that is calculated statically, before running the simulation, as that is sufficiently precise for its intended use. For a particular core, its load depends on the tasks allocated to the core, their rate of triggering and their average execution times. Core load is equal to the sum of average loads of all tasks allocated to the said core. Average load for a task is equal to its average execution time $((\text{BCET} + \text{WCET}) / 2)$ divided by the period of its chain (i.e. the period of the periodic task starting the chain).

An aspect of the simulation mechanism that we have not discussed yet is how it tackles task communication. Time tasks spend communicating can have a significant impact on timing-related extra-functional properties. In general, the simulation mechanism assumes that time tasks spend communicating is captured by the tasks' worst-case execution time parameter, meaning that communication time is part of execution time. This, however, does not take into account the potential difference in the duration of communication when two tasks communicate within the same core versus when they communicate between cores. In the former case, the communication can normally happen thorough fast memory that is local to a core, while in the latter the communication has to go through a slower memory shared between the cores. However, in the context of design-time model-based performance predictions, we have identified that the difference between intra-core and inter-core communication duration can be ignored without hurting the accuracy of predicting the extra-functional properties of interest. This was discussed in detail in Chapter 3. Having investigated the difference between intra-core and inter-core communication duration, we can validate the simulation mechanism. This will be presented in Section 4.3.

Other than being used by the optimization mechanism when assessing a particular allocation candidate, the simulation mechanism can be used directly by the software architect if there is a need to review a particular allocation in more detail. In addition to providing the numeric values for the properties of interest, the simulation mechanism can also output a visualization of different system aspects. Visualization of the following is supported: task execution trace, task state changes, dynamic load, task deadline misses, chain deadline misses, average end-to-end chain response times, and delay matrix (see Chapter 5 for a

definition of the delay matrix). Figure 4.4 shows one of the supported graphs, namely the task execution trace for one core. It was obtained by executing the simulation model generated from the software and hardware models shown in Figure 4.2, for the first initial affinity specification defined in Listing 4.1. The graph shows for each step of the simulation (x axis) which task was executed in that particular step (the one that reduces its execution time), and which tasks were ready for execution, along with their remaining execution time (y axis).

4.2.4 Simulation model

The simulation model is used as input to the simulation mechanism. It is an executable specification of the hardware and software of the system, in the form of a Java class file. It is generated automatically from a software and hardware model. For a particular software and hardware model, the simulation model is always the same, as affinity specifications are kept external to the simulation model. In other words, the simulation model takes an affinity specification as a parameter.

Listing 4.2 shows the simulation model generated from the example software and hardware models, and used to obtain the simulation visu-

Figure 4.4: Example of simulation visualization: task execution trace

alization shown in the previous subsection. The structure is straightforward. First the tasks are declared. Then their affinities are set according to the affinity specification. After this, the task connections are specified, using Java's events and listeners framework. Next, the chains and schedulers are declared. Finally, we have code for statically calculating core load (as mentioned in the beginning of Section 4.2.3, cores in the simulation mechanism are represented with the Scheduler class).

Listing 4.2: Example simulation model

```
package se.mdh.mcore.systems;

import se.mdh.mcore.model.AffinitySpecification;
import se.mdh.mcore.model.Chain;
import se.mdh.mcore.model.EventTriggeredTask;
import se.mdh.mcore.model.PeriodicTask;
import se.mdh.mcore.model.PreemptivePriorityScheduler;
import se.mdh.mcore.model.Scheduler;
import se.mdh.mcore.model.Task;
import se.mdh.mcore.simulation.Simulation;

public class ExampleSystem extends System {

    public void create(Simulation simulation, AffinitySpecification affSpec)
    {

        Task t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13, t14, t15,
            t16, t17, t18, t19, t20, t21, t22, t23, t24, t25, t26, t27, t28,
            t29, t30;

        // tasks
        t1 = new PeriodicTask("t1", 5, 2, 3, 100);
        t2 = new EventTriggeredTask("t2", 5, 2, 3);
        ...
        t30 = new PeriodicTask("t30", 1, 2, 3, 100);

        // read task affinities
        t1.setAffinity(affSpec.getAffinity(t1.getId()));
        t2.setAffinity(affSpec.getAffinity(t2.getId()));
        ...
        t30.setAffinity(affSpec.getAffinity(t30.getId()));

        simulation.addTask(t1);
        simulation.addTask(t2);
        ...
        simulation.addTask(t30);

        // task connections
        t1.setTriggeredTask((EventTriggeredTask)t2);
        t2.setTriggeredTask((EventTriggeredTask)t3);
        t3.setTriggeredTask((EventTriggeredTask)t4);
        t4.setTriggeredTask((EventTriggeredTask)t5);
        t5.setTriggeredTask((EventTriggeredTask)t6);
        t6.setTriggeredTask((EventTriggeredTask)t7);
```

```
t7.setTriggeredTask((EventTriggeredTask)t8);
t8.setTriggeredTask((EventTriggeredTask)t9);
t9.setTriggeredTask((EventTriggeredTask)t10);

t11.setTriggeredTask((EventTriggeredTask)t12);
t12.setTriggeredTask((EventTriggeredTask)t13);
t13.setTriggeredTask((EventTriggeredTask)t14);
t14.setTriggeredTask((EventTriggeredTask)t15);

t16.setTriggeredTask((EventTriggeredTask)t17);
t17.setTriggeredTask((EventTriggeredTask)t18);

t19.setTriggeredTask((EventTriggeredTask)t20);
t20.setTriggeredTask((EventTriggeredTask)t21);

t22.setTriggeredTask((EventTriggeredTask)t23);

t24.setTriggeredTask((EventTriggeredTask)t25);

// task chains
Chain c1 = new Chain(0, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10);
Chain c2 = new Chain(0, t11, t12, t13, t14, t15);
...
Chain c11 = new Chain(0, t30);

simulation.addChain(c1);
simulation.addChain(c2);
...
simulation.addChain(c11);

// schedulers, i.e. cores
Scheduler s0 = new PreemptivePriorityScheduler("0");
Scheduler s1 = new PreemptivePriorityScheduler("1");
simulation.addScheduler(s0);
simulation.addScheduler(s1);

// calculate average load of the tasks
for (Task t : simulation.getTasks()) {
    if (t instanceof PeriodicTask) {
        ((PeriodicTask) t).calculateLoadForTasksDownTheChain();
    }
}

// calculate core load
int load;
for (Scheduler s : simulation.getSchedulers()) {
    load = 0;
    for (Task t : s.getTasks()) {
        load += t.getAverageLoad();
    }
    s.setStaticLoad(load);
}

}

}
```

The automatic generation of a simulation model from a software and a hardware model is defined using the Xtend programming language [23]. This model-to-text transformation is relatively straightforward — it traverses the software and hardware models, and on parsing a particular model aspect it generates its counterpart in Java, according to the structure described above.

4.2.5 Optimization mechanism

Having described the model inputs to the framework and the simulation mechanism used to obtain the concrete performance metrics, in this subsection we focus on the optimization mechanism.

In Listing 4.3 we show the pseudocode of our algorithm for task allocation optimization with respect to end-to-end response time. The algorithm takes the following as input: the simulation model, the number of simulation steps (ticks), the initial affinity specifications provided by the software architect, and the two-level stop criteria described in Section 4.2.2. As depicted in Figure 4.1 and mentioned in Section 4.1.2 and Section 4.2.2, the algorithm is structured into two loops: the inner loop that runs the optimization cycle for a particular starting affinity

Listing 4.3: Optimization algorithm

```
optimize(simulation_model, number_of_simulation_steps,
        initial_affinity_specs, stop_criteria) {
    while (not(reached(stop_criteria.outer_criterion))) {
        affinity_spec = nextStartingAffinitySpecification(
            initial_affinity_specs);
        best_candidate = null;
        while (not(reached(stop_criteria.inner_criterion))) {
            simulation_results = simulate(simulation_model,
                number_of_simulation_steps, affinity_spec);
            candidate = <simulation_results, affinity_spec>;
            if (betterOrEqual(candidate, best_candidate)) {
                best_candidate = candidate;
            }
            affinity_spec = generateNewAffinitySpecification(best_candidate);
        }
        final_candidates.add(best_candidate);
    }
    overall_best = getBestCandidate(final_candidates);
    return overall_best;
}
```

specification, and the outer loop that restarts the optimization cycle for multiple starting affinity specifications.

Having executed the simulation model in the inner loop, in the general case simulation data is parsed in order to obtain concrete performance metrics, in the performance metric derivation activity shown in Figure 4.1. In the concrete case of task and chain deadlines, chain end-to-end response times and core load, the performance metrics derivation activity is trivial, as these metrics are directly produced by the simulation mechanism in their numeric form. Having obtained the metrics, the current allocation candidate is compared with the best candidate found so far. If the current candidate is at least as good as the best one found so far, it becomes the new best candidate.

Comparison of allocation candidates is done according to the algorithm defined in Listing 4.4. Since the goal of the optimization is to minimize the average end-to-end response time for a particular task chain, while keeping the number of chain deadline misses in the system

Listing 4.4: Candidate comparison

```
betterOrEqual(a, b) {
    if (not(a.isFeasible()) && not(b.isFeasible())) {
        if (a.getChainDeadlineMisses() < b.getChainDeadlineMisses()) {
            return true;
        } else if (a.getChainDeadlineMisses() > b.getChainDeadlineMisses()) {
            return false;
        } else {
            if (a.getChainToOptimize().getAverageResponseTime() <= b.
                getChainToOptimize().getAverageResponseTime()) {
                return true;
            } else {
                return false;
            }
        }
    } else if (a.isFeasible() && not(b.isFeasible())) {
        return true;
    } else if (not(a.isFeasible()) && b.isFeasible()) {
        return false;
    } else {
        if (a.getChainToOptimize().getAverageResponseTime() <= b.
            getChainToOptimize().getAverageResponseTime()) {
            return true;
        } else {
            return false;
        }
    }
}
```


below a desired limit, these two metrics are used for the comparison. We call an allocation candidate where the number of chain deadline misses is kept below the desired limit a feasible allocation candidate. A feasible candidate is always better than an infeasible one. Of two infeasible candidates, the one with fewer chain deadline misses is better. Of two feasible candidates, the better allocation candidate is the one with the lower average end-to-end response time for the chain being optimized.

Having done candidate comparison, we generate a new affinity specification, based on the best candidate found thus far and guided by a domain-specific heuristic. The new candidate is assessed in the next iteration of the optimization. Since the new candidate is obtained by making a small modification to the best one found so far, the inner loop performs local search around one starting affinity specification.

In order to avoid getting stuck in a local optimum, the optimization cycle is repeated for multiple starting allocations. This forms the outer loop of the algorithm. The outer loop starts by resetting the best candidate and defining a new starting affinity specification. The latter is either the next initial affinity specification specified by the software architect, or a new random affinity specification in the case that all initial affinity specifications have been used. In the end of each iteration of the outer loop, the best candidate found by the inner loop is saved. When the stop criterion for the outer loop is reached (as said in Section 4.2.2, this can either be a particular number of restarts or a given time limit), the algorithm compares the best candidates found in each restart, and outputs the overall best.

Since the candidate generation activity represents a key part of the optimization mechanism, it will be discussed in a separate chapter, namely Chapter 5. There we present the aforementioned domain-specific heuristic, which uses information about how tasks delayed each other during simulation when proposing a new allocation candidate to be tested in the next iteration of the optimization.

Next, we illustrate a run of the optimization mechanism using the example we introduced earlier (the software and hardware model from Figure 4.2 and the corresponding simulation model from Listing 4.2). We ran it for a single initial affinity specification, namely the first one from Listing 4.1 and for 20 iterations of the optimization cycle¹. The chain whose end-to-end response time was optimized was the one starting

¹Deliberately low numbers for illustration purposes. A real case would be run for more restarts and iterations per restart.

with task $t1$. The results are shown in Figure 4.5 and Table 4.1. The former shows the end-to-end response time of the current candidate in each iteration and the end-to-end response time of the best candidate found until a particular iteration. The latter illustrates the best affinity specification for each iteration in which an improvement occurred. A white cell means that the corresponding task is allocated to core 0, while a black cell marks the task as allocated to core 1.

Figure 4.5: Optimization example: average end-to-end response times of the current candidate and the best candidate

Table 4.1: Optimization example: changes of the best affinity specification

step:	1	2	6	8	10	12
t1						
t2						
t3						
t4						
t5						
t6						
t7						
t8						
t9						
t10						
t11						
t12						
t13						
t14						
t15						
t16						
t17						
t18						
t19						
t20						
t21						
t22						
t23						
t24						
t25						
t26						
t27						
t28						
t29						
t30						

4.3 Validation of the simulation mechanism

Having presented the framework, in this section we discuss the validation of the simulation mechanism. The validation of the optimization mechanism is given in Chapter 5. The main goal of this section is to demonstrate that the performance predictions obtained by the simulation mechanism are sufficiently precise when compared to a real running system. Even though the simulation mechanism does not represent a major scientific contribution of the thesis by itself, it is a necessary condition for three out of four contributions of the thesis, and as such has to give accurate performance predictions in order not to hinder the validity of the three contributions.

As aforementioned, there are four extra-functional properties of interest that the implemented framework supports: end-to-end response times for task chains, chain deadline misses, task deadline misses and core load. The validation focuses on the former two, so in this section we present the experiment performed to show that the simulation mechanism gives precise predictions for chain end-to-end response times and chain deadline misses.

The section has the following layout: in Section 4.3.1 we describe the organization of the experiment. In Section 4.3.2 we present the experiment results, and finally we discuss the meaning of the results in Section 4.3.3.

4.3.1 Experiment setup

Here we describe the set up of the performed experiment. For chain end-to-end response times and deadline misses, we compared predicted values obtained by the simulation mechanism to measured values obtained by executing different task architectures. In order to be able to generalize the results, we used three different representative systems: one with many short chains, one with a few long chains and one with a mix of short and long chains. We will refer to them as the short-chain system, the long-chain system and the mix-chain system, respectively. For each system we varied the load by scaling the tasks' best-case and worst-case execution times, giving in total six tested systems. The software models of the systems are shown in Figure 4.6 (the BCET and WCET parameters given there are valid for the low load systems, while for the high load systems the BCET and WCET parameters are multiplied by 2). All tasks

within one chain had the same priority. The priority of the tasks of the observed chain was roughly in the middle of the priority span for the respective system. In all systems the deadline of the observed chain was equal to the period of the periodic task starting the chain. The following were the observed chains for the different systems:

- short-chain system: chain $t4 - t6$,
- long-chain system: chain $t11 - t17$,
- mix-chain system: chain $t11 - t15$.

In the simulation mechanism, task communication within a core and between cores was treated the same, i.e. there was no additional cost used for inter-core communication. This decision was made based on the findings presented in Chapter 3.

For the implemented systems, the hardware and software environments and the task model were the same as the ones used in the experiment described in Section 3.3. As described there, tasks were implemented as Posix threads [20], and had read-execute-write semantics, meaning that during one job a task first reads input data, then performs calculations and finally writes output data. Task calculations were represented by incrementing an integer variable in a loop. For each job, the number of loop iterations was randomly generated within the loop limits that correspond to the BCET and WCET parameters of the equivalent task in the software model. For the communicating tasks, task communication consisted of reading and/or writing 10 integers.

For each system, 1000 random allocations were tested. Each allocation was simulated for 30 000 simulation steps and executed for 30 seconds. The number of simulation steps and the duration of the executions were chosen to: (i) run the systems long enough to capture the average behavior, and (ii) to result in a similar number of activations of the observed chain during simulation and during execution. The observed chain was activated roughly 300 times.

(a) Short-chain system

(b) Long-chain system

(c) Mix-chain system

Figure 4.6: Validation systems

4.3.2 Experiment results

In this section we present the results of the validation experiment. Please note that in the case of the short-chain system with high load and mix-chain system with high load, 996 and 997 allocations respectively contribute to the results, instead of 1000, due to the occurrence of more than two simultaneous instances of the observed chain during either simulation or execution. Since, as mentioned in Section 4.2.3, we do not support this kind of system, the results of these allocations were not used.

The results are summarized in Figure 4.7 – Figure 4.12, while the complete results, together with the code, are available at [25]. As the metric for comparing performance predictions obtained by simulation and performance measurements obtained by execution, we use the absolute value of the difference between the predicted and the measured value. The tables summarize the results by giving the average, standard deviation and median of the difference. Furthermore, in order to be able to judge the size of the difference, for each system we list the minimum and maximum response time of the executed allocations, and the minimum and maximum number of deadline misses of the executed allocations. As the observed chain is triggered 300 times, 300 is also the maximum theoretical number of deadline misses.

The high standard deviation indicates that in the general case the difference between the performance predictions and performance measurements can be quite scattered around the average. This is why the tables are complemented by boxplots (in the case of the systems with low load, there were no deadline misses in the simulations or executions, so their boxplots are omitted). The boxplots depict the distribution of the difference. The ends of the boxes show the first and third quartiles, the band inside the box is the second quartile (median), while the whiskers extend to the most extreme data point which is no more than 1.5 times the interquartile range away from the box. The dots show the outliers.

Finally, we illustrate the difference between the simulation and execution with line graphs for response times and deadline misses (in the case of the systems with low load, there were no deadline misses in the simulations or executions, so their graphs are omitted). The purpose of the line graphs is to illustrate how well the performance predictions (the blue line) follow the performance measurements (the red line) across different allocations. The graphs depict the first 50 allocations tested for

each system — the number of shown allocations is limited in order to keep the graphs readable. Note that there is no bias in choosing the first 50 allocations, as all allocations are random.

	Difference			Execution	
	avg	stdev	median	min	max
RT	1.651	0.977	1.472	14.645	58.709
DM	0	0	0	0	0

Figure 4.7: Short-chain system, low load

Figure 4.8: Short-chain system, high load

	Difference			Execution	
	avg	stdev	median	min	max
RT	0.467	0.332	0.405	27.110	37.431
DM	0	0	0	0	0

Figure 4.9: Long-chain system, low load

	Difference			Execution	
	avg	stdev	median	min	max
RT	1.638	1.361	1.294	56.642	85.708
DM	24.428	18.340	22.000	0	119

Figure 4.10: Long-chain system, high load

	Difference			Execution	
	avg	stdev	median	min	max
RT	1.761	0.704	1.673	24.062	54.245
DM	0	0	0	0	0

Figure 4.11: Mix-chain system, low load

Figure 4.12: Mix-chain system, high load

4.3.3 Discussion

In this section we discuss the results of the experiment. Since the criteria for whether performance predictions are good enough depend on the context, we keep the discussion here at a high level.

Focusing on end-to-end chain response time, the simulation mechanism generally reflects the executions well. From the tables it can be seen that the difference between the performance predictions obtained by simulation and performance measurements obtained by execution is relatively small, and the line graphs show that across different allocations the performance predictions follow the same trend as the performance measurements. The outliers in the boxplots show that particular cases give a bigger difference between simulation and execution.

When it comes to deadline misses, all simulations and executions of the systems with low load resulted in no deadline misses. The more interesting case here is the systems under high load. The tables and boxplots witness that in some cases the difference between simulation and execution is considerable, but the line graphs show that the simulation mechanism reflects the measurements well. The biggest difference between simulation and execution is exhibited for the long-chain system with high load.

As said in the introduction to Section 4.3, the simulation mechanism itself is not a major scientific contribution of the thesis, but is on the other hand an important aspect, as several contributions rely on it to provide satisfactory performance predictions. The experiment showed that the simulation mechanism is sufficiently accurate to be usable in general, and as such provides a good basis for the scientific contributions of the thesis. In particular cases the difference between the performance predictions and performance measurements can be considerable, but on average the simulation mechanism gives results that are close to the metrics obtained by execution, and the line graphs show that simulation follows execution well across different allocations.

4.4 Summary

In this chapter we introduced our model-based framework for task allocation optimization in multicore embedded systems. We discussed the key design decisions behind the framework (model simulation, local search paired with domain-specific heuristics, flexible start conditions and stop criteria for the optimization mechanism) and its structure. Then we presented the existing implementation of the framework, used for optimizing task allocations with respect to end-to-end response times of task chains. We focused on the model inputs to the framework and on the simulation mechanism used for predicting the extra-functional properties of interest (end-to-end response times for task chains, task deadline misses, chain deadline misses, core load). We also briefly presented the optimization mechanism in the implemented framework. We will get back to it in more detail in Chapter 5, where we discuss our domain-specific heuristic used to propose new allocation candidates. Finally, we presented how we validated the simulation mechanism, for which we had set the basis earlier in Chapter 3.

Chapter 5

Delay matrix heuristic

In this chapter we dive deeper into the optimization mechanism of our task allocation framework. At the center of the optimization mechanism is a novel heuristic that guides the iterative search for a good allocation. As explained in Section 4.1.1, rather than finding the optimal solution, the goal of the framework is to find a good solution quickly. We have therefore opted for a simple optimization method — local search paired with a domain-specific heuristic. Our heuristic is based on a so called delay matrix which contains information about how tasks delay each other during simulation. This information is used to propose a new candidate allocation for assessment in the next iteration of the optimization.

The chapter is organized as follows. Section 5.1 describes how the delay matrix heuristic works. Section 5.2 presents the experiment conducted in order to evaluate the heuristic. Finally, Section 5.3 concludes the chapter.

5.1 Definition of the heuristic

In Section 4.2.5 we presented how the optimization mechanism of the framework works. To briefly recap, the mechanism iteratively searches for a good allocation. Through model simulation it obtains performance metrics for a particular affinity specification and then, using these metrics, compares the current affinity specification to the best one found thus far. Based on the best affinity specification, a new candidate is

proposed for testing in the next iteration of the optimization. The creation of a new candidate could be done by a simple random change, for instance by relocating a random task of the best affinity specification to a random core, but in order to speed up the optimization, this change can be guided. This is the role of our delay matrix heuristic. This section describes the heuristic.

The heuristic is based on the following principle: the most problematic task on a core is: (i) a task that considerably delays tasks that belong to the chain we are optimizing, and/or (ii) a task from the chain we are optimizing that is considerably delayed by other tasks. When proposing a new candidate, the heuristic takes the best affinity specification found thus far as basis and moves such a problematic task to a core that has low load.

The pseudocode describing the heuristic is given in Listing 5.1 (notice that it builds on Listing 4.3). In each step (iteration) of the optimization, the heuristic starts by identifying a *chain of interest* for that particular step. If the best allocation candidate found thus far is feasible (i.e. the overall number of chain deadline misses is lower than the specified limit), the chain of interest is the one the software architect chose to be optimized. On the other hand, if the best allocation found so far is infeasible, the heuristic will prioritize minimizing the number of deadline misses over minimizing the response time. Therefore, the chain of interest for an infeasible allocation is randomly selected among the ones that have deadline misses, with the probability of choosing a particular chain being proportional to the number of deadlines missed by the chain.

Having identified the chain of interest, the heuristic chooses a task to be relocated to a different core. This is done based on the *delay matrix*, a structure that holds information about how tasks delay each other during simulation. We define task delaying in the following way: in each step of the simulation, the task that gets executed delays by one unit all the other tasks that were ready for execution on the same core. The more a task delayed the tasks in the chain of interest, and the more a task in the chain of interest was delayed, the higher the probability that it will get picked for relocation to a different core.

After identifying the task to be relocated, the heuristic decides where to place the task. This is done using core load — the less a core is loaded, the bigger the chance the task will be moved there. As described in Section 4.2.3, core load for each core was calculated statically before running the simulation.

Listing 5.1: Delay matrix heuristic

```

generateNewAffintySpecification(best_candidate) {
    if (isFeasible(best_candidate)) {
        chain_of_interest = CHAIN_TO_OPTIMIZE;
    } else {
        chain_of_interest = selectChainBasedOnDeadlineMisses(best_candidate);
    }
    delay_matrix = getDelayMatrix(best_candidate);
    task_to_move = selectProblematicTaskForChain(delay_matrix,
        chain_of_interest);
    old_affinity = getAffinity(task_to_move, best_candidate);
    new_affinty = selectCoreWithLowLoad(best_candidate, old_affinity);
    affinity_spec = getAffintySpecification(best_candidate);
    affinity_spec = moveTask(affinity_spec, task_to_move, new_affinty);
    if (random() <= TASK_SWITCH_PROBABILITY) {
        task_to_switch = selectRandomTaskFromCore(newAffinty);
        affinity_spec = moveTask(affinity_spec, task_to_switch, old_affinity);
    }
    return affinity_spec;
}

```

In addition to relocating a task from one core to another, the heuristic occasionally performs a task switch. This addresses particular situations where all cores are close to fully loaded. In such cases, simply moving a task would most likely result in an infeasible allocation. The probability of task switching is defined by the system designer as a parameter of the optimization. Having moved a problematic task to a different core as described above, a random task is picked from the new core and moved to the original core of the problematic task.

Next, we look in more detail how the delay matrix is used to choose a task to be relocated. The algorithm is given in Listing 5.2 and will be explained using a simple example system with four tasks: tasks $t1$, $t2$ and $t4$ are periodic, while task $t3$ is triggered by task $t2$. The $t2 - t3$ chain is the current chain of interest. Let us assume that the simulation resulted in the delay matrix shown in Table 5.1a. The matrix is read in the following way: task $t1$ delays $t2$ for a total of 50 units, $t1$ delays $t3$ for 20 units and so on. One unit of delay means a task delayed another task during one step of the simulation. The chain of interest is marked with gray in the table.

According to the algorithm shown in Listing 5.2, for each task a delay parameter is calculated — it corresponds to how much the task delays the tasks in the chain of interest. Additionally, if the task itself belongs

Listing 5.2: Identifying a problematic task using the delay matrix

```

selectProblematicTaskForChain(delay_matrix, chain_of_interest) {
    delay_sum = 0;
    foreach task in tasks {
        delay = 0;
        foreach task2 in chain_of_interest {
            delay += delay_matrix[task][task2];
        }
        if (belongsTo(task, chain_of_interest)) {
            foreach task2 in tasks {
                delay += delay_matrix[task2][task];
            }
        }
        delay_info = addPair(delay_info, task, delay);
        delay_sum += delay;
    }
    foreach task in tasks {
        probability = getDelay(delay_info, task) / delay_sum;
        prob_info = addPair(prob_info, task, probability);
    }
    selected_task = selectRandomWithProbabilities(prob_info);
    return selected_task;
}

```

Table 5.1: Delay matrix example

(a) Delay matrix					(b) Resulting probabilities		
	<i>t1</i>	<i>t2</i>	<i>t3</i>	<i>t4</i>		Delay	Prob.
<i>t1</i>	—	50	20	40	<i>t1</i>	70	50%
<i>t2</i>	0	—	0	25	<i>t2</i>	50	35.71%
<i>t3</i>	0	0	—	0	<i>t3</i>	20	14.29%
<i>t4</i>	0	0	0	—	<i>t4</i>	0	0%

to the chain of interest, its delay parameter is increased by the amount it is delayed by all the other tasks. For task $t1$ the delay parameter is $70 - 50$ for delaying task $t2$ plus 20 for delaying task $t3$. For task $t2$ the delay equals to 50 — it does not delay the other task in the chain ($t3$), but it is delayed by task $t1$ by 50 units. Similarly, the delay for task $t3$ is 20 and for task $t4$ it is 0. The delay parameter of each task is normalized, i.e., divided by the sum of the delay parameters of all tasks (in this case 140). The normalized delays represent the probabilities of selecting a particular task as the problematic one — the more it delays and is delayed, the bigger the chance it will be picked for relocation. The resulting probabilities in this particular example are shown in Table 5.1b.

5.2 Evaluation of the heuristic

An experiment was conducted to evaluate the efficiency of the delay matrix heuristic. Here we describe the experiment setup and experiment results, in their respective subsections. We used two reference heuristics to compare the delay matrix heuristic against: random and load. The former proposes a new allocation candidate by taking a random task from a random core of the best allocation found so far, and moving it to a random other core. The latter heuristic tries to balance the load evenly among the cores. In each step of the optimization, it moves a random task from a random core with high load to a random core with low load. The more a core is loaded, the higher the chance that a task belonging to it will be moved, and equivalently, the less a core is loaded, the higher the chance that the task chosen for relocation will be moved there.

5.2.1 Experiment setup

We ran task allocation optimization using the delay matrix, random and load heuristics, respectively, on four representative systems covering scenarios of both low and high load in the system, and scenarios with both short and long task chains. We used two software architectures (shown in Figure 5.1), one with short chains and one with long chains, and varied the load in both cases, giving in total four tested systems (see Table 5.2). The load was changed by multiplying the BCET and WCET parameters of the tasks by a factor of 2. The BCET and WCET parameters shown in Figure 5.1 are valid for the systems with low load.

(a) Short-chain system

(b) Long-chain system

Figure 5.1: Experiment systems

Table 5.2: Experiment systems and their load

	Best-case load	Worst-case load
Short-chain system, low load	0.40	0.47
Long-chain system, low load	0.36	0.42
Short-chain system, high load	0.80	0.94
Long-chain system, high load	0.72	0.84

Each system consisted of 30 tasks and 4 cores, which made a total of 4^{30} possible allocation candidates. All cores used a preemptive priority-based scheduler. Task communication within a core and between cores was treated the same, i.e. there was no additional cost used for inter-core communication (based on the findings presented in Chapter 3).

In the short-chain system, the tasks were organized into 5 chains consisting of 3 tasks, 5 chains of 2 tasks and 5 chains with a single task (Figure 5.1a). The long-chain system had 2 chains of 10 tasks, 1 chain of 5 tasks and 5 single-task chains (Figure 5.1b). All tasks within one chain had the same priority. One chain roughly in the middle of the priority span was chosen for optimization in each system:

- short-chain system: chain $t_{22} - t_{24}$,
- long-chain system: chain $t_{16} - t_{25}$.

The feasibility limit was set to 0, meaning that an allocation was considered feasible only in the case when no chain deadlines were missed. Chain deadlines were equal to the respective periods.

All optimization runs started from the same starting allocation, where the tasks were distributed evenly among the four available cores. 4 systems and 3 different heuristics gave 12 optimization runs. Each optimization run was executed for 150 steps (iterations). In other words, each optimization run tested 150 allocation candidates beginning with the same starting allocation. The 12 optimization runs were repeated 100 times, in order to be able to draw conclusions about the general performance of the heuristics. The probability of task switching was set to 30% for all three heuristics. Both the probability of task switching and the number of steps for each optimization run were chosen arbitrarily. Additional experiments would be necessary to be able to reason about possible rules of thumb for these parameters.

5.2.2 Experiment results

The results of the experiments are summarized in Table 5.3 and Table 5.4 and in Figure 5.2 – Figure 5.5.

The tables show the number of feasible final allocations, the average feasibility point, and the average final response time for the optimized chain, for each of the 4 systems and 3 heuristics. The number of feasible final allocations tells how many of the 100 repetitions of the optimization runs ended with a feasible allocation as the best one. A feasibility point is the step in an optimization run when the first feasible allocation was found. Since each optimization run had 150 steps, the average feasibility point is in the interval between 0 and 150. The average final response time is based only on the optimization runs that ended with a feasible allocation.

Focusing first on the systems with low load, we see that all three heuristics have an average feasibility point of 0 (Table 5.3). This is because of the fact that already the starting allocation for these runs happened to be feasible, due to the low load in the system and the equal distribution of the tasks to the cores. Starting an optimization run from a feasible allocation means that the run will always identify a feasible allocation as the best one. Therefore, all three heuristics have the maximum possible number of feasible final allocations. All three heuristics managed to minimize the response time of the chosen chain to a similar value on average, with the values found by the delay matrix heuristic being slightly lower.

Figure 5.2 and Figure 5.3 present in more detail the impact of the heuristics on the systems with low load. Each point in the diagrams shows the average value of the 100 optimization runs at the corresponding optimization step. Even though all three heuristics ended up with a roughly similar response time value after 150 optimization iterations, it is clear that the delay matrix heuristic converges much faster than the other two — it got close to the final value already after roughly 30 optimization steps.

Shifting focus to the systems with high load, from Table 5.4 we can see that the delay matrix heuristic both found a slightly lower final response time, and that it found a feasible allocation faster than the two other heuristics. Also, in the case of the short-chain system, it found feasible allocations more often.

The optimization runs for the systems with high load are illustrated with two separate diagrams in Figure 5.4 and Figure 5.5 — one diagram up to the feasibility point and one from the feasibility point onwards. This was necessary due to the fact that up to the feasibility point, the optimization process tries to minimize the number of chain deadline misses, while only from the feasibility point onwards does it try to minimize the chain response times, as explained in Section 5.1. The values shown in the diagrams before the feasibility point give the number of chain deadline misses, while the values in the diagrams after the feasibility point give the response time of the optimized chain. As feasibility points are different for different optimization runs, rather than showing the absolute optimization steps, the x-axes of the diagrams after the feasibility point show the optimization steps relative to the feasibility point, denoted by N . Also, since not all optimization runs ended up with a feasible allocation, the diagram values after the feasibility point represent an average of less than 100 values (while the diagram values before the feasibility point represent an average of exactly 100 values).

Looking at Figure 5.4a and Figure 5.5a, the trend is again clear, and confirms what the average feasibility points show — that the delay matrix heuristic minimized deadline misses faster than the two other heuristics. Similarly, from Figure 5.4b and Figure 5.5b, we can see that our heuristic minimized response times faster, and ended up with an overall lower response time.

It should be noted that the two reference heuristics showed to be quite similar. In other words, the load heuristic did not perform better than the random heuristic. The fact that our heuristic is better than the load heuristic shows that our heuristic performs good not only thanks to relocating tasks to less loaded cores, but predominantly thanks to the delay matrix.

Table 5.3: Experiment results, low load

(a) Short-chain system

	Random	Load	Delay matrix
Number of feasible final allocations	100	100	100
Average feasibility point	0	0	0
Average final RT	10.44	10.52	10.24

(b) Long-chain system

	Random	Load	Delay matrix
Number of feasible final allocations	100	100	100
Average feasibility point	0	0	0
Average final RT	35.41	35.62	35.18

Table 5.4: Experiment results, high load

(a) Short-chain system

	Random	Load	Delay matrix
Number of feasible final allocations	25	25	54
Average feasibility point	109.72	101.76	84.69
Average final RT	75.00	75.24	69.07

(b) Long-chain system

	Random	Load	Delay matrix
Number of feasible final allocations	38	44	38
Average feasibility point	101.11	101.14	85.42
Average final RT	88.81	89.55	87.79

Figure 5.2: Short-chain system, low load

Figure 5.3: Long-chain system, low load

(a) Before the feasibility point

(b) After the feasibility point

Figure 5.4: Short-chain system, high load

(a) Before the feasibility point

(b) After the feasibility point

Figure 5.5: Long-chain system, high load

5.3 Summary

In this chapter we presented a key part of the optimization mechanism of our task allocation framework: a novel heuristic for proposing new allocation candidates. The heuristic makes decisions on which task to relocate to a different core based on a delay matrix — a structure which holds information about how tasks delay each other during simulation.

In an experiment study we showed that the heuristic exhibits promising results, and it fulfills the goal of quickly finding a good allocation. The conducted experiments demonstrated that the heuristic converges towards a good allocation faster than two reference heuristics we used for comparison. Also, our heuristic identified final allocations which were on average slightly better than the final allocations found by the two reference heuristics.

Chapter 6

Enhancing model-based optimization with monitored system runs

In addition to using models for getting performance predictions, model-driven engineering promotes utilizing them as basis for automatic code generation. In the case when automatic generation of the complete system implementation from models is possible, performance prediction can be combined with performance measurement, resulting in improved optimization.

In this chapter we present our extended task allocation framework that exploits a combination of model-based predictions for optimization speed, and measured runtime values — gathered from executing the generated system code — for optimization accuracy. This allows quickly finding a good allocation candidate by prediction-based optimization at modeling level, and then further refining it by continuing the same optimization mechanism, but now based on monitored system runs. Enhancing optimization with performance measurements, rather than only using performance predictions, increases the accuracy of the performance metrics used for optimization, with respect to the actual system properties.

The chapter is organized as follows. As the idea of combined model-based and execution-based optimization is a contribution applicable

also outside of our task allocation framework, in Section 6.1 we describe it as a general architecture optimization method. Then, in Section 6.2 we present the application of the method in our task allocation framework. Section 6.3 demonstrates the feasibility of the approach — we present an experiment performed using the extended framework. Finally, in Section 6.4 we summarize the chapter.

6.1 Model-based and execution-based architecture optimization

In this section we elaborate the motivation for combining performance prediction and performance measurement, and then present our novel model-based and execution-based architecture optimization method. The method as described here is not dependent on our task allocation framework and could be used in any architecture optimization approach where a complete system implementation can be generated automatically from the system models.

As said in Section 2.1, architecture optimization approaches are typically structured in a similar way: they use model-based analysis to predict extra-functional properties (which are used to compare architecture candidates), and pair the analysis with an optimization strategy (search mechanism). Model-based analysis is needed to efficiently handle the large number of possible candidates, and to allow optimization in early stages of development, but there is always a limit to how accurate the optimization can be, since model-based analysis inherently relies on abstractions and estimations, and thus gives approximate results. On the other hand we have the more accurate execution-based optimization which uses performance measurements for comparing different architecture candidates. However, optimization based purely on runtime measurements is typically too time consuming to be feasible, as each candidate has to be implemented and executed. Even when it is possible to specify candidate-specific information as parameters external to the code, and thus reuse the same code for all candidates, it still takes longer time to execute and measure, than to simulate most extra-functional properties.

With model-driven engineering, we can utilize design models not only for performance prediction, but also for automatically generating code. A model of the system can be used to generate the implementa-

Figure 6.1: Combined model-based and execution-based architecture optimization

tion, instrumented with code for measuring the extra-functional properties of interest. This makes it possible to combine model-based and execution-based optimization, and leverage the speed of the former and the accuracy of the latter.

Figure 6.1 illustrates the combined model-based and execution-based architecture optimization method we propose. A system representation in the form of a system model (or system models) is used as input. From the model, by means of automatic model transformations (model-to-model and/or model-to-text), an analysis model is generated, from which the extra-functional properties of interest can be predicted.

Similarly, instrumented implementation code is also generated automatically, from which the extra-functional properties of interest can be measured during execution.

The method first runs an optimization cycle based on performance predictions by model analysis (and/or simulation). It uses the analysis model mentioned above, and complements it in each iteration of the optimization with candidate-specific information in order to analyze and evaluate a particular architecture candidate. Since model analysis is performed on an abstraction of the system, it can typically produce performance predictions faster than an execution-based approach can provide performance measurements. Therefore, model-based optimization is used to quickly assess a large number of candidate architectures and thus rapidly find a good candidate. Then, execution-based optimization takes over. In each iteration, it complements the generated code with candidate-specific information in order to execute and evaluate a particular architecture candidate¹. It is comparatively slower, but it leverages the good candidate identified by model-based optimization as its starting point, and can thus be run for fewer iterations. In return, the extra-functional properties that are obtained by runtime measurements are more accurate than the model-based predictions. In other words, we combine the speed of model-based analysis with the accuracy of performance measurements. The final output of the method is twofold: (i) the resulting architecture specification, and (ii) an implementation of the system.

6.2 Model-based and execution-based task allocation optimization

The method described in the previous section is general and can thus be used in different contexts. We instantiated the method in our task allocation framework, resulting in the extended framework depicted in Figure 6.2. In the remainder of this section we describe the extended framework.

¹The figure assumes that the code is the same for every candidate, with candidate-specific information given as parameters external to the code. If this is not possible, and the code is different for every candidate, the code generation step has to be redone for each candidate.

As part of the architecture design phase, using UML [21] and the MARTE profile [22], the software architect defines the software and hardware models of the system. The former defines the software architecture of the system being built, in terms of tasks and the connections between them, while the latter specifies the hardware platform, including the number of available cores and the type of scheduler for each core. In addition to these models, the architect can also define a set of initial affinity specifications, to be used as starting points for the optimization mechanism. This is however optional, as the framework can automatically generate the desired number of initial affinity specifications. This part is unchanged from the original framework.

The framework navigates the software and hardware models designed using UML and MARTE, and by means of automatic model-to-text transformations defined using the Xtend language [23] generates (i) a simulation model and (ii) instrumented system code. The former is a Java class file that represents an executable model of the system to be fed as input into our task simulator. The latter is an implementation of the system in C, instrumented with code to extract the extra-functional properties of interest. In other words, the former is used to obtain performance predictions, while the latter gives performance measurements. The former is unchanged from the original framework, while the latter is new.

Having generated all the necessary artefacts, in the next phase optimization (depicted by the two gray rounded rectangles in Figure 6.2) is performed. The optimization mechanism tries to minimize the end-to-end response times for a particular task chain. Please note that in the extended framework we did not implement all the performance metrics of interest supported in the original framework (task deadline misses, chain deadline misses and core load are currently unsupported). This does however not reduce the conceptual value of the extended framework, rather it represents an implementation detail.

The overall structure of the two optimization modules is the same, as can be seen in Figure 6.2. In each iteration, the framework generates a new architecture candidate by making a small modification to the best candidate found thus far, derives relevant performance metrics for the new candidate, and determines whether it was an improvement over the best candidate. This continues until the stop criterion has been met (see inner stop criterion in Section 4.2.2). In difference from the original framework, in the extended framework the stop criterion has to be specified separately for the two optimization modules.

Figure 6.2: Framework for combined model-based and execution-based task allocation optimization

Where the two modules differ is how the relevant performance metrics are derived: in the case of the model-based optimization module we have performance predictions obtained by model simulation, while in the case of the execution-based optimization module we have performance measurements obtained by executing the generated system code. These complementary ingredients, one based on model simulation and the other based on system execution, represent the novelty of the optimization mechanism.

Optimization starts with the model-based module (the left gray rounded rectangle in Figure 6.2). In each iteration, the simulation model is complemented with information about a particular affinity specification and as such represents a particular allocation candidate. By executing the simulation model we obtain the average end-to-end response times for a particular chain. This performance metric is used to compare the current candidate to the best one found so far. As mentioned above, the best candidate is kept, and used to generate a new candidate to be tested in the next iteration. Since model simulation is faster than executing the system, we use model-based optimization to quickly converge to a good affinity specification.

Having done this, the execution-based module of the optimization (the right gray rounded rectangle in Figure 6.2) takes over, using the affinity specification identified by the model-based optimization module as its starting point. In each iteration, the generated code is complemented with a particular affinity specification, representing a particular allocation candidate. Since executing the system in order to obtain performance measurements is slower than performing model simulation, execution-based optimization is done for fewer iterations compared to model-based optimization. Having executed the generated code and measured the end-to-end response time, we can compare the current allocation candidate to the best one found thus far. Again, the best candidate is kept and used to propose a new candidate for the next iteration.

As with the original framework, the optimization mechanism is restarted for each initial affinity specification provided by the software architect and for a desired number of randomly generated starting allocations. The stop criteria supported are the same as in the original framework (see outer stop criteria in Section 4.2.2): a fixed number of restarts or a given time limit. When the optimization mechanism stops, it compares the best affinity specifications identified in each restart, and

outputs the best one among them as the final affinity specification. Both optimization modules can either use the same candidate comparison criteria and search heuristic, or these could be tailored for each optimization module separately.

6.3 Experiment

In this section we present an experiment performed using the extended framework. The point of the experiment is to demonstrate the feasibility of the combined model-based and execution-based architecture optimization approach. We start by describing the experiment setup, and then present and discuss the results.

6.3.1 Experiment setup

The optimization goal in the experiment was to minimize the average end-to-end response time for a particular task chain. The software and hardware models of the experiment system are shown in Figure 6.3. We aimed for a representative system that contains task chains of varying length. The chain whose end-to-end response time was optimized is the one consisting of tasks $t1$ to $t10$. The execution platform had two cores, each running a preemptive priority-based scheduler.

The experiment consisted of four parts: pure model-based optimization, pure execution-based optimization, and two runs of combined optimization with different settings. All four parts were executed for roughly the same amount of time, so we could compare the end results. In every experiment part, we repeated 100 optimization runs, in order to be able to generalize the results. All optimization runs started from the same initial affinity specification with an equal number of tasks allocated to each core: tasks $t1$ to $t3$ allocated to core 0, tasks $t4$ to $t6$ allocated to core 1, tasks $t7$ to $t9$ to core 0 and so on. Regarding the inner stop criteria, the following was used for the different parts of the experiment:

- i model-based optimization — 1250 optimization iterations,
- ii execution-based optimization — 270 optimization iterations,
- iii combined optimization 1 — 1000 model-based optimization iterations followed by 50 execution-based optimization iterations,
- iv combined optimization 2 — 350 model-based optimization iterations followed by 200 execution-based optimization iterations.

Figure 6.3: Software and hardware models of the experiment system

As mentioned above, the numbers were chosen so that all parts of the experiment take roughly the same time.

Each execution took about 4.5 times longer than each simulation. Each simulation was performed for 4000 simulation steps (clock ticks), while each execution was run for 4 seconds. This was chosen in order:

- i to run the system long enough (10 hyper-periods) to capture the average behavior, and
- ii to result in a similar number of activations of the optimized chain during simulation and execution.

In all optimization runs, for proposing the candidate to be tested in the next iteration, we used a simple heuristic that randomly relocated one task to a different core. The heuristic was deliberately kept simple, in order for the experiment to focus on the core novelty of the approach rather than on the heuristic. Regarding the comparison criteria, the following was used: of two candidates, the one with the lower end-to-end response time for the selected chain is the better one.

Before running the optimization, from the models defined in UML and MARTE, the framework automatically generated:

- i a simulation model in Java, to be fed as input to the model-based optimization module, and
- ii an instrumented implementation in C, to be used by the execution-based optimization module.

In the implementation, the task model was the same as the one used in the experiments described in Section 3.3 and Section 4.3.1. Each task was a POSIX thread with read-execute-write semantics — it first reads input data, then performs calculations and finally writes output data. Task calculations were represented by a loop that repeats a simple addition operation, and they take a considerably longer time than reading and writing data. The implementation was instrumented with code for measuring the end-to-end response times for a particular chain.

The hardware and software environments were also the same as the ones used for the experiments described in Section 3.3 and Section 4.3.1: an Intel Core 2 Duo E6700 processor [18], running the 32-bit version of the Ubuntu 12.04 LTS operating system (kernel version 3.2.29), which was patched with the PREEMPT RT patch (version 3.2.29-rt44) [19].

6.3.2 Experiment results

Figure 6.4 depicts the results of the experiment. Each point in the diagrams shows the selected chain's end-to-end response time for the best affinity specification found after a particular number of optimization iterations, as an average of 100 optimization runs. In the subfigures, model-based optimization iterations are depicted with squares, while execution-based optimization iterations are depicted with triangles. An important aspect that needs to be clarified is the jump in the response time value at iteration 1001 in Figure 6.4c and iteration 351 in Figure 6.4d. This is not a deterioration or a step back in the optimization. The jump is expected when moving from the model-based optimization module to the execution-based one. It depends on the accuracy of the model-based performance prediction, and could in the general case be a negative or a positive jump. This also means that the last value before the jump cannot be directly compared with the final value, since one is a predicted value from the simulation mechanism, while the other is a measured value, and thus has a higher confidence.

In Figure 6.5 we can see all four experiment parts plotted against time rather than optimization iteration. The colors are used to make it easier to identify which line in the figure corresponds to which subfigure of Figure 6.4.

The similar duration of all parts of the experiment allows us to compare their end results. We can see that for this particular system, combined optimization gives the best result on average, followed by execution-based optimization and then model-based optimization, although the differences are not significant. Please note that for model-based optimization (Figure 6.4a), we added a single execution at the end, in order to make all final results measurements. Since it takes a longer time to execute and measure the chain end-to-end response time than to simulate it, execution-based optimization performs a smaller number of iterations than model-based and combined optimization in the same amount of time. The smaller number of iterations translates to not being able to minimize the response time as much as combined optimization. The fact that model-based optimization gave the worst result is probably due to a small difference in what simulation and execution consider to be a good allocation, which led the search in a slightly wrong direction.

(a) Model-based optimization

(b) Execution-based optimization

(c) Combined model-based and execution-based optimization (1000 + 50 iter.)

(d) Combined model-based and execution-based optimization (350 + 200 iter.)

Figure 6.4: Experiment results

Figure 6.5: Experiment results over time

An important aspect of the combined optimization method is choosing the point when to perform the change from the model-based to the execution-based optimization module. Providing a general rule that works for all approaches is not possible, since this depends on how well model-based analysis and execution agree on what is a good allocation, and on the difference in duration between obtaining performance predictions and performance measurements. For instance, if performance prediction is imprecise, the model-based optimization module could guide the search in a slightly wrong direction before handing over to the execution-based module. Also, the smaller the difference in duration between model-based analysis and execution, the smaller the motivation for running the model-based optimization module for a large number of iterations. On the other hand, the closer that model-based analysis comes to execution in terms of accuracy, the better it is to run many iterations of the model-based module before switching to the execution-based one. The same is valid when model-based analysis is much faster than execution. For instance, when performance predictions can be obtained analytically (without simulation), the difference

in duration between model-based analysis and execution is bigger than in the experiment, meaning that the model-based module could process a much larger part of the search space than the execution-based module in the same amount of time.

Since our generated tasks have a uniform distribution of execution times, the performance predictions reflect the performance measurements quite well. In the case when the task code is generated from detailed functional specifications, the difference between the two would be bigger, thus increasing the motivation for the execution-based optimization module.

6.4 Summary

Since model-based analysis gives performance predictions using abstractions and approximations, architecture optimization based only on model analysis has limited accuracy. However, in a model-driven approach, code generation facilities can be utilized to generate the complete system implementation instrumented with code for extracting performance measurements for the extra-functional properties of interest. This way, model-based optimization can be combined with execution-based optimization that uses performance measurements for candidate comparison.

In this chapter, we presented a novel method for combined model-based and execution-based architecture optimization. The method relies on model-based optimization to quickly converge to a good architecture candidate, which is then used as the starting point for the slower but more accurate execution-based optimization. We applied the method in our framework for optimizing task allocation and carried out an experiment that, although limited in size and scope, demonstrated the feasibility and value of the approach. For the concrete system used in the experiment, combined optimization found on average a slightly better solution than both pure execution-based and pure model-based optimization.

Chapter 7

Related work

In this chapter we present work related to the contributions of the thesis. As our research has two perspectives, a software engineering one and a real-time one, we survey related work from two viewpoints, each given in its respective section. First, we discuss related approaches for model-based performance analysis and architecture optimization, and then we present how task allocation is addressed in real-time systems.

7.1 Model-based performance analysis and architecture optimization

Our research is done in the area of software engineering, or more specifically, model-driven engineering (MDE) [4] and software performance engineering (SPE) [5]. MDE and SPE have a general goal (among other goals) of lifting the level of abstraction from code to models, thus simplifying software development. Models are used both to reason about the extra-functional properties of the system under development, and as a specification from which the implementation can be automatically generated. Regarding the former, models enable obtaining performance predictions already at an early stage of development, prior to the implementation, via model-based analysis. These model-based performance predictions make it possible to quickly assess a large number of system configurations (e.g., deployment of software components to hardware nodes), thus enabling architecture optimization at an early development

stage. Architecture optimization typically aims to find near-optimal architectures, since finding the optimal ones is usually not feasible within reasonable time due to the large search space.

Performance analysis and architecture optimization for embedded systems

Here we present several approaches for model-driven performance analysis and architecture optimization in embedded systems. However, in contrast to our work, none of the approaches we are aware of are specifically tailored for soft real-time multicore embedded systems.

ProCom [26] is a component-based and model-based approach for developing automotive embedded systems. It has a notion of a rich component, which is a set of models, documentation and code. Through different modeling formalisms, ProCom can be used to predict worst-case execution times [27], end-to-end response times [28] and resource usage [29]. It also provides support for automatic synthesis of code from the models, however it does not support architecture optimization.

A defacto industry standard for model-based analysis of embedded systems is Mathworks Matlab and Simulink [24]. This is an integrated development environment that comes with built-in libraries of blocks (for instance the Stateflow toolbox for defining and executing state charts) that enable analysis and simulation of embedded systems, and ultimately code generation. It is also possible to define custom blocks using the Matlab programming language, which makes Simulink extendable with custom analysis and simulation techniques. As mentioned in Section 4.2, the prototype implementation of our framework uses Matlab and Simulink as the modeling technology and simulation engine.

DeepCompas [30] is an analysis framework for predicting performance-related properties of real-time embedded systems. The basis of the approach are composable models of individual software components and of hardware blocks, which are then synthesized into an executable model of the system. Simulation-based analysis of the executable model results in predicted performance properties for the system. DeepCompas also makes a step towards architecture optimization, by providing support for performing trade-off analysis between several architecture alternatives.

ProMARTES [31] is a framework for performance analysis and architecture optimization of component-based real-time distributed systems. It combines schedulability analysis for worst-case behavior and simulation for average-case behavior, and uses multi-objective optimization to find Pareto-optimal solutions.

ArcheOpterix [32] is a framework for optimizing embedded systems modeled using the Architecture Analysis and Description Language (AADL) [33]. It supports several categories of extra-functional properties, such as reliability and performance. The optimization mechanism employs various general-purpose optimization techniques including genetic algorithms, Bayesian learning and hill climbing. The approach can account for uncertainties of design-time parameter estimations through its extension called Robust ArcheOpterix [34], which searches for architecture candidates that reduce the impact of the uncertainties.

PerOpteryx [35] is a framework for optimizing component-based software architectures. Its distinctive feature is the extensible degrees of freedom model. A degree of freedom is a modifiable aspect of a software architecture that the optimization process is allowed to change in search for good architecture candidates. Conceptually, PerOpteryx is independent of the considered extra-functional properties, software architecture meta-modeling language, and degrees of freedom, while the current implementation is based on Palladio [36], a component-based framework for business information systems.

Additional approaches (not limited to the embedded systems domain) can be found in the survey of model-based performance prediction approaches by Balsamo et al. [37], the survey of component-based approaches for performance evaluation by Koziolok [38], and the survey of architecture optimization approaches by Aleti et al. [1].

Approach classification

In addition to surveying the state of the art in architecture optimization, Aleti et al. aim to integrate the existing diverse approaches by defining a common classification taxonomy. In this subsection, we classify the work presented in the thesis against the taxonomy, in order to enable easier comparison of our work to related research. In running text, we combine a brief introduction to the taxonomy with the classification of our approach, and then we summarize the classification in Figure 7.1.

For a more detailed definition of the taxonomy, we advise the reader to consult [1].

The first level of the taxonomy classifies related work against three fundamental categories: the optimization problem, the optimization solution and the optimization validation. Each fundamental category is divided into subcategories, where each subcategory has a number of predefined classification values to choose from.

Problem Our approach targets the embedded systems domain in the design-time phase. It supports a set of performance-related quality attributes (extra-functional properties). The dimensionality of our optimization problem is single-objective with a constraint set on timing: we minimize end-to-end response times for task chains, while keeping the number of chain deadline misses under a selected limit.

Solution We use an architectural model specified in UML as input. Regarding quality evaluation, or in other words the means to obtain values for the quality attributes of interest, our simulation mechanism belongs to the model-based category. In addition, we support monitored execution of the system, but the taxonomy does not provide a suitable classification value. Allocation is the supported degree of freedom in our approach. Additionally, the software architect can choose different scheduling options for each core, but currently the optimization mechanism cannot vary the selected schedulers during the search process. Since we use local search paired with a custom heuristic, our optimization strategy is approximative. We discard unfeasible allocations, meaning that our constraint handling approach is prohibition.

Validation The validation category differentiates between approach validation and optimization validation. The former includes assessing the practicality and accuracy of the approach, more specifically evaluating the accuracy of the mechanism for obtaining values for the quality attributes of interest. This in our case is the simulation mechanism, which we validated by an experiment where we compared the output of the simulation mechanism to a running system, for multiple system architectures. However, the experiment classification value of the taxonomy requires randomly generated problems, which was not the case with our validation, so we classify the approach validation as a simple

Figure 7.1: Classification of our approach against the architecture optimization taxonomy defined in [1]

example. Furthermore, our approach also uses performance measurements, which do not require explicit validation. However, there is no way to specify this using the taxonomy. In contrast to approach validation, optimization validation focuses on the optimization strategy. Here we used comparison with random search (comparison to the random heuristic) and internal comparison (comparison to the load heuristic).

Combined approaches for performance prediction and performance measurement

Next, we discuss approaches that combine performance prediction with performance measurement, related to our Research contribution 4. We are not aware of any optimization method that combines performance prediction and performance measurement, regardless of the system domain. Sailer et al. [39] come close in their approach for allocating tasks to multicore electronic control units (ECUs) in the automotive domain. However, rather than combining simulation-based and execution-based optimization, they run only simulation-based optimization which uses measurements obtained by system execution as input to the simulation mechanism.

A similar concept is found in approaches that support back-propagation, as can, for example, be seen in the work by Ciccozzi et al. [40]. There, models are used both to obtain performance predictions via model-based analysis and to generate the system implementation. The generated code is then executed and obtained measurements are back-propagated to the models. The back-propagated parameter values are used for evaluating the accuracy of the original parameter values used for model-based analysis (and thus implicitly for evaluating the results of the model-based analysis). Furthermore, the back-propagated parameter values are used as input for repeated model-based analysis. The repeated analysis is more accurate, as it now uses parameters that came from measurements, unlike the original parameters which were expert estimates.

7.2 Task allocation in real-time systems

In addition to the software engineering aspect of our work, it also has a real-time aspect. Task allocation, which we tackle in the scope of

our model-based optimization framework, is traditionally addressed as a sub-problem of scheduling for multicore, multiprocessor and distributed real-time systems. In general, scheduling for real-time systems is the process of deciding the execution order of tasks, such that all tasks meet their timing constraints.

Task allocation in multicore and multiprocessor systems

Scheduling for multicore and multiprocessor real-time systems can be grouped into partitioning and global scheduling. In partitioning, tasks are statically allocated to cores/processors, and each core/processor has its own scheduler. This corresponds to bound multiprocessing with no task migration in multicore systems, and asymmetric multiprocessing in multiprocessor systems (see Section 2.2). In global scheduling, which corresponds to symmetric multiprocessing, tasks are governed by a common scheduler which can move them between cores/processors. Recently, also hybrid approaches that combine partitioning and global scheduling have been introduced.

Our approach belongs to the former group. The advantage of task partitioning is that it reduces the multicore/multiprocessor scheduling problem to a set of single-core/uniprocessor ones. Once a partition of tasks (allocation of tasks) to cores/processors has been found, a myriad of well known scheduling techniques and analyses can be applied to each core/processor. As mentioned before, task allocation is analogous to a bin packing problem, which is NP-hard, meaning that finding an optimal allocation in polynomial time is not realistic in the general case [6]. Therefore, approaches for task allocation typically use heuristics paired with uniprocessor schedulability analysis techniques that target hard-real time systems. In contrast, our approach targets soft-real time systems, and pairs our heuristic with model-based analysis (model simulation).

Early approaches for allocating tasks to a multiprocessor were defined by Dhall and Liu [41] — rate monotonic next-fit scheduling and rate monotonic first-fit scheduling. These try to allocate tasks to processors using the next-fit and first-fit heuristics, respectively, while keeping each processor schedulable according to rate monotonic scheduling. In other approaches, additional bin-packing like heuristics (such as best-fit, best-fit decreasing, first-fit decreasing) have been combined with different scheduling algorithms. For instance, Monot et al. [42] define a

heuristic based on the decreasing worst-fit scheme, for allocating tasks on multicore automotive ECUs, with respect to core load. Also, custom heuristics have been developed for the purpose of task allocation. For example, Nemati, Nolte and Behnam [43] define a heuristic for allocating tasks to a multicore platform in such a way that the total amount of blocking time is reduced. More heuristics can be found in the survey by Davis and Burns [44] and the survey by Braun et al. [45].

Many approaches (e.g., [46, 47]) tackle the allocation problem by modeling the task set as a directed acyclic graph (DAG), where nodes represent tasks, while edges represent task dependencies (data or triggering). Heuristics are employed to allocate the DAG to the execution platform. Kwok and Ahmad [48] give a taxonomy of DAG scheduling approaches.

As automotive applications are moving from single-core to multicore processors, we see an increasing interest in task allocation in the automotive domain. Sailer et al. [39] tackle task allocation within the AUTOSAR initiative [49]. They employ a genetic algorithm (elitist non-dominated sorting genetic algorithm (NSGA-II)) to build a Pareto-optimal front with respect to real-time properties, resource consumption and data-communication overhead. In order to obtain performance metrics, they use a simulation tool included in the Timing-Architects Tool Suite [50]. The simulation tool takes as input detailed information such as instruction count for each task, which impedes using the methodology prior to having implemented the system. In a similar approach, Schmidhuber et al. [51] also use the Timing-Architects Tool Suite and genetic algorithms, but rather than building a Pareto-optimal front of solutions, they evaluate similarities between highly ranked solutions in order to derive task allocation guidelines. In another approach targeting AUTOSAR, Faragardi et al. [52] define three allocation strategies, two based on simulated annealing and one on max-min ant system, for allocating tasks to cores in such a way that communication overhead is minimized.

In contrast to the listed hard real-time approaches, Devi [53] focuses on soft real-time multicore systems. However, her approach uses global scheduling, and, similarly to the hard real-time approaches, is based on schedulability analysis and focuses on worst-case behavior, while our approach relies on model simulation and focuses on average-case behavior.

Task allocation in distributed systems

Other than for multicore and multiprocessor systems, task allocation has also been extensively studied in the domain of distributed real-time systems, but again focusing mostly on hard real-time. In distributed systems, computing nodes (which can have multiple cores or processors) are interconnected via a network. Thus, approaches targeting such systems must address network scheduling in addition to task allocation and task scheduling. Although each of these problems is NP-hard on its own, and thus some tackle the problems separately [54], most approaches advocate a holistic solution and address them together [55, 56, 57].

As with multicore and multiprocessor systems, many approaches for task allocation in distributed systems use heuristics. For instance, Briceno et al. [58] define two custom heuristics for allocating multiple data-intensive applications (each specified as a DAG) onto a distributed cluster of multicore machines. Kang et al. [59] base their approach on honeybee mating optimization (which combines the principles of simulated annealing, genetic algorithms and custom local search heuristics) with the goal of maximizing the system reliability.

The influence of allocation on task communication

Next, we look at how related approaches address the influence that task allocation has on task communication. Typically, task communication is addressed by providing an abstract mechanism that could then be tailored for different memory hierarchies. For instance, Faragardi et al. [52] give a mechanism that supports specifying four levels of task communication latencies — intra-core communication with an L1 cache hit, intra-core communication with an L1 cache miss, inter-core communication via L2 cache, and inter-core communication via the shared memory or L3 cache. Similarly for distributed systems, Yadav et al. [60] require the specification of a so called inter-task communication cost matrix.

While such abstract mechanisms are flexible and can thus be used for different types of systems, they also pose a burden on the user of the method. In our approach we took a different direction — rather than providing an abstract mechanism that the user of the method has to instantiate, we chose to investigate the concrete influence of task alloca-

tion on task communication for the type of systems that we support — we identified that the difference in intra-core and inter-core task communication can be ignored when obtaining model-based performance predictions. Thus, we have relieved the user of our method (the software architect) from the burden of dealing with task communication cost explicitly.

Chapter 8

Conclusion

As the features of today's embedded systems become more and more advanced and complex, the systems exhibit increasing performance demands. Over the recent years we have seen that multicore technology, previously successfully used in general-purpose computer systems, made an entry into the domain of embedded systems. While on the one hand it does provide a higher performance capacity, on the other hand it introduces a problem that was not present with single-core processors — how to allocate software tasks to the cores of the hardware platform to achieve satisfactory performance. In this thesis we made several important contributions towards solving this broad problem.

The thesis proposes a model-based framework for optimizing the allocation of software tasks onto the cores of a multicore embedded soft-real time system. The definition and implementation of the framework represents the broadest contribution of the thesis, in the sense that the research question it answers (see Research question 1 in Section 1.1) corresponds to the overall goal of the thesis, while the remaining research questions (Research question 2, 3 and 4) originate from this question. The remaining contributions build upon the framework and each tackles a specific aspect of the framework. The second contribution clarifies how the duration of task communication changes depending on whether communicating tasks are allocated to the same core or to different cores. This information is used by the simulation mechanism of the framework in order to give accurate performance predictions. The performance predictions are in turn used by the optimization mecha-

nism of the framework when comparing different allocation candidates. A key part of the optimization mechanism — the third contribution of the thesis — is a novel heuristic that guides the search for a good task allocation, based on information about how tasks delay each other during simulation. Finally, the fourth contribution is an extension of the framework — an enhancement of the prediction-based optimization process with performance measurements.

In the remainder of this chapter we first summarize the contributions and then provide a high-level discussion about whether they fulfill the overall goal of the thesis. In the last section of the chapter we present future work by providing some ideas how the results of the thesis can be advanced.

8.1 Contributions

In this section we summarize the contributions of the thesis, discuss their limitations, and present possible extensions of the contributions.

A model-based framework for task allocation optimization in soft real-time multicore embedded systems

Summary We defined a general framework for automatically finding a good allocation of software tasks to the cores of a soft real-time multicore embedded system. The goal of the framework is to quickly find an allocation that yields satisfactory performance, and it is based on the following key principles: model simulation, local search paired with a domain-specific heuristic, and flexible start conditions and stop criteria for the optimization mechanism. The targeted user of the framework is the software architect.

The optimization mechanism of the framework searches for a good allocation by making a small modification to the best allocation candidate found so far, and then testing in the next iteration whether the modification resulted in an improvement or not. Whether an allocation candidate exhibits satisfactory performance or not is tested using the simulation mechanism of the framework. The simulation mechanism outputs performance predictions that are used to compare different allocation candidates to each other.

The framework is generic and agnostic to particular extra-functional properties and implementation details. Our provided implementation is based on the following: it uses UML, MARTE and Xtend as the modeling technologies; end-to-end response times, task and chain deadline misses and core load as the extra-functional properties of interest; a custom simulation engine implemented in Java; and a custom domain-specific optimization heuristic. We validated the implemented simulation mechanism by comparing its performance predictions to measurements performed on a running system.

Limitations The framework targets soft real-time systems (systems where timing is crucial for correctness but occasional deadline misses can be tolerated). It assumes homogeneous multicore systems (the cores are identical), with uniform shared memory access (the time it takes to access a particular memory location is the same for each core, except when accessing local cache) and bound multiprocessing (each core has a local scheduler and task migration is not allowed under runtime).

Task triggering of multiple dimensionality and triggering by outside events is not supported. The simulation mechanism is not optimized for simulation speed, and it does not allow more than two active instances of the same chain at the same time.

Possible extensions Here we give two possible directions for improving the framework: the first is relaxing the listed implementation limitations (described in the second paragraph above) and generally increasing the user-friendliness of the implemented tools, while the other is providing support for additional extra-functional properties. The latter requires specifying additional information as part of the software and hardware models. Furthermore, it requires modifications to the simulation mechanism, so it can provide performance predictions for the said property. Also, the comparison of allocation candidates would have to be redefined, in order to use the new property in the optimization mechanism.

The impact of task allocation on communication duration in the context of design-time model-based performance prediction

Summary In order for the simulation mechanism of the framework to give accurate performance predictions, we needed to identify how task allocation influences the duration of task communication, which in turn has a high impact on the extra-functional properties of interest. By performing a series of experiments, we showed that this difference was smaller than anticipated. We argued that in the context of design-time model-based analysis, the difference in duration between intra-core and inter-core communication can be ignored without hurting the accuracy of the performance predictions. This means that our simulation mechanism does not need to include a separate communication penalty for inter-core task communication.

Limitations The experiment results are valid for systems with uniform shared memory access. The experiment did not cover a scenario that is very memory intensive.

Possible extensions The experiment could be extended for a memory intensive scenario, where tasks might often have to wait before being allowed to access the RAM, due to high activity on the memory bus, which might favor intra-core communication. Furthermore, other than in the context of design-time model-based analysis, a similar experiment could be performed in the context of runtime optimizations. When the code is available and the data access patterns are known, grouping communicating tasks on the same core might be a good strategy for minimizing communication duration.

A novel heuristic for task allocation optimization with respect to end-to-end response times

Summary A key part of the optimization mechanism is our domain-specific heuristic. It is in charge of proposing a new allocation candidate to be tested in the next iteration of the optimization. The heuristic guides the search by identifying tasks that are problematic from a timing perspective and moving them to a less loaded core. Whether a task is

problematic is decided based on a structure called the delay matrix, containing information about how the tasks delayed each other during simulation. The experiments that we performed showed that the delay matrix heuristic converges faster and finds an overall better solution than two reference heuristics we used for comparison.

Limitations The heuristic assumes systems that are built from software modules, where delaying between the modules can be identified (in order to enable building a delay matrix).

Possible extensions We would like to combine the idea of the delay matrix with additional task relocation strategies, other than the one based on core load. Furthermore, we consider the heuristic suitable for any approach for design-time optimization of timing related properties, even outside of the domain of soft real-time embedded systems, including, for instance, distributed systems and general-purpose multicore systems, as long as a delay matrix can be built. Finally, it would be interesting to test the heuristic for runtime optimization.

A novel task allocation optimization method that combines performance prediction and performance measurement

Summary Model-based analysis is necessary both in order to handle the large search space and to enable optimization at an early phase of development. However, since model-based analysis inherently uses approximations and estimates, the accuracy of model-based optimization is limited. On the other hand, execution-based optimization can be more accurate, but is generally too time consuming to be feasible. Having the possibility to generate the implementation from models, as model-driven engineering advocates, we can combine the two. We defined a novel architecture optimization method that leverages both the speed of model-based performance predictions and the accuracy of execution-based performance measurements. The method proposes running model-based optimization first, to quickly converge to a good solution, and then continuing with execution-based optimization, which is run for fewer iterations but leverages the aforementioned good solution as its starting point. Other than defining the general method, we

also applied it to our task allocation framework. Using the extended framework we performed an experiment that demonstrated the feasibility the method.

Limitations The method assumes that the complete implementation instrumented with code for obtaining performance measurements can be generated from the design models.

Possible extensions The experiment was performed optimizing only chain end-to-end response time using a simple random heuristic. Support for building the delay matrix can be added to the generated code and then the experiment can be repeated for chain end-to-end response times and deadline misses, using the delay matrix in both the model-based and execution-based optimization module. Furthermore, an interesting experiment would be to use separate heuristics for the two optimization modules, each specially tailored to the respective optimization module. Finally, as the method is general, we would like to apply the idea of combined model-based and execution-based optimization in a framework that is significantly distinct from ours (for instance, one that does not use model simulation, but obtains performance predictions analytically).

8.2 Discussion

We stated in Section 1.1 that the goal of the thesis was to advance the way multicore embedded systems are developed, through an automatic mechanism for deciding early in the development process which software task to run on which processing core of the hardware platform. In this section we discuss whether this was achieved.

Overall, the thesis defines a well-rounded approach that contributes to the state of the art in multicore embedded systems and architecture optimization. All the individual contributions of the thesis have been verified by experiments. The approach provides methods and tools for modeling of real-time multicore embedded systems, for analyzing their timing-related extra-functional properties, and for automatically finding an allocation of tasks that exhibits satisfactory performance. The modeling formalisms and tools are straightforward and do not come

with a steep learning curve. Analyzing the performance for a particular task allocation gives both numerical and graphical feedback.

The framework represents the broadest contribution of the thesis, and the remaining individual contributions fit well in the scope of framework, but they are also usable on their own. The identified difference between intra-core and inter-core communication duration is applicable to any design-time performance analysis method. The delay matrix heuristic can be applied in an arbitrary approach for architecture optimization with respect to timing properties, as it identifies tasks that are problematic from a timing perspective. Finally, the idea of a combined model-based and execution-based optimization method, while implemented in and demonstrated using our framework, is broader than the framework and worthwhile to test in other approaches for architecture optimization.

Next, we reason briefly about applying the approach within an existing development process. We consider that a company developing the type of systems our approach targets would already have to have in place a more or less formal way of specifying the hardware and software architecture at an early stage of system development. To use our approach, the company would either have to specify the hardware and software architecture using our modeling formalisms, or an automatic transformation between the company's modeling technologies and our hardware and software model would have to be developed. In the former case, the change would be straightforward as our modeling formalisms are simple, but they might lack other information the company uses later in the development process. In the latter case, defining such a transformation would be easy, provided that the company's design models contain all the information that our hardware and software models require. Once a suitable allocation has been identified with the help of our framework, it can be used as a specification for subsequent implementation activities in the company. Or, using the extended framework, the implementation could be automatically generated, which would require redefining the code generation so that it supports the targeted software and hardware environments of the company's execution platform. Of course, prior to using the approach, we would have to check if the systems the company develops fit our assumptions and limitations. If that is not the case, our approach would have to be extended in order to relax the limitations in question. Also, before using the approach, the company's software architect would have to be given training, but this

should not require a long time. Altogether, we believe the approach is usable and applicable in a industrial context, but a through case-study would have to be performed in order to confirm this.

In general, we consider that the thesis, despite the listed assumptions and limitations, fulfills its goal and that it provides valuable new insight to academics and practitioners in the field of multicore embedded systems and architecture optimization, both as a whole or through its specific parts.

8.3 Future work

In this section we discuss how the overall results of the thesis can be advanced and how the thesis could spawn new research directions.

One interesting direction of future work is relaxing the scope limitations of the framework and thus broadening its applicability. There are no conceptual barriers to using the framework for other types of hardware architectures, such as multicore systems with heterogeneous cores, manycore systems (multicore systems with a particularly high number of cores, tens or hundreds), non-uniform memory access systems (systems where the time it takes to access a particular memory address depends on its location in relation to a particular processor), heterogeneous systems (systems that use more than one type of processing unit, the typical combination being a multicore CPU, a graphics processing unit (GPU) and FPGA) or distributed systems (systems that have multiple nodes connected via a network). However, for each new type of hardware architecture we would like to use the framework for, we would have to modify and revalidate the simulation mechanism, in order to reflect the specifics of this type of hardware. For instance, in multicore systems with heterogeneous cores, tasks would have a different execution time on different cores. Similarly in heterogeneous systems tasks would have a different execution time on the CPU than on the GPU or on the FPGA. To account for this, we would need to have a mechanism to scale the tasks' BCET and WCET parameters for the different processing units. In distributed systems, the simulation mechanism would have to take into account the additional delay that the network connections impose on communicating tasks.

The approach targets soft real-time systems, but could also be extended for hard real-time systems. There, the absence of deadline

misses has to be guaranteed beforehand, and this is typically done by performing schedulability analysis. Schedulability analysis and simulation could be combined in the following way: only allocations passing the schedulability test are considered feasible and are then simulated to establish how good they are in terms of e.g. average response time.

If we step out of the domain of embedded systems, another possible extension of the scope are general-purpose multicore/multiprocessor systems, where the approach can be used for load balancing or improving the throughput. However, such systems usually use symmetric multiprocessing, meaning that threads can move between cores during runtime. This means that the general ideas from our approach can be used, but considerable parts of the framework would have to be redefined.

The combined optimization method has a potential improvement aspect that we did not explore in the scope of the thesis. When executing the system, we could measure not only the extra-functional properties of interest (the ones used for comparing the candidates to each other), but also the parameters used for model-based analysis, and back-propagate them to the design models, as proposed in [40]. This way, these parameters could evolve from, for example, expert estimates into measured values, which should result in increased accuracy of model-based analysis. Increasing the accuracy of model-based analysis may mean that we can increase the portion of the search space the model-based optimization module processes before handing over to the execution-based optimization module, and thus find an overall better solution without sacrificing accuracy or speed.

Bibliography

- [1] A. Aleti, B. Buhnova, L. Grunske, A. Koziulek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE Transactions on Software Engineering*, 2013.
- [2] A Helmerich, N Koch, L Mandel, P Braun, P Dornbusch, A Gruler, P Keil, R Leisibach, J Romberg, B Schätz, T Wild, and G Wimmel. Study of worldwide trends and R&D programmes in embedded systems in view of maximising the impact of a technology platform in the area. *Final Report for the European Comission*, 2005.
- [3] T. Ulversoy. Software defined radio: Challenges and opportunities. *Communications Surveys Tutorials, IEEE*, 2010.
- [4] Stuart Kent. Model Driven Engineering. In *Third International Conference on Integrated Formal Methods (iFM)*, 2002.
- [5] Murray Woodside, Greg Franks, and Dorina C. Petriu. The Future of Software Performance Engineering. In *Workshop on the Future of Software Engineering*, 2007.
- [6] David S. Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, Dept. of Mathematics, 1973.
- [7] Juraj Feljan, Jan Carlson, and Tiberiu Seceseanu. Towards a model-based approach for allocating tasks to multicore processors. In *38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2012.
- [8] Juraj Feljan and Jan Carlson. The impact of intra-core and inter-core task communication on architectural analysis of multicore embed-

- ded systems. In *8th International Conference on Software Engineering Advances (ICSEA)*, 2013.
- [9] Juraj Feljan and Jan Carlson. Task allocation optimization for multicore embedded systems. In *40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2014.
- [10] Federico Cicciozzi and Juraj Feljan. Model-driven deployment optimization for multicore embedded real-time systems: the OptimAll approach. In *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2014.
- [11] Juraj Feljan, Federico Cicciozzi, Jan Carlson, and Ivica Crnkovic. Enhancing model-based architecture optimization with monitored system runs. In *41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2015.
- [12] ARM Cortex-A Series. <http://www.arm.com/products/processors/cortex-a/index.php>, [Accessed: 2015-09-09].
- [13] DragonBoard 810. <https://developer.qualcomm.com/hardware/dragonboard-810>, [Accessed: 2015-09-09].
- [14] Gordon E. Moore. Cramming more components into integrated circuits. *Electronics*, 38(8), 1965.
- [15] Multicore Processing. http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/sys_arch/smp.html, [Accessed: 2015-08-13].
- [16] D. Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf, [Accessed: 2015-02-06].
- [17] U. Drepper. What every programmer should know about memory. <http://lwn.net/Articles/250967>, [Accessed: 2015-02-06].
- [18] Intel Core 2 Duo E6700 processor. http://ark.intel.com/products/27251/Intel-Core2-Duo-Processor-E6700-4M-Cache-2_66-GHz-1066-MHz-FSB, [Accessed: 2015-02-06].

- [19] PREEMPT RT patch. https://rt.wiki.kernel.org/index.php/Main_Page, [Accessed: 2015-02-06].
- [20] POSIX. <http://pubs.opengroup.org/onlinepubs/9699919799>, [Accessed: 2015-02-06].
- [21] Unified Modeling Language (UML). <http://www.uml.org/>, [Accessed: 2014-11-28].
- [22] The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems. <http://www.omgarte.org/>, [Accessed: 2014-11-28].
- [23] Xtend programming language. <http://www.eclipse.org/xtend/documentation.html>, [Accessed: 2014-11-28].
- [24] Mathworks Matlab and Simulink. <http://www.mathworks.com/>, [Accessed: 2014-11-28].
- [25] Juraj Feljan. Validation code and complete results. <https://goo.gl/TqKkgg>, [Accessed: 2015-07-02].
- [26] Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnković. A component model for control-intensive distributed embedded systems. In *11th International Symposium on Component-Based Software Engineering (CBSE)*, 2008.
- [27] Jan Carlson. Timing Analysis of Component-based Embedded Systems. In *The 15th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE)*, 2012.
- [28] Rafia Inam, Jan Carlson, Mikael Sjödín, and Jiří Kunčar. Predictable integration and reuse of executable real-time components. *Journal of Systems and Software*, 2014.
- [29] Aneta Vulgarakis Feljan. *A Resource-Aware Framework for Designing Predictable Component-Based Embedded Systems*. PhD thesis, Mälardalen University, 2012.
- [30] Egor Bondarev. *Design-time performance analysis of component-based real-time systems*. PhD thesis, Eindhoven University of Technology, 2009.

- [31] Konstantinos Triantafyllidis, Egor Bondarev, and Peter H. N. de With. Guided Rule-Based Multi-objective Optimization for Real-Time Distributed Systems. In *41th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2015.
- [32] A. Aleti, S. Björnander, L. Grunske, and I. Meedeniya. ArcheOpterix: An extendable tool for architecture optimization of AADL models. In *ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, 2009.
- [33] SAE standard, no. AS5506. Architecture Analysis & Design Language (AADL). 2012.
- [34] I. Meedeniya, A. Aleti, I. Avazpour, and Ayman Amin. Robust ArcheOpterix: Architecture optimization of embedded systems under uncertainty. In *2nd International Workshop on Software Engineering for Embedded Systems (SEES)*, 2012.
- [35] Anne Koziulek. *Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes*. PhD thesis, Karlsruhe Institute of Technology, 2011.
- [36] Steffen Becker, Heiko Koziulek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1), 2009.
- [37] S. Balsamo, A. di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5), 2004.
- [38] Heiko Koziulek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation, Special Issue on Software and Performance*, 67(8), 2010.
- [39] A. Sailer, S. Schmidhuber, M. Deubzer, M. Alfranseder, M. Mucha, and J. Mottok. Optimizing the task allocation step for multi-core processors within AUTOSAR. In *International Conference on Applied Electronics*, 2013.
- [40] Federico Ciczozzi, Antonio Cicchetti, and Mikael Sjödin. Round-trip Support for Extra-functional Property Management in Model-driven Engineering of Embedded Systems. *Information and Software Technology*, 55(6), 2013.

- [41] Sudarshan K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1), 1978.
- [42] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion. Multisource Software on Multicore Automotive ECUs — Combining Runnable Sequencing With Task Scheduling. *IEEE Transactions on Industrial Electronics*, 59(10), 2012.
- [43] Farhang Nemati, Thomas Nolte, and Moris Behnam. Partitioning Real-Time Systems on Multiprocessors with Shared Resources. In *14th International Conference On Principles Of Distributed Systems (OPODIS)*, 2010.
- [44] Robert I. Davis and Alan Burns. A Survey of Hard Real-time Scheduling for Multiprocessor Systems. *ACM Computing Surveys*, 43(4), 2011.
- [45] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Blni, 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), 2001.
- [46] H. Turner and J. White. Multi-core Deployment Optimization Using Simulated Annealing and Ant Colony Optimization. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2013.
- [47] A. Monot, A. Vulgarakis, and M. Behnam. PASA: Framework for partitioning and scheduling automation applications on multicore controllers. In *19th IEEE International Conference on Emerging Technology and Factory Automation (ETFA)*, 2014.
- [48] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, 31(4), 1999.
- [49] AUTOSAR. <http://www.autosar.org/>, [Accessed: 2015-09-28].
- [50] The Timing-Architects Tool Suite. <http://www.timing-architects.com/>, [Accessed: 2015-09-28].

- [51] Stefan Schmidhuber, Michael Deubzer, Ralph Mader, Michael Niemetz, and Jrgen Mottok. Towards the Derivation of Guidelines for the Deployment of Real-Time Tasks on a Multicore Processor. In *Model-Based Safety and Assessment*, volume 8822 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014.
- [52] Hamid Reza Faragardi, Kristian Sandström, Björn Lisper, and Thomas Nolte. A Communication-Aware Solution Framework for Mapping AUTOSAR Runnables on Multi-core Systems. In *19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2014.
- [53] Umamaheswari C. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill, 2006.
- [54] C.E. Houstics. Module allocation of real-time applications to distributed systems. *IEEE Transactions on Software Engineering*, 16(7), 1990.
- [55] K.W. Tindell, A. Burns, and A.J. Wellings. Allocating hard real-time tasks: An NP-Hard problem made easy. *Real-Time Systems*, 4(2), 1992.
- [56] Krithi Ramamritham. Allocation and Scheduling of Precedence-Related Periodic Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 6(4), 1995.
- [57] Qi Zhu, Haibo Zeng, Wei Zheng, Marco Di Natale, and Alberto Sangiovanni-Vincentelli. Optimization of task allocation and priority assignment in hard real-time distributed systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(4), 2013.
- [58] Luis Diego Briceno, Jay Smith, Howard Jay Siegel, Anthony A. Maciejewski, Paul Maxwell, Russ Wakefield, Abdulla Al-Qawasmeh, Ron Chi-Lung Chiang, and Jiayin Li. Robust static resource allocation of DAGs in a heterogeneous multicore system. *Journal of Parallel and Distributed Computing*, 73(12), 2013.
- [59] Qin-Ma Kang, Hong He, Hui-Min Song, and Rong Deng. Task allocation for maximizing reliability of distributed computing systems using honeybee mating optimization. *Journal of Systems and Software*, 83(11), 2010.

- [60] PK Yadav, MP Singh, and Kuldeep Sharma. An optimal task allocation model for system cost analysis in heterogeneous distributed computing systems: A heuristic approach. *International Journal of Computer Applications*, 28(4), 2011.

Appendix A

Impact of allocation on task communication — Experiment results

In the appendix we list the complete results of the experiment described in Chapter 3. For each of the 10 sizes of shared data we give the results in diagram and tabular form.

Comm	Stride	Mean	Median	Std dev
intra	1	1.502	1.503	0.012360
	2	1.503	1.504	0.008235
	3	1.499	1.500	0.010980
	4	1.502	1.501	0.009413
	8	1.503	1.503	0.009071
	12	1.513	1.512	0.008504
	16	1.512	1.512	0.010000
	24	1.516	1.515	0.007973
	32	1.521	1.519	0.008815
inter	1	1.684	1.681	0.040450
	2	1.751	1.748	0.040890
	3	1.992	1.989	0.043510
	4	2.201	2.203	0.051940
	8	2.768	2.760	0.070600
	12	3.057	3.060	0.087660
	16	3.276	3.271	0.044340
	24	3.452	3.444	0.126000
	32	3.824	3.827	0.068350

Figure A.1: 128 elements

Figure A.2: 256 elements

Comm	Stride	Mean	Median	Std dev
intra	1	4.053	4.053	0.01768
	2	4.056	4.056	0.02181
	3	4.037	4.034	0.02301
	4	4.057	4.053	0.01964
	8	4.081	4.079	0.02486
	12	4.271	4.270	0.02708
	16	4.759	4.759	0.02106
	24	4.759	4.759	0.02235
	32	4.763	4.763	0.02161
inter	1	4.256	4.255	0.04140
	2	4.652	4.650	0.05303
	3	5.464	5.459	0.05090
	4	6.424	6.466	0.12720
	8	8.200	8.180	0.08310
	12	9.408	9.402	0.15260
	16	9.094	9.082	0.14000
	24	9.584	9.602	0.26130
	32	7.662	7.658	0.10350

Figure A.3: 512 elements

Comm	Stride	Mean	Median	Std dev
intra	1	27.78	27.78	0.15230
	2	28.53	28.54	0.09635
	3	28.76	28.75	0.03701
	4	29.04	29.03	0.09420
	8	30.17	30.18	0.06000
	12	32.09	32.12	0.08949
	16	33.71	33.74	0.09159
	24	33.80	33.81	0.06978
	32	33.50	33.52	0.08963
inter	1	28.18	28.16	0.10560
	2	29.81	29.86	0.19550
	3	33.04	33.05	0.13970
	4	34.68	34.90	0.43980
	8	35.59	35.59	0.23020
	12	36.88	36.88	0.17670
	16	37.79	37.80	0.19410
	24	38.06	38.07	0.28280
	32	36.30	36.31	0.18630

Figure A.4: 4 096 elements

Comm	Stride	Mean	Median	Std dev
intra	1	55.62	55.62	0.10620
	2	56.57	56.56	0.08258
	3	56.96	56.95	0.03281
	4	57.32	57.37	0.42000
	8	59.59	59.65	0.27350
	12	63.43	63.51	0.17330
	16	66.72	66.79	0.32010
	24	66.97	67.01	0.28860
	32	66.82	66.82	0.28040
inter	1	55.56	55.52	0.16190
	2	57.45	57.44	0.29110
	3	61.00	61.01	0.28800
	4	62.66	63.04	0.72460
	8	64.40	64.36	0.23120
	12	68.14	68.18	0.24870
	16	70.70	70.73	0.33780
	24	71.10	71.14	0.33300
	32	69.51	69.52	0.23290

Figure A.5: 8 192 elements

Comm	Stride	Mean	Median	Std dev
intra	1	111.1	110.8	1.952
	2	114.3	112.4	4.036
	3	123.0	123.5	1.582
	4	122.7	123.9	3.289
	8	126.7	126.9	0.958
	12	118.1	118.4	1.644
	16	113.0	113.0	1.077
	24	113.5	113.8	1.010
	32	121.7	122.3	1.991
inter	1	112.1	110.3	3.757
	2	121.3	121.3	0.920
	3	123.9	124.1	0.940
	4	126.9	127.0	1.277
	8	130.0	130.2	1.420
	12	121.8	121.5	2.183
	16	116.1	116.2	1.427
	24	116.2	116.2	1.264
	32	120.4	120.5	2.172

Figure A.6: 16 384 elements

Comm	Stride	Mean	Median	Std dev
intra	1	1055	1055	1.745
	2	1075	1075	1.654
	3	1103	1103	1.038
	4	1128	1127	6.456
	8	2298	2298	6.546
	12	3118	3118	7.154
	16	3844	3843	11.670
	24	5185	5186	24.270
	32	5931	5932	20.320
inter	1	1050	1049	3.010
	2	1071	1074	4.725
	3	1093	1093	2.366
	4	1167	1166	4.202
	8	2341	2342	6.702
	12	3133	3131	12.480
	16	3854	3851	14.000
	24	5235	5236	23.350
	32	5921	5921	21.490

Figure A.7: 262 144 elements

Comm	Stride	Mean	Median	Std dev
intra	1	2114	2115	3.733
	2	2183	2183	3.371
	3	2361	2361	2.467
	4	2727	2727	5.075
	8	4167	4168	13.750
	12	5457	5456	28.330
	16	6741	6739	50.620
	24	9852	9861	45.370
	32	11275	11273	41.270
inter	1	2105	2106	4.494
	2	2182	2181	4.445
	3	2370	2370	2.863
	4	2757	2757	5.415
	8	4200	4199	18.020
	12	5455	5454	22.750
	16	6846	6846	59.140
	24	9852	9859	48.230
	32	11270	11271	37.630

Figure A.8: 524 288 elements

Comm	Stride	Mean	Median	Std dev
intra	1	4252	4252	5.377
	2	4455	4455	3.759
	3	4836	4836	6.396
	4	5311	5310	10.950
	8	7478	7475	37.390
	12	10158	10156	86.150
	16	12620	12623	136.900
	24	19049	19074	98.080
	32	21912	21906	73.130
inter	1	4246	4245	6.372
	2	4463	4460	10.480
	3	4855	4852	14.460
	4	5351	5348	14.890
	8	7529	7522	54.050
	12	10556	10556	32.440
	16	12746	12752	129.200
	24	19090	19108	77.610
	32	21885	21880	75.120

Figure A.9: 1 048 576 elements

Comm	Stride	Mean	Median	Std dev
intra	1	5356	5355	4.508
	2	5578	5577	5.843
	3	6008	6007	7.890
	4	6513	6513	9.756
	8	9169	9164	63.720
	12	12418	12419	106.500
	16	15546	15550	174.000
	24	23619	23648	131.800
	32	27197	27190	98.740
inter	1	5353	5352	5.630
	2	5598	5595	12.900
	3	6034	6031	15.410
	4	6568	6564	21.320
	8	9170	9163	66.4200
	12	12496	12495	114.600
	16	15531	15535	178.600
	24	23656	23685	111.700
	32	27202	27193	81.430

Figure A.10: 1 310 720 elements