Automatic Parallelization for Embedded Multi-Core Systems using High-Level Cost Models

Dissertation
zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik
von

Daniel Alexander Cordes

Dortmund
2013

Tag der mündlichen Prüfung:	11. November 2013
Dekan / Dekanin:	Prof. Dr. Gernot A. Fink
Gutachter / Gutachterinnen:	Prof. Dr. Peter Marwedel
	Prof. Dr. Albert Cohen

Acknowledgments

First of all, I would like to thank my advisor Prof. Dr. Peter Marwedel for his commitment and the possibility to accomplish the research leading to this PhD thesis. Without his initial ideas and his advices, this thesis would not have been feasible in this form. I would also like to thank him for the interesting work which I could contribute to the MNEMEE European research project. This project and the following funding allowed me to participate in many interesting project meetings and conferences all over the world. I would also like to thank Prof. Dr. Albert Cohen for his commitment to participate as an external reviewer for this PhD thesis. Special thanks go to Dr. Michael Engel for his advices over the years, the various technical discussions, and his intensive proof-reading support for the originated publications and this thesis.

The collaborating work with all my colleges at our Department of Computer Science XII also had a big influence on the quality of this work. I would also like to thank Florian Schmoll, my office colleague, who always found time for various extensive technical and non-technical discussions. Additional acknowledgments go to Andreas Heinig who provided the required operating system as well as the middleware necessary to evaluate the approaches of this thesis. Besides, I would like to thank Olaf Neugebauer, Timon Kelter, Helena Kotthaus, Björn Bönninghoff, and Jan C. Kleinsorge for their technical feedback and proof-reading support. It was an enjoyable time for me while working in this excellent research team.

Above all, I would like to mention that it would not be possible for me to write this thesis without the love and support of my family. I owe special thanks to my girlfriend Regina Fritsch, my parents Ernst Walter and Claudia Catharina Cordes as well as my brother Steven Cordes. All of them play an important role in my life and are always there for me. Therefore, I dedicate this work to them.

Additional thanks go to the European Community since parts of the research leading to this thesis have received funding from the European Community's MNEMEE Project which was part of the Seventh Framework Programme FP7 under grant agreement no 216224. Later parts of the work have been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A3.

Abstract

Nowadays, embedded and cyber-physical systems are utilized in nearly all operational areas in order to support and enrich peoples' everyday life. To cope with the demands imposed by modern embedded systems, the employment of Multiprocessor System-on-Chip (MPSoC) devices is often the most profitable solution. However, many embedded applications are still written in a sequential way. In order to benefit from the multiple cores available on those devices, the application code has to be divided into concurrently executed tasks. Since performing this partitioning manually is an error-prone and also time-consuming job, many automatic parallelization approaches were developed in the past. Most of these existing approaches were developed in the context of high-performance and desktop computers so that their applicability to embedded devices is limited. Many new challenges arise if applications should be ported to embedded MPSoCs in an efficient way. Therefore, novel parallelization techniques were developed in the context of this thesis that are tailored towards special requirements demanded by embedded multi-core devices.

All approaches presented in this thesis are based on sophisticated parallelization techniques employing high-level cost models to estimate the benefit of parallel execution. This enables the creation of well-balanced tasks, which is essential if applications should be parallelized efficiently. In addition, several other requirements of embedded devices are covered, like the consideration of multiple objectives simultaneously. As a result, beneficial trade-offs between several objectives, like, e.g., energy consumption and execution time can be found enabling the extraction of solutions which are highly optimized for a specific application scenario.

To be applicable to many embedded application domains, approaches extracting different kinds of parallelism were also developed. The structure of the global parallelization approach facilitates the combination of different approaches in a plug-and-play fashion. Thus, the advantages of multiple parallelization techniques can easily be combined. Finally, in addition to parallelization approaches for homogeneous MPSoCs, optimized ones for heterogeneous devices were also developed in this thesis since the trend towards heterogeneous multi-core architectures is inexorable.

To the best of the author's knowledge, most of these objectives and especially their combination were not covered by existing parallelization frameworks, so far. By combining all of them, a parallelization framework that is well optimized for embedded multi-core devices was developed in the context of this thesis.

Publications

Parts of this thesis have been published in proceedings of the following conferences and workshops (in chronological order):

1. Daniel Cordes, *Loop Analysis for a WCET-optimizing Compiler Based on Abstract Interpretation and Polylib (in German)*, Master's thesis, Technische Universtität Dortmund, 2008.

2. Niklas Holsti, Jan Gustafsson, Guillem Bernat, Clément Ballabriga, Armelle Bonenfant, Roman Bourgade, Hugues Cassé, Daniel Cordes, Albrecht Kadlec, Raimund Kirner, Jens Knoop, Paul Lokuciejewski and Merriam, *WCET Tool Challenge 2008: Report*, in Proceedings of the International Workshop on Worst-Case Execution Time Analysis (WCET), Prague, Czech Republic, 2008.

3. Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel, *A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models*, in Proceedings of the International Symposium on Code Generation and Optimization (CGO), Seattle, Washington, USA, 2009.

4. Daniel Cordes and Peter Marwedel, *An automatic parallelization tool for embedded systems, based on hierarchical task graphs*, Research Poster at the Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications (DEPCP'2010) (DATE Workshop), Dresden, Germany, 2010.

5. Christos Baloukas, Lazaros Papadopoulos, Dimitrios Soudris, Sander Stuijk, Olivera Jovanovic, Florian Schmoll, Daniel Cordes, Robert Pyka, Arindam Mallik, Stylianos Mamagkakis, François Capman, Séverin Collet, Nikolaos Mitas, and Dimitrios Kritharidis, *Mapping Embedded Applications on MPSoCs: The MNEMEE Approach*, in Proceedings of the International Symposium on VLSI (ISVLSI), Washington, DC, USA, 2010.

6. Daniel Cordes, Peter Marwedel, and Arindam Mallik, *Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming*, in Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Scottsdale, Arizona, USA, 2010.

7. Daniel Cordes, Andreas Heinig, Peter Marwedel, and Arindam Mallik, *Automatic Extraction of Pipeline Parallelism for Embedded Software Using Linear Programming*, in Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS), Tainan, Taiwan, 2011.

8. Daniel Cordes and Peter Marwedel, *Multi-objective aware extraction of task-level parallelism using genetic algorithms*, in Proceedings of the Design, Automation and Test in Europe Conference Exhibition (DATE), Dresden, Germany, 2012.

9. Daniel Cordes and Peter Marwedel. *PAXES – Parallelism Extraction for Embedded Systems: Three Approaches – One Tool* Research Poster at the Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications (DEPCP'2010) (DATE Workshop), Dresden, Germany, 2012.

10. Daniel Cordes, Michael Engel, Peter Marwedel, and Olaf Neugebauer, *Automatic extraction of multi-objective aware pipeline parallelism using genetic algorithms*, in Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Tampere, Finland, 2012.

11. Daniel Cordes, Michael Engel, Olaf Neugebauer, and Peter Marwedel, *Automatic Extraction of Multi-Objective Aware Parallelism for Heterogeneous MPSoCs*, in Proceedings of the International Workshop on Multi-/Many-core Computing Systems (MuCoCoS), Edinburgh, Scotland, UK, 2013.

12. Daniel Cordes, Michael Engel, Olaf Neugebauer, and Peter Marwedel, *Automatic Extraction of Pipeline Parallelism for Embedded Heterogeneous Multi-Core Platforms*, in Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES), Montreal, Canada, 2013.

13. Daniel Cordes, Michael Engel, Olaf Neugebauer, and Peter Marwedel, *Automatic Extraction of Task-Level Parallelism for Heterogeneous MPSoCs*, in Proceedings of the International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI), Lyon, France, 2013.

Contents

List of Abbreviations

AHTG	Augmented Hierarchical Task Graph
DMA	Direct Memory Access
DSP	Digital Signal Processor
DSWP	Decoupled Software Pipelining
DVFS	Dynamic Voltage and Frequency Scaling
DVS	Dynamic Voltage Scaling
FCDG	Forward Control Dependence Graph
GA	Genetic Algorithm
HSCG	Hierarchical Structured Control-Flow Graph
HTG	Hierarchical Task Graph
ILP	Integer Linear Programming
KPN	Kahn Process Network
LP	Linear Programming
MNEMEE	Memory Management Technology for Adaptive and Efficient Design of Embedded Systems
MoC	Model of Computation
MPI	Message Passing Interface
MPSoC	Multiprocessor System-on-Chip
PA4RES	Parallelization for Resource Restricted Embedded Systems
PDG	Program Dependence Graph
PS-DSWP	Parallel-Stage Decoupled Software Pipelining
RAW	Read-after-Write
SA	Simulated Annealing
SDF	Synchronous Data Flow
SDL	Specification and Description Language
TLP	Thread-Level Parallelism
UMA	Unified Memory Architecture
UPC	Unified Parallel C
VLIW	Very Long Instruction Word
WAR	Write-after-Read
WAW	Write-after-Write
WSCDFG	Weighted Statement Control Data Flow Graph

Chapter 1

Introduction

Contents

The importance and pervasiveness of embedded and cyber-physical systems have significantly increased in the last two decades. Like defined in [Mar11], embedded systems are information processing systems that are embedded into enclosing products. Already in 2008 the number of sold smart phones reached 50% of the amount of desktop computer devices sold [Bus09]. Only three years later, in 2011, smart phone shipments surpassed the number of sold PCs [Bus11]. This statistic peaks in numbers which show that the smart phone shipments grew by 87% year over year while the number of sold PCs only grew by 3%, which highlights the increasing importance of embedded devices. But besides obvious devices like smart phones, most people do not recognize or even know when and with how many embedded systems they get into touch every day. Nowadays, a large number of areas of life are difficult or nearly impossible to cope without the help of embedded or cyber-physical devices. For example, traffic would collapse without the help of traffic lights especially in large cities. Also many means of transportation, like, e.g., cars, railways or airplanes are based on several distributed embedded controllers like anti-lock braking systems (ABS), electronic stability protection systems (ESP), airbags or collision avoidance systems. In addition, many people have to rely on their pacemakers to continue their lives in a regular way. These examples cover only a minor part of the enormous number of application domains of embedded systems that can be found in areas like automotive electronics, avionics, railways, telecommunication, the health sector, security, consumer-electronics, fabrication equipment, smart buildings, logistics, robotics, military applications, and many more [Mar11].

To solve the requirements of these examples, highly specialized embedded systems are required. Such systems, in contrast to desktop or high-performance architectures, underlie specific characteristics and resource limitations. These limitations have to be taken into account when embedded systems are designed and optimized. Certainly, one of the most significant aspects is the limited supply of energy caused by battery-driven devices. Other limitations, like, e.g., less computational power, small memories, and limited package space for the designed devices apply to many embedded systems, as well. Consequently, dependability, efficiency and real-time behavior are highly important due to the restrictions and safety-critical application domains mentioned.

Besides these indispensable systems, many other embedded systems exist which try to make the peoples' everyday life more pleasant with multi-media, mobile and other services from the consumer electronics area. Especially for this application domain, an enormous alteration of the devices available on the market could be observed in the last two decades. Simple mobile phones with b/w screens and low resolution displays were replaced by powerful smart phone devices. A study [ITF12] from 2012 has recently shown that 49.7% of all American citizens already exchanged their ordinary mobile phones for such smart phone devices. High-resolution photos and videos are taken via mobile tablet PCs or smart phone devices before they are sent to friends all around the world via E-Mail or Messaging Services. In the home entertainment area, tube televisions were substituted by feature rich smart TVs enabling additional multi-media services like browsing, gaming and social networking while watching a movie. In the domain of automotive systems, not only luxury cars are nowadays equipped with a large network of sensors and embedded processors observing and steering the car while providing the occupants with music, video and navigation services. These examples cover only a minor part of the enormous changes that happened in the domain of multi-media rich embedded systems.

However, to be able to provide all these feature-rich services on mobile embedded devices, the complexity of the employed embedded software has also drastically increased. To fulfill the performance requirements imposed by today's embedded software, the performance of the underlying hardware has to scale, as well. In contrast to desktop and high-performance architectures, most embedded devices are battery-driven. This discourages the solution of just further increasing the cores' frequencies since higher clock frequencies generally lead to higher energy consumption. The last years have shown that the trend towards multi-core architectures seems to be the most promising solution to gain more performance while maintaining energy efficiency. In contrast to the desktop and high-performance community, embedded designers had to draw their conclusions in a much shorter period so that today's embedded systems often benefit from multi-core architectures. The trend towards multi-core architectures is also reflected in Figure 1.1 which compares performance and energy efficiency characteristics of ARM's most popular embedded processors. As can be seen, the shaded single-core processors ARM7 and ARM9 as well as most processors belonging to the Cortex-M and Cortex-R series are located at the bottom left to bottom central-position of the diagram denoting few features, low performance and low energy efficiency. In contrast, the shown multi-core processors ARM11 and the processors of the Cortex-A series are located at a position of the diagram which denotes higher energy efficiency and more performance. However, the performance of the presented single-core processors is also often increased by combining them to form multi-processor architectures. Especially the combination of different and specialized processing units to heterogeneous Multiprocessor

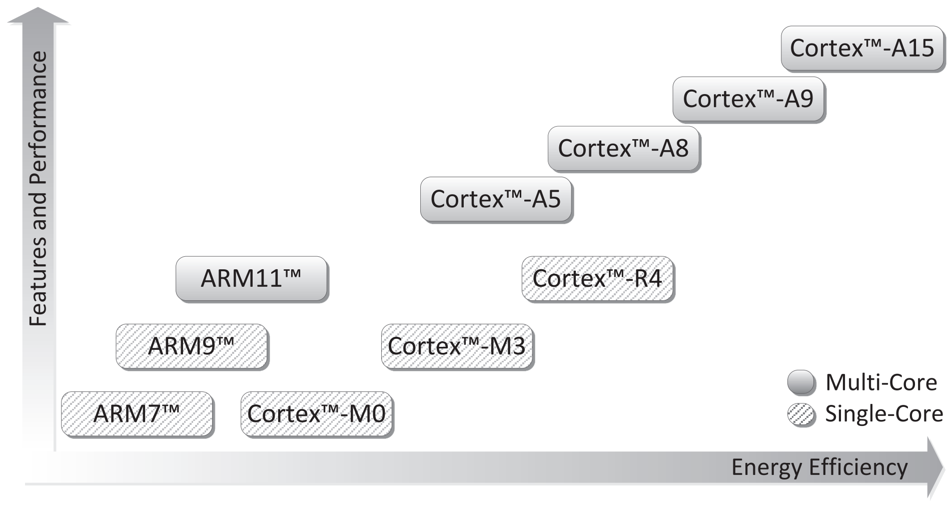


Figure 1.1: Comparison of Performance and Energy Efficiency of Single- and Multi-Core ARM Processors (Based on Figure from [ARM13c])

System-on-Chip (MPSoC) devices revealed gainful trade-offs between performance, energy consumption, and other objectives, which are hard to obtain by homogeneous multi-core devices. The diagram emphasizes that the current state-of-the-art approach to provide feature-rich embedded systems with enough performance lies in the utilization of embedded MPSoC devices. Up to now, this technology has not reached an insurmountable limitation. Hence, it can be assumed that this trend will continue in the future, as well.

1.1 Motivation

By providing multiple, less complex cores on one device, performance can be increased with a lower energy consumption compared to a platform containing only one core operating at a high CPU frequency. As a consequence, Multiprocessor System-on-Chip devices replace traditional embedded single-core architectures wherever more computational power is required. Unfortunately, the benefits of MPSoCs imply additional effort. A single application has to be partitioned into concurrently executed tasks to benefit from the multiple cores available on an MPSoC. Approaches developed earlier, extracting Instruction Level Parallelism for Very Long Instruction Word (VLIW) machines or superscalar processors, are not well applicable to multi-processor systems. Instruction Level Parallelism is too fine-grained since it executes only single statements in parallel. For MPSoC architectures, task creation and communication overhead is too high to use this kind of parallelism. Instead, Thread-Level Parallelism (TLP) is much more coarse-grained so that, in spite of the additional overhead introduced by task creation and communication primitives, this kind of parallelism can still lead to performance gains on multi-core architectures.

Recent surveys, as well as online articles, like, e.g., [TIO13], [Won12], and [Mer07], state that most software – especially for embedded systems – is still developed by using the sequential programming language C. This is not surprising since C compilers exist for a large amount of architectures which eases portability of highly optimized low-overhead application code. Moreover, the C language supports direct access to various hardware components and can be compiled into efficient machine code without the need for runtime interpreters. However, the sequential mindset of this programming language makes it difficult to extract Thread-Level Parallelism to exploit the performance provided by current MPSoCs. Traditionally, one of the following approaches can be used to exploit TLP:

- **Re-Design in High-Level MoC:** In early design phases of embedded systems high-level Models of Computation (MoCs), like, e.g., Kahn Process Networks [Kah74] or State Charts [Har87], are often applied. Many high-level MoCs inherently express parallelism by, e.g., concurrent states or services connected via explicit communication channels. This eases the step of implementing parallelism in later phases. Unfortunately, as already stated, most embedded software was developed in sequential C for decades. Thus, millions of lines of legacy code have to be ported to one of the considered MoCs to benefit from the proposed advantages. Most companies will not invest the large amount of time and money required to port existing functionality. Hence, high-level MoCs may better be applied for new software projects instead of existing ones.

- **Manual Parallelization:** Manual parallelization of existing legacy code seems to be less time consuming than re-designing entire applications in high-level MoCs. Several libraries and language extensions, like, e.g., PThreads [NBF96] and OpenMP [DM98], have been proposed for the C language enabling the extracting of parallelism for sequentially written C-applications. However, the task of manually parallelizing an application for MPSoCs is a particularly error-prone and also time consuming job. The application designer has to deliver the expected functionality of the designed software in an efficient and portable way, optimized for a specific hardware platform, and validated against hundreds of test cases. Besides this time consuming and complicated job, the application designer has to extract and balance tasks as well as to insert communication and synchronization primitives manually. If one of these primitives is missed or placed at a wrong position, the application might get invalid or end up in a deadlock. This is a challenging problem which gets even more complicated if the targeted architecture is equipped with heterogeneous cores with differing performance characteristics.

- **Automatic Parallelization:** Automatic parallelization seems to be the most promising solution since existing legacy code can be divided into concurrently executed tasks automatically. In addition, tasks can be balanced for the available processing units and communication as well as synchronization primitives can be inserted in a correct manner to avoid deadlock scenarios in an automated way. Furthermore, applications just have to be re-compiled by such tools to port them to various hardware platforms efficiently. Fortunately, many researchers invented a large amount of parallelization techniques in the last decades with a focus on desktop- and high-performance architectures. However, the problem of extracting efficient parallelism from sequentially written applications still remains unsolved and many limitations, especially for embedded systems, are not considered, as will be discussed in the following section.

1.2 Automatic Parallelization for Embedded Systems

By combining multiple cores on embedded MPSoCs, new possibilities arise in the context of embedded computing due to the increased performance provided by these devices. However, existing applications have to be parallelized to benefit from the additionally available cores. In an optimal way, this parallelization step should be automated as stated in the previous section. But even though a large amount of parallelization approaches already exists, most of them are optimized for high-performance architectures and are hence not well applicable to resource-restricted embedded MPSoCs. The reason for this limitation is the rise of rather new requirements for embedded multi-core systems, which were hard to foresee from the perspective of the high-performance community. Therefore, new and highly-optimized parallelization techniques are indispensable to utilize embedded MPSoCs efficiently, which was the ambitioned idea for this thesis.

From an embedded perspective, for example, less parallelism and thus less performance is often more. An application which runs several times faster than its given deadline consumes an unnecessarily large amount of energy. As soon as the given deadline is still met, less parallelism may be extracted so that an architecture with fewer and less performant cores can be used which drastically reduces the overall energy consumption. These energy savings lead to a higher battery service life and are important for embedded systems that are often applied in a mobile context. Moreover, due to the simplicity of many embedded devices – in contrast to high-performance architectures – overhead introduced by parallelism (e.g., task creation and communication overhead) is often costly. Accordingly, techniques are necessary to weigh whether parallel execution really accelerates the application. Misjudgment may directly lead to lower performance and higher energy consumption. These and also other requirements and characteristics, like, e.g., heterogeneity of the employed processing units, are mostly not considered by existing parallelization approaches so far. Therefore, the parallelization approaches presented in this thesis were designed to fill this desideratum of missing parallelization tools tailored towards special requirements of embedded MPSoCs.

The most important aspects, which have to be taken into account if applications should be parallelized for embedded MPSoCs, are discussed from a more technical perspective in the following:

Task Balancing: In order to profit most from multi-processor platforms created tasks should be balanced so that all tasks finish nearly at the same time. Otherwise, much performance and also energy may be lost since some of the cores wait for completion of other ones leading to an unbalanced execution behavior. For embedded devices, task balancing is even more complicated and also more important. In contrast to desktop and high-performance architectures, most embedded systems are not constructed as an Unified Memory Architecture (UMA) where each core can access all memory locations. In the case of embedded systems, memory hierarchies are often employed, providing fast and low-energy private memories. Since these memory locations can neither be accessed nor cached by other processing units, communication is often much more expensive for most considered objectives. This even increases the gap between communication and computation costs for embedded systems. As a consequence, extracted parallelism has to be much more coarse-grained for many embedded devices, and it should be clearly deliberated whether parallelization really accelerates the performance of the application. Otherwise, too expensive task creation and communication costs may shadow the benefits of the extracted parallelism and may even lead to a decrease of the application's performance in the worst case.

Multiple Optimization Objectives: Most existing parallelization tools focus on the optimization of the execution time as their only optimization objective. This is, in general, acceptable for desktop and high-performance architectures since large memories and – spoken from an embedded perspective – a nearly unlimited amount of energy are available. The situation changes if parallelization tools focus on embedded devices. Here, multiple objectives should be taken into account simultaneously. It may, for example, be beneficial to reduce the amount of extracted parallelism to put some of the cores into idle mode or to move to an architecture with less provided cores if a given timing criterion is still met. This can reduce the system's energy consumption, heating problems, and can also save chip area.

Online vs. Offline Decisions: Additional overhead for runtime decisions should be avoided as much as possible due to lower computational power of embedded devices and the demand for timing predictability. OpenMP, for example, observes the number of executed tasks to decide at runtime how many tasks will be created if a new parallel region is reached. This behavior is not well suitable for embedded systems. Here, the number of created tasks should be determined off-line at compile time. As a result, the number and computational complexity of the extracted tasks can be optimized for a given architecture.

Type of Parallelism: Applications often profit differently from the available parallelization strategies. This makes it hard to find an optimal parallelization type for various application domains. However, many embedded applications have a streaming-oriented structure and profit from pipeline parallelism [TF10]. In addition, other parallelization strategies, like, e.g., task- and data-level parallelism, should be combined with pipeline parallelism to profit from different parallelization strategies and also a combination of them. Most existing parallelization frameworks focus only on the extraction of one kind of parallelism so that they are not well suitable for a wide range of application domains.

Heterogeneity: The advantages of heterogeneity in type and performance characteristics of the employed processing units are often utilized in embedded MPSoCs. By combining cores with different performance characteristics, less computational intensive tasks can, for example, be mapped to less performant processing units which consume less energy. However, the task of extracting and balancing parallelism for a heterogeneous MPSoC is much more complicated than for homogeneous ones but should also be considered by parallelization tools that are optimized for embedded systems.

1.3 Contribution of this Work

Even though automatic parallelization has been an active research area for decades, existing approaches are not well applicable to parallelize sequentially written applications for embedded MPSoCs. In order to overcome this limitation, this thesis presents a new framework including several novel approaches tailored towards limitations and special requirements that have to be taken into account if applications should be efficiently parallelized for embedded MPSoCs.

As already discussed in the previous section, automatic balancing of extracted tasks is an important aspect to parallelize embedded software efficiently (cf. Task Balancing). To achieve this, sophisticated parallelization approaches based on Integer Linear Programming and Genetic Algorithms are proposed and integrated into the presented framework of this thesis. All approaches employ high-level cost models to evaluate the benefit of different parallel solution candidates. The cost models contain information about task creation, execution, and communication costs for multiple objectives to steer the granularity of the extracted parallelism automatically. Integer Linear Programming is NP-complete in the general case but can be solved efficiently for small or medium-sized problems. Therefore, the framework presented in this thesis employs a hierarchical parallelization approach using an Augmented Hierarchical Task Graph (AHTG) as central intermediate representation. The hierarchical structure of the graph directly correlates to the hierarchical structure of the application's source code. Only a small number of statements are processed at once, due to the segmentation into different hierarchical levels. This enables the use of the sophisticated parallelization algorithms presented in this thesis.

To extract parallelism from applications of different application domains, the framework presented in this thesis combines three different parallelization types, namely, task-level parallelism, loop-level parallelism and pipeline parallelism (cf.

Type of Parallelism). The considered applications can profit either from one or also from a combination of the presented parallelization types. All parallelization extraction techniques are integrated into the hierarchical parallelization approach. By combining different approaches on several hierarchical levels, parallelism with different granularities can be extracted to find solutions optimized for various applications. The framework can also be easily extended by additional parallelization approaches, due to its hierarchical structure.

In contrast to many existing parallelization approaches, this thesis also presents parallelization techniques considering multiple optimization objectives at the same time (cf. Multiple Optimization Objectives). In this way, energy consumption and communication overhead can be optimized in addition to the execution time. The presented techniques return a front of Pareto-optimal solutions to the application designer so that the solution fitting best to a particular application scenario can be chosen as final solution. All considered parallelization types (task-level, loop-level and pipeline parallelism) are developed as multi-objective aware parallelization techniques.

The parallelization techniques presented in this thesis also make use of platform specific information of the target architecture (cf. Online vs. Offline Decisions). This enables platform specific optimizations, taken at compile-time, which are directly integrated into the parallelization process. One example for these offline optimizations is the limitation of the maximum number of extracted tasks. For each presented approach, the upper bound of extractable tasks is set to the number of available processing units by default. Then, the hierarchical approach determines the best combination of different solution candidates which do not exceed the upper task boundary. As a consequence, additional scheduling overhead at runtime can be avoided.

Heterogeneity is one key aspect for current and future embedded MPSoCs (cf. Heterogeneity). By combining cores with different performance characteristics, performance increases can be achieved with lower energy consumption and less heat dissipation issues compared to homogeneous MPSoCs. Unfortunately, the complexity of the parallelization problem drastically increases since these performance variances have to be taken into account if the extracted tasks should be automatically balanced. Therefore, the presented framework also contains novel approaches extracting the considered parallelization types for single and also multiple objectives simultaneously. The approaches optimize tasks for specific processing units and take care that these tasks are mapped to the corresponding cores by processor class premappings. This makes the presented framework also able to utilize heterogeneous multi-core architectures in an efficient way.

To summarize, this thesis presents and combines the following novel aspects:

- Exploitation of platform specific information, like estimated execution and task creation costs

- Automatic balancing of tasks by all presented approaches

- Integration of cost models into sophisticated Integer Linear Programming and Genetic Algorithm-based approaches instead of applying simple heuristics
- Multiple objectives are considered at the same time instead of just optimizing execution time
- Support for homogeneous and heterogeneous architectures
- Combination of several parallelization types with different granularities optimized for embedded applications
- Many decisions are taken offline at compile-time to avoid additional runtime overhead
- Use of a hierarchical divide-and-conquer based parallelization approach to prune the vast solution space of the complex parallelization problem for sophisticated algorithms

1.4 Author's Contribution to this Dissertation

According to §10(2) of the "Promotionsordung der Fakultät für Informatik der Technischen Universität Dortmund vom 29. August 2011", a dissertation within the context of doctoral studies has to contain a separate list that highlights the author's contributions to research and results obtained in cooperation with other researchers. Even though, the approaches presented in this thesis were entirely envisioned and developed by the author of this thesis, Prof. Dr. Peter Marwedel contributed the generalized idea to develop parallelization approaches that are optimized for resource-restricted embedded systems. He also gave useful advices, like, e.g., extending the framework to parallelization approaches for heterogeneous architectures. Thus, the author of this thesis would like to thank him, here, once again. Besides these advices, the following list describes the author's contribution to publications leading to the chapters of this thesis in more detail:

Chapter 3: Chapter 3 presents internals of the parallelization framework and additional tools which are used to map the parallelized applications to an embedded MPSoC. A brief overview of the parallelization framework's internals is also given in [CMM10]. The integration of the parallelization approaches into the framework developed in the European FP7 project MNEMEE was described in [BPS+10]. Several authors cooperated in writing this publication. The author of this thesis provided the text for Chapter II.C in [BPS+10] describing the first parallelization approach developed in the context of thesis. The remainder of Chapter 3 describes the target platforms used for simulation-based evaluations in Chapters 5-8. The employed simulators are the MPARM [BBB+05] and Synopsis's CoMET [Syn13a], which is more recently know as the Virtualizer tool suite [Syn13b]. The provision and adaptation of the platforms were mostly done by Andreas Heinig and was only assisted by the author of this thesis.

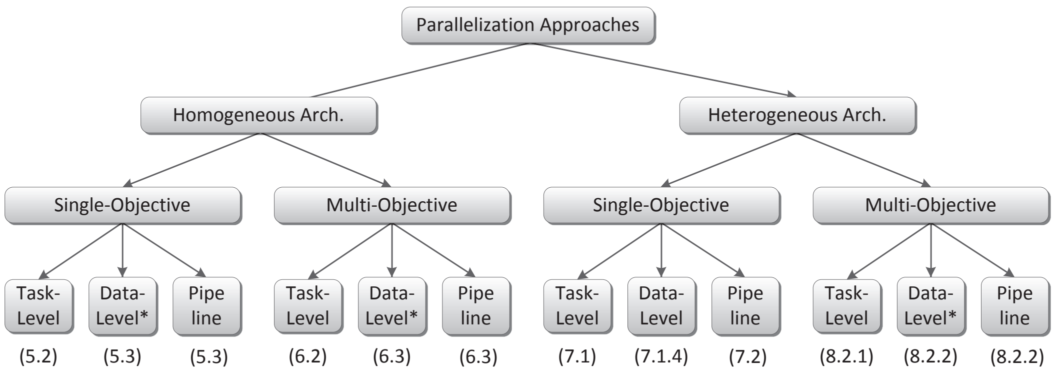
Chapter 4: General concepts of the parallelization framework including the employed intermediate representation are presented in Chapter 4. The general idea described in this chapter was briefly published in [CMM10] and [CHM+11] but was also summarized in other publications of the author for comprehensiveness. Chapter 4 of this thesis contains also many unpublished details about the employed hierarchical parallelization approach. The presented Augmented Hierarchical Task Graph (AHTG) is based on the Hierarchical Task Graph (HTG) presented in [GP94]. The changes made to the graph intermediate representation and the way how to use it to extract parallelism with the sophisticated parallelization techniques presented in this thesis were completely developed by the author of this thesis. However, the author was inspired by many technical discussions with Prof. Dr. Peter Marwedel, members of the department, and participants of the MNEMEE project. Nonetheless, the publications in [CMM10] and [CHM+11] were entirely designed by the author of this thesis. The co-authors of the publications, as well as other members of the department, assisted the author in various ways.

Chapter 5: Two Integer Linear Programming-based parallelization approaches extracting task-level and pipeline parallelism for homogeneous architectures are presented in Chapter 5. Both approaches were entirely designed and developed by the author of this thesis. The corresponding publications [CMM10] and [CHM+11] are entirely based on the author's work. The co-authors of the publications, as well as other members of the department, assisted the author in technical and conceptual discussions especially how to structure the publications. Additionally, many parts were intensively revised by the co-authors.

Chapter 6: Chapter 6 describes multi-objective aware approaches for homogeneous MPSoCs. It is based on the publications presented in [CM12] and [CEM+12]. Both approaches, as well as the publications, were completely designed, developed and mostly written by the author of this thesis. The co-authors, as well as other members of the department, assisted the author in proof-reading, providing small text fragments, and various technical as well as methodological discussions.

Chapter 7: This chapter presents two ILP-based parallelization approach for heterogeneous embedded MPSoCs, which are based on the homogeneous ones presented in Chapter 5. The developed approaches were published in [CEN+13c] and [CEN+13b]. The co-authors of the publications assisted the author of this thesis in writing the introduction of the papers, proof-reading and technical discussions.

Chapter 8: This chapter finally presents the last developed parallelization approaches focusing on multi-objective aware parallelization approaches for heterogeneous MPSoCs, which were published in [CEN+13a]. The co-authors of the publication assisted the author of this thesis in writing the introduction of the paper, proof-reading and technical discussions.



* DoAll Data-Level Parallelism is also extracted as a special case of the pipeline parallelization approaches presented in the corresponding sections.

Figure 1.2: Structure of Thesis and Contributions of Dissertation

1.5 Outline

This section gives an overview about the remaining structure of this thesis. The tree visualized in Figure 1.2 shows the developed parallelization approaches and the corresponding sections describing them. In detail, the following content is described in this thesis:

Chapter 2: A survey of related work is presented in Chapter 2. The approaches selected for discussion are the most relevant ones for the work presented in this thesis. The presented publications are grouped into categories reflecting the different key concepts considered by the approaches of this thesis.

Chapter 3: The internal structure of the parallelization framework with all its sub-tools and its integration into larger projects, like, e.g., the MNEMEE European FP 7 project, is described in Chapter 3. Furthermore, the chapter also presents the target platforms used for evaluation purposes in the remainder of this thesis.

Chapter 4: This chapter presents the general idea and the techniques used to divide the large search space of the parallelization problem into manageable sub-problems. These sub-problems can later be processed by the sophisticated parallelization approaches presented in Chapters 5-8. In more detail, the chapter defines the employed intermediate representation and gives an overview about the structure of the general parallelization algorithm.

Chapter 5: The first parallelization approaches developed within this thesis focus on the extraction of parallelism for homogeneous architectures. They are presented in Chapter 5 and extract task-level, loop-level and pipeline parallelism on basis of Integer Linear Programming (cf. Homogeneous Single-Objective Parallelization in

Figure 1.2). All approaches use high-level cost models to be able to evaluate and balance extracted tasks automatically.

Chapter 6: The parallelization techniques presented in Chapter 6 extract the different parallelization types in a multi-objective aware manner for homogeneous architectures (cf. Homogeneous Multi-Objective Parallelization in Figure 1.2). They are based on Genetic Algorithms and also employ high-level cost models to evaluate the different parallelization candidates.

Chapter 7: Heterogeneous architectures are first considered by the parallelization approaches presented in Chapter 7 (cf. Heterogeneous Single-Objective Parallelization in Figure 1.2). The presented ILP-based techniques are based on the ones presented in Chapter 5. However, the newly presented techniques are extended to, e.g., distinguish between different performance characteristics of the available processing units and to perform a pre-mapping of extracted task to processor classes.

Chapter 8: The parallelization approaches presented in Chapter 8 are a consequent combination of the techniques presented in Chapters 6 and 7. Multi-objective aware parallelization approaches, which are able to extract and balance tasks fully automatically for heterogeneous architectures, are presented there (cf. Heterogeneous Multi-Objective Parallelization in Figure 1.2).

Chapter 9: Finally, Chapter 9 concludes this thesis and provides possible directions for future research.

Chapter 2

Related Work

Contents

Parallel architectures have been invented decades ago. As a consequence, a lot of research effort was invested to utilize those platforms efficiently. Even though many high-level programming languages or extensions to existing ones have been proposed, like PThreads [NBF96], MPI [SOHL+98], OpenMP [DM98], OpenCL [SGS10], X10 [CGS+05], StreamIt [TKA02], UPC [EGCS+03], Cilk [BJK+95], and many others, most of them are not prevalent to parallelize applications in the domain of embedded systems. A different strategy for implementing parallel embedded applications, suggested in the last years, was to model those applications by high-level Models of Computation (MoCs). MoCs, like State Charts [Har87], Petri Nets [Pet66], Specification and Description Language (SDL) [RS82], Kahn Process Networks (KPNs) [Kah74], and Synchronous Data Flow (SDF) [LM87], to mention only some of them inherently express parallelism. However, most existing legacy code for embedded devices is written in sequential C code and most companies are not willing to invest a huge budget to rewrite existing, comprehensive application code in another programming language or to transform it to one of the mentioned MoCs. Hence, the demand for automatic parallelization frameworks was created and increased over the last decades in order to be able to reuse already existing functionality.

Since the research work of this thesis presents approaches which extract coarse-grained Thread-Level Parallelism (TLP) in an automatic fashion, this chapter gives a brief overview about related frameworks and critically discusses the approaches presented in this area. The primary objective of this chapter is to compare functionalities and limitations of existing approaches, which are most relevant to the techniques presented in this thesis. Therefore, those approaches are discussed in more detail instead of aspiring completeness over all presented approaches. The approaches are grouped into different categories in the following sections. Of course, some approaches may be mapped to more than one category since they extract different types of parallelism. In this case, they are placed into the category matching their main contribution.

The structure of this chapter is as follows: Section 2.1 presents approaches extracting coarse-grained task-level parallelism, followed by a discussion of finer grained data-level parallelization techniques in Section 2.2. Pipeline parallelism is highly effective for many embedded applications. Therefore, Section 2.3 opposes different approaches extracting this kind of parallelism. Parallelization techniques considering multiple objectives and heterogeneous architectures are rarely new research topics. Sections 2.4 and 2.5 give a brief overview about the work done in these areas. Finally, Section 2.6 discusses some approaches which go beyond the extraction of TLP before Section 2.7 summarizes features and limitations of existing approaches.

2.1 Task-Level Parallelism

Task-Level parallelism is a coarse-grained kind of Thread-Level Parallelism. Large independent blocks of an application are processed by concurrently executed tasks. These blocks may consist of functions, basic blocks or also single statements, depending on the desired level of granularity (for more details see Section 5.2). Task-Level parallelism can be employed in the context of embedded systems efficiently since in many cases only few data has to be communicated between the different tasks. Therefore, the approaches presented in Section 5.2, Section 6.2, Section 7.1 and Section 8.2.1 propose techniques extracting this kind of parallelism for homogeneous and heterogeneous embedded MPSoCs for one and also for multiple objectives. In the following, the most relevant existing approaches extracting this kind of parallelism are discussed.

Sarkar: The approaches presented by Sarkar et al. in [Sar91a] are most relevant to the task-level parallelization approaches presented in this thesis. They are based on the previous publications in [SH86] and [Sar89] and were integrated into IBM's PTRAN compiler [Sar91b]. Their approaches extract coarse-grained task-level parallelism combined with the extraction of DoAll loops (running independent loop iterations in parallel) from sequential applications written in Fortran. The employed Program Dependence Graph (PDG) is augmented with estimated execution times and transformed into a Forward Control Dependence Graph (FCDG) which is similar to the Augmented Hierarchical Task Graph (AHTG) used as central intermediate representation in this thesis. Both graph representations have in common that backward-dependence edges (pointing in the opposite direction of the regular control flow) are redirected to special exit (or communication-out) nodes. This ensures that the entire graph is cycle-free, enabling the calculation of execution times based on high-level models. Even though the employed intermediate representation and the calculation of estimated execution times are comparable to the ones used in this thesis, Sarkar only exerts a simple greedy-based partitioning heuristic which is applied to one procedure at a time. The approaches presented in this thesis apply sophisticated Integer Linear Programming (ILP) and Genetic Algorithm (GA)-based approaches to extract task-level parallelism from sequential applications. This is only possible since the approaches divide an application into finer-grained chunks based on the hierarchical structure of the given source code. In this way, only a small portion of the application is considered at the same time which drastically reduces the vast solution space of the complex parallelization problem. The approaches presented in this thesis are able to balance the extracted tasks automatically, which is hard to achieve with the greedy heuristics proposed by Sarkar.

Polychronopoulos: Polychronopoulos and Girkar presented automatic scheduling techniques [Pol91] based on Hierarchical Task Graphs (HTGs) [GP94]. The approaches presented in this thesis are based on an Augmented Hierarchical Task Graph (AHTG) and differs from the original HTG representation published in [GP94] in three points. First, the approaches presented by Polychronopoulos et al. only create new hierarchical levels for nested loops of the original application. In contrast, the approaches presented in this thesis create new hierarchical nodes for all hierarchical levels present in the original application. Thus, the hierarchical structure of the graph directly correlates to the hierarchical structure of the application. In addition, the hierarchical granularity is more fine grained in the approach presented in this thesis which enables more sophisticated parallelization algorithms. The second difference to the original presentation in [Pol91] is that the approach presented here adds two new node types to the AHTG, namely communication in- and communication out-nodes. These nodes encapsulate the communication in each hierarchical level. A third difference is that the approach presented in [Pol91] only deliberates about whether it makes sense to generate more tasks for the next hierarchical level based on architectural properties like the number of available processing units. Instead, the approaches presented in this thesis employ the AHTG as a layer to reduce the number of nodes which have to be processed at the same time while parallelizing the application. This means that the approach is also able to group some of the nodes on each hierarchical level to tasks instead of either executing all of them in parallel or executing all of them sequentially.

SUIF: Hall et al. presented coarse-grained thread-level parallelization techniques for C and Fortran applications in, e.g., [HAM+95] and [HAA+96], integrated in the Stanford University Intermediate Format Compiler Framework (SUIF) [WFW+94]. Their techniques employ interprocedural analyses to spawn threads spanning function boundaries. The presented framework is also able to apply analyses and optimization techniques, like, e.g., scalar privatization, reduction recognition, array analyses, and cache optimizations. As a target platform, Hall has chosen an eight-core Digital Alpha Server 8400 which was a high-performance architecture in 1995. Additional work on this framework was presented by, e.g., Sungdo et al. in [MSH00] who evaluated the absent performance gain due to missing data-level parallelization support. Compared to the work presented in this thesis, Hall did not provide any information on cost models which are necessary to balance the created tasks. Moreover, the target architecture was a homogeneous high-performance one which is very different to embedded (heterogeneous) MPSoCs.

MAPS: A more recent task-level parallelization approach was presented by Ceng et al. in [CCS+08]. Their approach is integrated in the MPSoC Application Programming Studio (MAPS) and performs a semi-automatic parallelization technique in which the user can manually steer the granularity of the extracted parallelism. MAPS uses the Tightly-Coupled-Thread framework (TCT) [ZIU+08] as a backend for implementation and simulation of the extracted parallelism. MAPS combines static and dynamic profiling-based information to extract a Weighted Statement Control Data Flow Graph (WSCDFG) annotated with cost information. This cost information is based on a simple multiplication of a configurable execution cost with the number of executions per statement. Based on the WSCDFG, a heuristic clustering algorithm is applied to group statements subsequently to coarse-grained tasks. The heuristic of the original approach was further optimized in [LC10]. Later, C for Process Networks was presented in [CSL11] which allows an application designer to describe parallelism manually through Kahn Process Networks (KPNs) directly in C. Compared to the work presented in this thesis, MAPS extracts a similar kind of parallelism. In contrast to many other approaches, the authors use cost models to balance the extracted tasks. However, the precision of cost information is not very accurate, and the clustering algorithm is based on a simple heuristic compared to the novel sophisticated ILP-based approaches.

2.2 Data-Level Parallelism

Data-level parallelism was not a main focus of this thesis. Only a simple approach is employed extracting data-level parallelism from loops without loop-carried dependencies. Therefore, a direct comparison to the approaches presented in this thesis is omitted in this section. However, many techniques that are able to extract fine-grained data-level parallelism from loops of sequentially written applications have been proposed in the past.

PIPS: The Parallélisation interprocédurale de programmes scientifiques project (PIPS) was first published in [IJT91] representing a modularly source-to-source parallelization framework. Initially, PIPS concentrated on DoAll parallelism by extracting tasks from loops of sequentially written Fortran 77 applications. PIPS's approach employs a Hierarchical Structured Control-Flow Graph (HSCG) as intermediate representation and was designed modular so that it could be extended by other parallelization approaches. Today, over 20 years later, the project is still active and has been extended to, e.g., support the C language and polytope-based parallelization extraction techniques by various approaches like [KAC+96] and [KAI11].

Polaris: The Polaris parallelization compiler was presented in [BEF+95] targeting at the automatic extraction of DoAll parallelism in Fortran 77 applications. To be able to extract parallelism from loops with loop-carried dependencies, optimizations like, e.g., symbolic analysis, induction and reduction variable recognition and array privatization are applied to remove those dependencies. Polaris also contains speculative parallelization for loops whose dependencies could not be determined at compile time. However, such a parallelization technique is often unacceptable for embedded systems since predictability is important for many embedded devices. In addition, Polaris implements function inlining to circumvent inter-procedure analysis techniques. This is also not applicable for embedded devices since it increases the code size and the amount of available memory is often limited. The framework was also extended by other researchers in, e.g., [PE95] and [VE99].

Cetus: The Cetus parallelization compiler was presented in several publications, like, e.g., [LJE04] and [DBM+09]. The framework was written in Java and the source code was published as a freely available research compiler. The authors have taken the Polaris compiler as an inspiring example and tried to create a similar framework for C programs and other target languages instead of Fortran 77. The framework also contains several analysis and code optimization techniques, like, e.g., privatization, reduction variable recognition, and induction variable substitution. However, the framework focuses on data-level parallelism only, and does not apply any cost models to weigh the granularity of parallelism. This can also be seen in their evaluation, where the authors claim that their parallelization tool flow performs better than Intel's icc compiler and the COINS framework [SFF+05]. This comparison is only based on the number of successfully parallelized loops without measuring the performance gain. In reality, many loops may reduce the overall performance if the benefit from parallelization is lower than the required communication and task-creation costs. This is not considered or at least not mentioned in their publications.

Polytope-based approaches: Polytope-based parallelization approaches like the one presented in [Fea96] are favored for extracting data-level parallelism. The iteration space of sequential loop(-nest)s including data- and control flow dependencies is transformed into a form of linear inequalities. Based on this mathematical description, loops can be parallelized in an automated way. Another work in this area was presented in [VNS07] and is based on results of the Compaan project [BRD00; RDK00; TK04]. It extracts Process Networks (PNs) from sequential C applications and is integrated in the MADNESS project's tool flow [CDF+11]. Process Networks can then be efficiently mapped to multiprocessor platforms. Verdoolaege et al. also tried to optimize communication and to determine FIFO buffer sizes. However, their work does not evaluate the speedup of an extracted PN and is limited to static affine nested loop programs so that it can, in general, not be applied to existing applications without manual transformations. Furthermore, no performance estimation is applied to deliberate about the granularity of the extracted parallelism. This may lead to an unbalanced execution behavior which drastically reduces the applications' performance. Their tool was also used in the Daedalus project [NTS+08] to extract the required KPNs for the succeeding optimizations for multiprocessor architectures. Additional polytope-based parallelization frameworks were, e.g., presented in LooPo [GL97] and PLUTO [BHR+08].

Franke: Franke et al. also aim at the extraction of data-level parallelism in [FO05]. The difference to the other presented approaches is that their work focuses on specific issues, which have to be solved if applications are parallelized for embedded multi-core Digital Signal Processor (DSP) platforms. Particular program recovery techniques like array recovery and modulo removal are applied before data-level parallelism is extracted. Moreover, their approach also performs memory optimizations, which use Direct Memory Access (DMA) transfers to optimize the overall performance further for DSP architectures.

Li: Li et al. presented an approach in [LPC12] which extracts data-flow threads from sequentially written imperative programming languages. Therefore, a Program Dependence Graph (PDG) is transformed into SSA form (SSA-PDG) to ease the definition of dependencies. This SSA-PDG is coarsened in the following by several coalescing techniques. The merged nodes of the final graph represent data-flow tasks, which are implemented by a GCC compiler [The13] extension. Unfortunately, the presented approach is only able to exploit data dependencies for scalar variables, which limits its applicability to real world applications. The approach was neither evaluated for embedded applications nor embedded devices.

Pouchet: Pouchet et al. published a framework combining different multi-dimensional loop optimization techniques in [PBC+08]. Their framework was later extended in [PBB+10] to support the extraction of loop-level parallelism. The complexity of their optimization problem as well as the employed optimization algorithms are comparable to the ones used in the context of this thesis. In both cases, a large optimization space is present so that smart optimization techniques had to be chosen. Therefore, Pouchet et al. used an iterative model-driven Genetic Algorithm-based approach in [PBC+08] with specialized mutation and cross-over operators. They have shown that this technique is able to find very good solution candidates in a short amount of time. Similar results could also be observed for the GA-based approaches presented later in the context of this thesis.

Benoit: The dissertation of Benoit [Ben11] describes a source-to-source parallelism adaption tool which is integrated into the GCC Compiler. It combines static and dynamic analysis techniques to detect and describe parallelism opportunities at several hierarchical levels. However, the thesis concentrates on defining a suitable intermediate representation and does not focus on the extraction of parallelism in an automated way like done in this thesis.

2.3 Pipeline Parallelism

Pipeline parallelism is the third kind of thread-level parallelism discussed in this chapter. It can be used to extract efficient parallelism from many embedded applications, especially those which are written in a streaming-oriented structure. Pipeline parallelism can often be applied even if ordinary data-level parallelism (e.g. DoAll loops) cannot be extracted due to loop-carried dependencies. The statements contained in a loop's body are partitioned into disjunctive pipeline stages which execute in an overlapping, pipelined manner on different cores (for more details see Section 5.3). Since pipeline parallelism is often hidden in embedded applications, this thesis also presents different approaches in Section 5.3, 6.3, 7.2, and 8.2.2 which extract this kind of parallelism in an automated fashion for homogeneous and heterogeneous embedded MPSoCs. In the following, the most important existing approaches extracting pipeline parallelism are discussed.

Rangan: Decoupled Software Pipelining (DSWP) was first introduced by Rangan et al. in [RVV+04]. The proposed approach focuses on loops operating on recursive data structures. Rangan et al. manually extracted extremely fine-grained pipeline stages and recognized that the communication delay on a Pentium 4 Xeon Processor is too high to benefit from DSWP. Therefore, they proposed low-latency synchronization arrays for communication between different cores. Compared to the approaches presented in this thesis, Rangan et al. manually applied DSWP to the chosen benchmarks.

Ottoni: Ottoni et al. based their DSWP approach [ORS+05] on the one proposed by Rangan et al. in [RVV+04]. In contrast to the work of Rangan et al., Ottoni et al. extract DSWP fully automatically and integrated their approach into the IMPACT compiler back-end [ACM+98]. Moreover, they have shown that DSWP can be applied efficiently to various loops, even if they are not operating on recursive data structures. Ottoni's approach employs message passing to communicate data between producing and consuming tasks. The algorithm operates on a program dependence graph (PDG) [KA02] which is transformed into a directed acyclic graph (DAG) [Tar72] by clustering the strongly connected components formed by data- and control dependencies. The extracted pipeline stages are balanced by a greedy heuristic which merges the node with the highest estimated cycles (extracted by profiling in the compiler back-end) to the currently processed pipeline stage. This step is repeated until the estimated cycles of the current partition reaches the overall estimated cycles divided by the number of extracted stages. Compared to the approaches presented in this thesis, Ottoni's DSWP approach has some disadvantages. First, even though it extracts pipeline parallelism – which is efficient for many embedded applications – the approach was not optimized nor evaluated for embedded or at least heterogeneous architectures. Second, their approach only extracts disjunctive pipeline stages which are not replicated to further increase the overall performance. And, finally, it operates on assembler level which drastically limits portability, readability and the possibility to present the extracted results to the application designer in a comprehensible form.

Raman: Parallel-Stage Decoupled Software Pipelining (PS-DSWP) was proposed by Raman et al. in [ROR+08] and subsequently continues the work of Ottoni et al. [ORS+05]. Raman has observed that the number of extractable tasks with DSWP is limited by the number of strongly connected components in a loop's body. To increase the amount of extracted parallelism their approach replicates stateless pipeline stages without loop-carried dependencies. Some of the stages are split into concurrently executed sub-tasks, like performed by traditional DoAll loop parallelization methods. Raman's approach is integrated into the VELOCITY research compiler [TBR+06]. Since the pipeline parallelization approaches presented in this thesis are also able to extract pipeline stages, which can also be replicated, the work of Raman et al. is most relevant to this work. However, compared to the approaches presented in this thesis, PS-DSWP is only able to replicate pipeline stages which are stateless, which means that no loop-carried dependencies exist for the stage to be split. In contrast, the pipeline parallelization approaches presented in this thesis are also able to duplicate stages with loop-carried dependencies if the iteration level (the minimum distance of loop iterations between producing and consuming the data) is greater than one. Additionally, Raman's approach employs only a simplistic greedy heuristic to extract the tasks. Only one stateless pipeline stage, the one with the highest estimated execution costs, is replicated at most which drastically reduces the solution quality. Like DSWP, PS-DSWP also employs a platform with multiple high-performance Itanium 2 cores and a low-latency synchronization array for evaluation purposes, which is not comparable to an embedded device.

Tournavitis: Tournavitis et al. presented a profiling-based parallelization framework in [TF09]. Their framework extracts data and control dependencies dynamically by annotating and executing a medium-level Intermediate Representation in the CoSy compiler. The profiling-based approach was later used in [TWF+09] to extract loop level parallelism annotated with OpenMP pragmas automatically. Instead of employing accurate high-level models, the authors used machine learning to decide whether a loop may increase the overall program performance by parallelization. The machine learning-based approach also specifies the iteration scheduling policy used by OpenMP. Their approach seems to be promising, but the evaluation was performed on a workstation equipped with two Intel Dual-Core Xeon 5160 processors, running at 3 GHz and 16 GB of main memory. A second evaluation was performed on a Cell Blade with two 3.2 GHz processors. Both platforms are applied in high-performance computing and it is not clear whether their approaches perform well on embedded devices. Another restriction of the presented work is that it is only able to extract DoAll parallelism from for-loops. This restriction was removed by their following publication in [TF10] which extracts pipeline parallelism from nested loops of streaming applications. Compared to the pipelining-based approaches presented in this thesis, the work of [TF10] is only able to replicate stateless pipeline stages like [ORS+05]. In addition, the work of Tournavitis employs only a simple parallelization heuristic based on a fixed threshold. Also here, the Xeon architecture is used for evaluation.

Thies: The approach presented by Thies et al. [TCA07] assists the programmer in extracting pipeline parallelism by a semi-automatic profiling-based technique. In a first step, the programmer has to group statements of the applications' outer loop to pipeline stages manually. Afterwards, a profiling run is started to extract data and control-flow dependencies. Those are finally visualized in a stream graph with additional profiling-based performance information per pipeline stage. If the programmer is not satisfied with the extracted speedup, he has to redefine pipeline boundaries over several steps. Communication and synchronization directives are finally inserted by the parallelization framework, based on the profiling information. Compared to the approaches presented in this thesis, the work of Thies is not able to extract parallelism fully automatically and only assists the programmer in extracting pipeline parallelism. Also here, the evaluation was performed on a high-performance architecture equipped with two AMD Opteron 270 dual-core processors and 8 GB main memory.

Gordon: Another interesting approach was presented by Gordon et al. in [GTA06]. The authors present an approach combining task, data, and pipeline parallelism into one framework. However, compared to the framework presented in this thesis, the programmer has to rewrite the application in the StreamIt [TKA02] language where parallelism has to be modeled manually by independent actors that use explicit data channels for communication and synchronization. Based on this description, the proposed approach automatically reduces synchronization and communication overhead by splitting tasks at the necessary granularity level. The approach was integrated into the stream compiler presented in [GTK+02].

Wang: The original version of the StreamIt compiler contains only very simplistic greedy-based optimization techniques to find, e.g., good partitionings of a given task graph structure. Therefore, Wang et al. developed a more sophisticated partitioning approach in [WO10] and [Wan11]. This approach is based on machine-learning to estimate the execution time of a solution candidate by finding a comparable parallelized application for which this objective is known. Therefore, a costly simulation of each solution candidate can be omitted which makes the complexity of the large solution space manageable. In this thesis, costly simulations are also avoided by employing high-level cost models. But, unfortunately, Wang's approach is not optimized for embedded applications nor evaluated on embedded architectures.

Pop: Pop et al. recently presented OpenStream in [PC13] based on their former publication in [PC11]. Even though OpenStream is not able to extract parallelism in an automatic fashion (like done by the approaches presented in this thesis), it is mentioned here since it extends the OpenMP API [DM98] such that streaming applications can easily be parallelized by using high-level annotations (C pragmas). OpenMP is the de-facto standard in the high-performance community so that its use in the domain of embedded systems would be desirable, as well. However, in its original form, OpenMP does not support explicit communication in its shared memory model. This is altered by the new annotations provided by OpenStream so that, among others, streaming-based pipeline parallelism can now be expressed efficiently. The authors implemented compilation strategies for the newly inserted pragmas as front and middle-end extensions into the GCC compiler [The13]. Since the proposed techniques of OpenStream are orthogonal to the ones presented in this thesis, OpenStream could also be employed in the future to implement the applications parallelized by the approaches of this thesis. However, the applicability of OpenStream to embedded target architectures has not been evaluated so far.

2.4 Multi-Objective Aware Extraction

In contrast to high-performance architectures, embedded ones are usually battery-driven, contain smaller memories, and miss high-performance communication structures, to mention only some of their limitations. Hence, to parallelize applications efficiently for embedded devices new parallelization approaches need to be developed. One way to achieve this is to find efficient trade-offs between multiple objectives. Most existing approaches try to extract as much parallelism as possible to minimize the execution time as their only optimization objective. However, it could also make sense to reduce the amount of extracted parallelism to move to an architecture providing less processing units if the specified application deadlines are met. In this way, a lot of energy can be saved and the communication overhead is also decreased. This section provides a brief overview about parallelization approaches considering more than only one objective like done in the multi-objective aware parallelization techniques presented in Chapters 6 and 8 of this thesis.

Kadayif: The publication [KKS02] presented by Kadayif et al. is interesting for both, the ILP-based and GA-based approaches presented in this thesis. Kadayif employed Integer Linear Programming to determine the best amount of allocated processing units while considering both, execution time and energy consumption. His approach operates in two steps. In the first one, already parallelized loops of the given application are simulated for one up to eight processing units in isolation to determine profiling-based execution times and energy values. Afterwards, an ILP is applied which can optimize either execution time or energy consumption. However, compared to the ILP-based approaches presented in this thesis, Kadayif's formulations can only determine the best number of processing units used per loop for manually parallelized loop-nests instead of automatically extracting and balancing tasks from sequentially written applications. The GA-based approaches presented in this thesis are also able to extract a front of Pareto-optimal solutions for multiple objectives and are not limited to return just one solution optimized for either execution time or energy consumption.

Qiu: Qui et al. presented an energy-aware parallelization approach for embedded DSP architectures in [QNY+10]. Their Energy-Aware Loop Parallelism Maximization (EALPM) approach has some similarities to the multi-objective aware parallelization approaches presented in Chapter 6 and 8. All approaches try to reduce the system's energy consumption while extracting parallelism. Qiu's approach employs a two phase strategy. First, their approach extracts task- and data-level parallelism before the energy consumption is reduced by Dynamic Voltage Scaling (DVS). This two phase strategy may lead to suboptimal results since the DVS technique relies on the task structure, extracted in the first step. In contrast, the approaches presented in this thesis extract task-, data- and pipeline-parallelism for multiple objectives at the same time by using Genetic Algorithms. In addition, the approaches presented in this thesis also try to reduce the amount of extracted parallelism and also of the allocated processing units to reduce the overall energy consumption. Finally, the Intel Core 2 Quad processor used for evaluation purposes in [QNY+10] is not an embedded MPSoC so that the applicability to embedded devices must still be shown.

Wang: The approach presented by Wang et al. in [WLL+11] is perhaps most relevant to the multi-objective aware parallelization approaches presented in Chapters 6 and 8. However, Wang also employs a two phase strategy like Qiu [QNY+10] which may lead to suboptimal results. In the first phase, a so called RDAG algorithm is employed to optimize coarse-grained pipeline parallelism by re-timing techniques [LS91]. Afterwards, a Genetic Algorithm-based scheduling approach, namely GeneS, is applied to optimize the system's energy consumption by using Dynamic Voltage Scaling and Dynamic Power Management techniques. The multi-objective aware approaches presented in this thesis also employ Genetic Algorithms. But compared to the work proposed by Wang, the approaches presented in this work extract parallelism in a multi-objective aware manner. Wang only optimizes pipeline parallelism by re-timing techniques in a first step followed by a reduction of the energy consumption afterwards in a single objective fashion. Moreover, Wang only tries to map extracted tasks to available processing units in combination with Dynamic Voltage Scaling (DVS). The approaches presented in Chapters 6 and 8 combine the extraction of parallelism with mapping and iteration scheduling techniques in one algorithmic step while considering multiple objectives at the same time. Other approaches comparable to Wang were, e.g., presented in [LSW+08] and [ZHC02].

Cho: Cho et al. presented an analytical model similar to Amdahl's Law [Amd67] to evaluate the interplay of parallelization, program performance and energy consumption for multi-core architectures in [CM10]. The models assist to determine the maximum reduction of execution time and energy consumption (both, dynamic and static energy) by Dynamic Voltage and Frequency Scaling (DVFS) techniques and the capability to turn off processors completely. However, the paper just presents a simplistic model for evaluation purposes which neglects important parts, like, e.g., inter-processor communication costs. The publication in [CM10] further just presents analytical models and relies on "perfectly parallelized" applications.

2.5 Extraction for Heterogeneous Architectures

Heterogeneity has proven to be the most promising alternative to reduce the energy consumption and costs of homogeneous MPSoCs. By combining processing units with different performance characteristics on one device, applications can be accelerated by parallel execution with reduced energy consumption compared to homogeneous multi-core architectures. However, new problems arise if applications should be efficiently parallelized for heterogeneous architectures. While the automatic extraction of parallelism for homogeneous architectures is still a challenging research problem, the complexity further increases for heterogeneous ones. The extracted tasks have to be balanced automatically even though their performance characteristics vary on the available processing units. While heterogeneity is considered for a long time as a key aspect in mapping, scheduling, and design-exploration tools, like, e.g., [TBH+07], [KSS+09], [SGB10], and [JMB+12], only a few publications exist considering heterogeneity in the parallelism extraction domain. These approaches will be discussed and compared to the parallelization approaches for heterogeneous architectures presented in this thesis in Chapter 7 and Chapter 8.

MAPS: The MAPS framework was already discussed in the section presenting related task-level parallelization approaches. However, the work on MAPS was continued until today and in the most recent version presented in [SSO+13] the framework was extended to support heterogeneous architectures. The authors have extended the C programming language to C Process Networks (CPN) which integrate KPN annotations in C. With the help of these annotations, the user has to specify parts of the applications as concurrently executed processes manually. In the current version, no parallelization tools are integrated which extract parallelism automatically for heterogeneous systems. But at least a good basis for new tools is now available. In contrast, the approaches presented in Section 7.1, Section 7.2, Section 8.2.1 and Section 8.2.2 extract parallelism fully automatically from sequentially written applications for heterogeneous MPSoCs.

HELIX: The Helical Execution of Loop Iterations across cores (HELIX) parallelization framework was presented by Campanoni et al. in, e.g., [CJH+12b] and [CJH+12a]. HELIX concentrates on the extraction of simple data-level parallelism to be able to predict and automatically balance the parallel execution behavior by an extension of Amdahl's Law [Amd67]. The authors also claim in [CJH+12b] that their approach can be used to parallelize applications for heterogeneous architectures. However, the supported heterogeneity consists of a fast core which can only be used to execute the sequential parts of the application and a homogeneous block of equal, slower cores for parallel execution. In contrast, the heterogeneous parallelization approaches presented in this thesis also support architectures with arbitrary different cores and all of them can be used for parallel execution. Moreover, the approaches of this thesis focus on embedded MPSoCs and do not require a high-performance Intel Core i7 architecture for evaluation purposes as used in [CJH+12b].

AHP: The Automatic Heterogeneous Pipelining Framework (AHP) was presented by Pienaar et al. in [PCR12] and is based on the previous work in [PRC11]. AHP is able to exploit pipeline parallelism from annotated C++ code and maps it to heterogeneous architectures with different processing units. A Parallel Operator Directed Acyclic Graph (PO-DAG) annotated with profiled execution times of all tasks on various processing units is employed to express pipeline parallelism. Even though AHP's algorithm optimizes and maps various pipeline stages by heuristically merging different nodes to stages, the user has to annotate and thereby extract the different pipeline stages manually. This distinguishes it from the heterogeneous pipeline parallelization approaches presented in this thesis. The approaches presented here extract this kind of parallelism fully automatically for single and also multiple objectives at the same time. In addition, task-level, (simple) data level and pipeline parallelism can be extracted at the same time. Similar approaches to the one presented by Pienaar depending on manually extracted parallelism were published in, e.g., [LCW+08], [LHK09], and [ATN+11].

2.6 Additional Approaches

The approaches and frameworks discussed so far should only provide an overview about the most relevant publications in the wide area of thread-level parallelization frameworks relevant for the approaches presented in this thesis. A complete list of all approaches goes beyond the scope of this thesis. Among others, Par4All [Par13], Open64 [CGC+08], and Intel Parallel Studio [Int13] are also able to semi or fully automatically parallelize sequentially written applications to mention only some of them. Nevertheless, the following section briefly mentions related topics, which go beyond the scope of thread level parallelization approaches.

Instruction Level Parallelism: The research discipline of automatically extracting Instruction Level Parallelism started decades ago and is much older than the research area of Thread-Level Parallelism (TLP). While TLP executes large blocks of the application in parallel on various processing units, Instruction Level Parallelism was employed to execute single instructions in parallel on, e.g., Very Long Instruction Word (VLIW) machines or superscalar processors. As a consequence, Instruction Level Parallelism approaches are more fine-grained than TLP approaches so that different techniques have to be used for both research problems. Early approaches of Instruction Level Parallelism extraction were published in, e.g., [Fis81], [CNO+88], and [HMC+93] and are, in general, orthogonal to TLP approaches.

Speculative Parallelization: Speculative parallelization executes parts of the application speculatively in parallel and is also interesting in the context of TLP. However, it was not considered in this thesis since speculative execution can often not be applied to embedded devices. Timing predictability is crucial for many embedded systems. Unfortunately, it is hard to guarantee timing constraints for applications applying speculative parallelization. Representative approaches in this area were presented in, e.g., [BF02], [JEV04], and [ZS02].

Parallelization Implementation: Many parallelization approaches presented so far concentrate on the extraction of parallelism. The implementation is afterwards done by a tool specialized for implementation issues of parallel applications. This separation was also employed in the approaches presented in this thesis. Recent parallelization implementation tools or parallel languages were presented in, e.g., [BBW+09], [DM98], and [SGS10].

Mapping Applications to MPSoCs: The extraction and implementation of parallelism are only the first steps which have to be applied if applications should be efficiently ported to Multi-Processor System on Chip (MPSoC) devices. Afterwards, among others, scheduling, mapping, memory optimizations and also design-space exploration should be applied. Many research projects faced these research topics in recent years. Some of them can be found in, e.g., [BPS+10], [CDF+11], [TBH+07], [KSS+09], [SGB10], [NTS+08], and [JMB+12].

2.7 Summary

As shown in this chapter, a lot of research had been done in the last decades to develop approaches which are able to extract thread-level parallelism from sequentially written applications in an automated way. However, limitations could be observed for most of them: To summarize, the majority of the previously published approaches ...

- ... are designed for high-performance architectures and are hence not well applicable for resource restricted embedded devices. Some of the presented approaches even require special communication structures since the throughput of their high-performance unified memory architecture (UMA) was not high enough. Instead, the approaches should cautiously trade-off parallel execution versus task-creation and communication overhead. This is even more important for embedded devices since communication is, in general, much more expensive for these systems.

- ... extract as much parallelism as possible without validating whether parallel execution really leads to the desired speedup. The usage of high-level cost models for these purposes could be a promising solution.
- ... are evaluated on high-performance architectures even if they demonstrate parallelization approaches for embedded architectures. Instead, those approaches should be evaluated on at least a simulated embedded platform.
- ... extract only one kind of parallelism (e.g., data-level parallelism) without combining the advantages of several parallelization types (like, e.g., task-level and pipeline parallelism).
- ... focus on the optimization of execution time as their only optimization objective at the expense of other resources, like, e.g., energy consumption or communication overhead. For resource restricted embedded devices, it makes more sense to reduce the amount of extracted parallelism and move to an architecture with fewer cores to save energy if a given speedup is sufficient.
- ... are optimized for homogeneous architectures even though the pervasiveness of heterogeneous MPSoCs significantly increased in the last years. These approaches do not distinguish between different performance characteristics of the available processing units. This is indispensable since tasks should be balanced automatically to utilize heterogeneous architectures efficiently.

As already exposed in Chapter 1, the parallelization approaches presented in this thesis try to fill the existing desideratum by considering the aforementioned points to form a parallelization framework which is tailored towards the special requirements of embedded systems.

Chapter 3

Framework

Contents

Motivated by the observations which have been made while examining the existing state-of-the-art parallelization approaches discussed in the related work section, a new parallelization framework was developed in the context of this thesis. This framework contains and combines several parallelization approaches, presented later in this thesis, which are tailored towards special requirements imposed by resource-restricted embedded systems. The demands claimed by these newly developed parallelization approaches, comprise, e.g., execution time estimation, a new hierarchical divide-and-conquer-based parallelization approach, and specialized intermediate representations. These demands required the development of a completely new parallelization framework. Before the different developed parallelization techniques and the global parallelization approach are presented in detail in Chapters 4 - 8, this chapter describes the main components of the new parallelization framework and the framework's integration into two research tool flows in Section 3.1. This section also presents the tools employed to evaluate the presented approaches in the remainder of this thesis.

The internal structure of the developed parallelization framework with all its components is presented in Section 3.2. This section also gives a brief overview of the employed dependence extraction techniques, as well as objective estimations performed, like, e.g., execution time estimation. These parts are fundamental for the high-level models integrated into the approaches presented in the remainder of this thesis. Finally, Section 3.3 describes the embedded target platforms which are used for evaluation purposes for the novel parallelization approaches.

3.1 Integrated Parallelization Tool Flows

The parallelization framework presented in this thesis was initially developed in the context of the MNEMEE (Memory Management Technology for Adaptive and Efficient Design of Embedded Systems) European Union FP7 project [BPS+10]. As a result, it was part of MNEMEE's optimization tool flow and was responsible to extract tasks from sequentially written embedded applications. Later, the developed parallelization framework was also integrated into a second internal project called PA4RES (Parallelization for Resource Restricted Embedded Systems). The integration of the developed parallelization framework into both research tool flows is presented in the following Sections 3.1.1 and 3.1.2, respectively.

3.1.1 MNEMEE Tool Flow

The first two approaches presented in this thesis (cf. Chapter 5) as well as the overall parallelization approach with its divide-and-conquer-based parallelization technique (cf. Chapter 4) were developed in the context of the MNEMEE European Union FP7 project [BPS+10]. The focus of the MNEMEE project was the development of scientific approaches which should be able to map and optimize sequentially written C applications to embedded multi-core platforms. Therefore, parallelization approaches, static and dynamic data allocation techniques as well as mapping approaches were developed and finally implemented in several optimization tools. All tools can be executed in isolation or also in a combined tool flow in a transparent way. Each tool is designed to perform source-to-source transformations so that the application designer can easily observe the results of each optimization step. Most tools, including the parallelization framework presented in this thesis, are based on an intermediate representation called ICD-C IR [Inf13] which facilitates the design and development of source code analysis and optimization techniques while staying as close as possible to the original source code representation.

The resulting tool flow of the MNEMEE project is depicted in Figure 3.1. As can be seen, the parallelization approaches developed in the context of this thesis (Parallelizer) are executed in the second position of MNEMEE's tool flow. They are started as soon as the Dynamic Data Type Refinement tool (DDTR) [BRMA+09] has optimized and re-allocated dynamic data structures to, e.g., scratchpad memories. The parallelization approaches, developed in the context of this thesis, also perform source-to-source transformations (more information is given in Section 3.2). They take the sequential application code optimized by DDTR as input, extract parallelism by one or also by a combination of the different developed parallelization techniques, and annotate the final results to the source code of the application. These annotations are compliant to the input specifications of the MPSoC Paral-

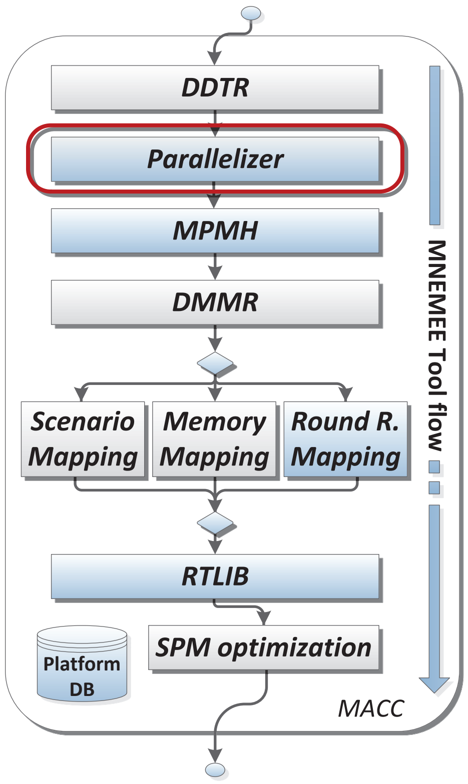


Figure 3.1: Tool Flow Developed in the Context of the MNEMEE European FP7 Project

lelization and Memory Hierarchy tool (MPMH) [IMM+10]. MPMH is used in the following step to implement the extracted parallelism exploited by the approaches presented in this thesis. Among other optimizations, the MPMH tool also optimizes large data structures by splitting them into smaller parts so that they can be placed in smaller and more efficient memories.

The parallelized application is now further optimized by the Dynamic Memory Management Methodology tool (DDMR) before the extracted tasks are mapped to the available processing units of the targeted MPSoC. Therefore, multiple mapping tools were developed in the context of this project. The first two ones are either scenario based [SGB10] or memory aware [JMB+12] and combine the mapping of tasks with additional optimization techniques. The third mapping tool is a trivial one which just maps the tasks in a round-robin fashion to the available processing units. This tool was initially developed for debugging purposes but was also used in the context of this thesis to be able to create a mapping without additional optimizations. The mapped application is further linked against a runtime library (RTLIB) implementing, among others, task creation and communication directives. Finally, a scratchpad memory optimization tool (SPM optimization) allocates static data objects to scratchpads or other efficient memories in the memory hierarchy of the targeted embedded MPSoC.

All tools are based on the MACC framework [PKM+10] which is used to fa-

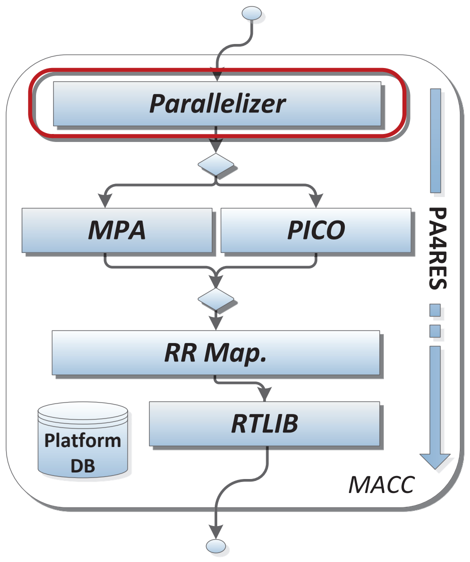


Figure 3.2: Tool Flow Developed in the Context of the Internal PA4RES Project

cilitate communication between all optimization steps provided by the MNEMEE tool flow. In addition, the MACC framework models the target architecture so that platform-dependent information, like, e.g., the amount and clock frequencies of the available processing units can be inquired by the different optimization approaches.

To be able to evaluate solutions generated by the parallelization approaches presented in Chapters 4 - 8, some of the tools of the MNEMEE tool flow were used. Those tools are highlighted in Figure 3.1 by blue shapes. More details on the interconnection of these tools for the evaluation tool flow used in this thesis are later given in Section 3.2. Results obtained by the whole MNEMEE tool flow, as well as more details on the specific optimization techniques and their integration into the combined tool flow, are presented in [BPS+10].

3.1.2 PA4RES Tool Flow

The PA4RES tool flow (cf. Figure 3.2) is the second one which employs the novel parallelization approaches developed in this thesis to extract parallelism from sequentially written embedded applications. The tool flow also uses some of the other tools which were developed in the context of the MNEMEE European FP7 project. Accordingly, the developed parallelization approaches (Parallelizer), MPA (the parallelization part of MPMH), the round robin mapping tool as well as the RTLIB are used in the tool flow of the PA4RES project as well. Besides MPA, a second parallelization implementation tool, namely, PICO (Parallelization Implementation and Communication Optimization), is available in the PA4RES tool flow.

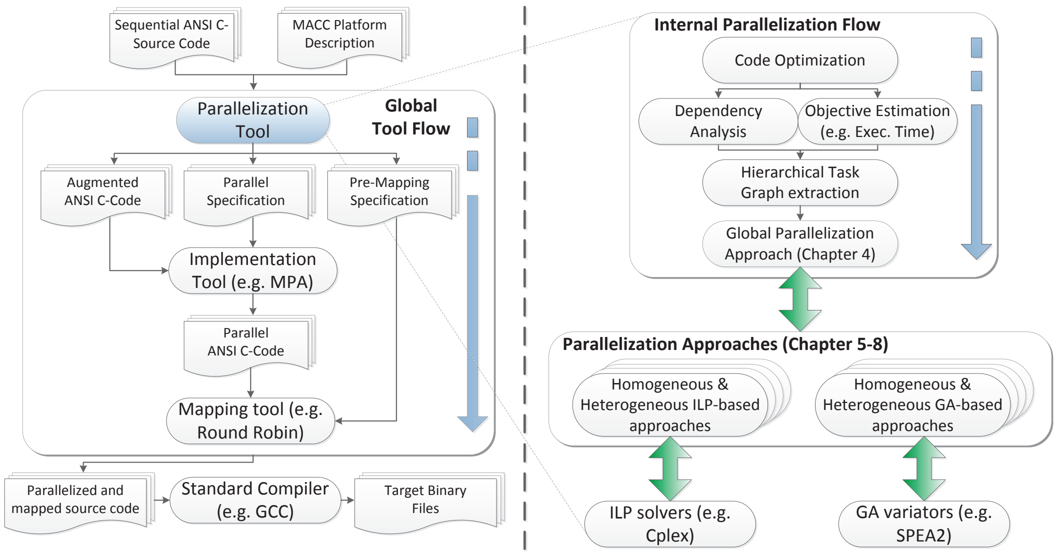
To support the use of both parallelization implementation tools, the parallelization framework of this thesis was extended to be able to annotate the extracted solutions for the PICO tool as well. Fortunately, both input specifications of MPA and PICO do not exclude each other. While MPA expects sequential source code annotated with label statements mapped to tasks by a separate parallel specification file, PICO expects C statements annotated with pragmas. Therefore, the parallelization framework developed in this thesis annotates both, label statements as well as pragmas to the applications' source code to describe the extracted parallel solutions. Thus, both tools can parse and optimize the same output generated by the novel parallelization extraction approaches.

Up to now, results could not be obtained by the PA4RES tool flow using the PICO tool for parallelization implementation since the tool is still under development at the time this thesis was finalized. In the future, it is planned to tightly couple the parallelization approaches developed in this thesis with the ones of the PICO implementation tool to further optimize the quality of the parallelized applications.

3.2 Parallelization Framework

So far, the integration of the developed parallelization framework into two research projects was presented. This section will further describe the internal structure of the developed parallelization framework and the employed subset of MNEMEE tools used for evaluation purposes in the remainder of this thesis. Both parts are visualized in Figure 3.3.

The global perspective of the employed evaluation tool flow is visualized in Figure 3.3(a). As shown, the tool flow expects sequential ANSI C code together with a platform description (based on the MACC framework) of the targeted heterogeneous architecture as input. Compared to many other parallelization tool flows, the one presented here directly operates on sequentially written ANSI C source code. Thus, many embedded applications can be parallelized without manual transformations into other programming languages or Models of Computation. The parallelization framework developed in the context of this thesis automatically parallelizes the given application with the approaches presented in Chapters 5 - 8 while considering architectural properties of the given embedded target platform. As a result, the parallelization tool annotates the source code of the application to describe the extracted parallelism. Figure 3.3(a) shows the tool flow which is used if the MPA tool is employed to implement the extracted parallelism. Here, a parallel specification that maps labeled statements of the application to tasks, is also created by the parallelization framework. With both inputs, the MPA tool automatically implements the extracted parallelism which is further processed by a mapping tool. The presented parallelization framework of this thesis optimizes the extracted tasks so that they are automatically balanced even for processing units of heterogeneous architectures with different performance characteristics. Therefore, depending on the given target architecture, a pre-mapping specification can be generated which is passed to the mapping tool. This specification contains information about the extracted task-to-processor class mapping (only for heterogeneous target architectures) to ensure


Sequential ANSI C-Source Code
MACC Platform Description
Parallelization Tool
Global Tool Flow
Augmented ANSI C-Code
Parallel Specification
Pre-Mapping Specification
Implementation Tool (e.g. MPA)
Parallel ANSI C-Code
Mapping tool (e.g. Round Robin)
Parallelized and mapped source code
Standard Compiler (e.g. GCC)
Target Binary Files
Internal Parallelization Flow
Code Optimization
Dependency Analysis
Objective Estimation (e.g. Exec. Time)
Hierarchical Task Graph extraction
Global Parallelization Approach (Chapter 4)
Parallelization Approaches (Chapter 5-8)
Homogeneous & Heterogeneous ILP-based approaches
Homogeneous & Heterogeneous GA-based approaches
ILP solvers (e.g. Cplex)
GA variators (e.g. SPEA2)

(a) Global Tool Flow (b) Internal Structure

Figure 3.3: Global Perspective and Internal Tool Flow of the Parallelization Framework

that tasks are mapped to processing units for which they are optimized.

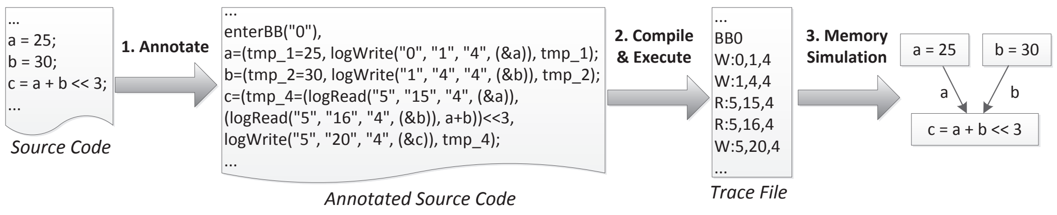
All tools described so far perform source-to-source transformations. This has the advantage that the designer can observe the applied code modifications after each step. In addition, a standard compiler can be used to compile the parallelized source code into binary files, which are linked against a library that implements task creation and synchronization primitives (RTLIB). The developed tool flow also contains links to the cycle-accurate Vast [Syn13b] and MPARM [BBB+05] simulators, so that the sequentially written applications can fully automatically be parallelized, mapped and evaluated on several architectures without manual intervention.

The internal tool flow of the parallelization framework developed in the context of this thesis is shown in Figure 3.3(b). All tools shown in this figure can be executed in a combined fashion or as stand-alone tools, which enables an easy exchange of the tools. A code optimization tool is executed first to enable an easier code analysis for the succeeding parallelization steps (cf. Section 3.2.2). The optimized code is then analyzed to extract data and control flow dependencies (cf. Section 3.2.2) as well as objective values, like, e.g., execution time and energy consumption required by the statements of the application (cf. Section 3.2.3). Finally, the global parallelization approach extracts the Augmented Hierarchical Task Graph (AHTG) as described in Chapter 4 before it starts to extract parallelism from the AHTG.

The following subsections describe the developed tools in more detail.

3.2.1 Code Optimization

The code optimization tool performs simple code transformations, like, e.g., constant propagation, constant folding, dead code elimination, and other standard compiler



```
...
a = 25;
b = 30;
c = a + b << 3;
...
```

```
...
enterBB("0"),
a=(tmp_1=25, logWrite("0", "1", "4", (&a)), tmp_1);
b=(tmp_2=30, logWrite("1", "4", "4", (&b)), tmp_2);
c=(tmp_4=(logRead("5", "15", "4", (&a)),
(logRead("5", "16", "4", (&b)), a+b))<<3,
logWrite("5", "20", "4", (&c)), tmp_4);
...
```

```
...
BB0
W:0,1,4
W:1,4,4
R:5,15,4
R:5,16,4
W:5,20,4
...
```

Figure 3.4: Structure of Dependency Analyzer

optimizations described in [Muc97]. These optimizations enable an easier code analysis for the different parallelization approaches executed later in the internal tool flow. The code optimizations are provided by the ICD-C framework [Inf13]. Besides these standard optimizations, the code optimization tool also analyses the loops of the application to be parallelized and unrolls a few iterations. This helps the task-level parallelization approach to extract parallelism from loops of the application to be parallelized. This unrolling transformation is only performed temporarily and finally reverted before the parallelized application code is written out.

3.2.2 Dependency Analyzer

The dependency analyzer developed in the context of this thesis expects the intermediate representation of the optimized source code as input to extract all data dependencies of the application to be parallelized. This information is required to build the Augmented Hierarchical Task Graph. Therefore, a profiling-based approach was developed here[1]. Of course, parallelization hints might ignore dependencies which are not manifested in the profiling run, due to profiling driven analysis. This does not harm the correctness of the developed parallelization tool flow since MPA and PICO are employed to implement the extracted parallel solutions. Both tools are based on safe static analysis techniques and introduce synchronization and communication directives if necessary. Hence, an extracted solution may be less performant than expected but is always valid. Nevertheless, for none of the evaluated benchmarks, it could be observed that the parallelization approaches extracted parallelism where non-detected data dependencies prevented parallel execution.

The profiling-based approach also has some advantages compared to a static analysis. For example, it delivers extremely detailed information about access patterns and very fine-grained dependency information. By using the profiling-based approach, it is possible to identify, for example, that loop iterations modifying arrays or pointers are independent and therefore possible parallelization candidates. Usually, this is particularly hard to detect via static analysis techniques. These observations are also reported in [TF09], where another profiling driven parallelization approach was presented.

[1]The author of this thesis focused on the development of new parallelization techniques for embedded systems. Since a dependency analysis was indispensable but not available, this solution seemed to be a good trade-off between effort and applicability. However, the dependency analyzer is implemented by a separate tool and can be exchanged as soon as a static analyzer is available.

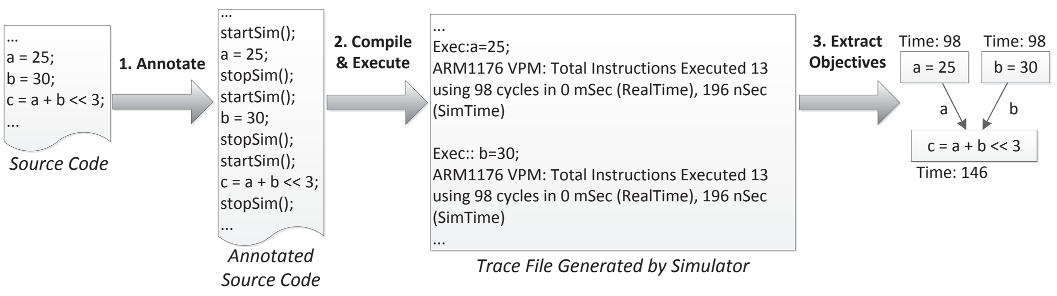


Figure 3.5: Structure of Objective Estimator

The structure of the developed profile-driven data dependency analysis is visualized in Figure 3.4. In a first step, function calls are added to the given application's source code tracking read and write operations to variables (i.e., memory locations). Also the entrances to basic blocks are annotated to register dependencies crossing function boundaries. All inserted function calls write an unique statement id, an unique expression id, and the size of the read or written memory locations to a trace file. The annotated source code is compiled in the next step and linked against a library implementing the inserted function calls. The compiled and annotated application is executed in the following by the analysis tool to generate trace files covering all memory accesses of the application to be analyzed. Since those trace files often get extremely large, the developed approach communicates with the annotated application's executable via Unix Sockets. As a result, only a manageable small trace file is generated (e.g., 200MB), first. Afterwards, the annotated application passes the filename to the analyzer so that a simulation of accessed memory locations is started. In the meanwhile, a second trace file is generated by the annotated application. Since the memory simulation of the dependency analyzer takes significantly longer than the trace file generation, the latter one is paused until the analyzer starts to simulate the next trace file. By using this handshake model, trace files of enormous sizes are avoided so that the approach can – in theory – analyze large applications. As a result of the analysis, dependencies between different statements as well as their expressions are annotated to the IR so that they can be used for the creation of the Augmented Hierarchical Task Graph in the following.

3.2.3 Objective Estimation

The parallelization approaches presented later in this thesis employ high-level cost models to evaluate the benefit of a parallelized part of the application. Therefore, cost information for the different objective values, like, e.g., execution time and energy consumption, are extracted for each statement of the application.

The structure of the developed objective estimator is visualized in Figure 3.5. Each statement of the application is copied to a new source code file and `startSim` and `stopSim` function calls are added surrounding each statement. The augmented source code file is then processed by a cross-compiler generating target code for the considered architecture. This application is linked with a library implementing the inserted function calls for the employed simulator to start and stop the measurement before and after each statement, respectively. The generated output of the simulator contains results for the considered objectives (only the execution time is visualized in Figure 3.5 to ease comprehensiveness) which are parsed by the objective analyzer. These values are then annotated to the application's IR so that they can be attached to the AHTG in the following. The tool supports the cycle-accurate Vast [Syn13b] and MPARM [BBB+05] simulator. As a result, objective values for all considered target platforms (cf. Section 3.3) can be evaluated.

3.2.4 Parallelization Approaches

The global parallelization approach with its divide-and-conquer-based parallelization technique (Chapter 4) is started as soon as the Augmented Hierarchical Task Graph (AHTG) is extracted based on the gathered information. The global approach iterates through the hierarchy of the AHTG and tries to extract parallelism for the nodes of the graph in isolation (cf. Chapter 4). For these, one or also a selected set of parallelization techniques presented in Chapters 5 - 8 is executed. Most of these approaches create Integer Linear Programming (ILP) or Genetic Algorithm (GA)-based problem descriptions which are solved by different external tools. To solve, e.g., created ILP systems, IBM's CPLEX [IBM13] or also the freely available lp_solve [BKP13] can be used. For Genetic Algorithms, the PISA framework [BLT+03] is connected with the different approaches containing several variators, like, e.g., SPEA2 [ZLT01] (cf. Figure 3.3(b)). As soon as the entire application is processed, the extracted tasks are annotated to the application's source code and a parallel specification file is created describing the structure of the created parallel sections.

3.3 Target Platforms

Three different target platforms are employed in the context of this thesis to evaluate efficiency and portability of the developed approaches. They are presented in the remainder of this chapter. The author of this thesis could benefit of earlier developed platforms and did not have to contribute in their development.

3.3.1 MPARM Platform

MPARM [BBB+05] is the cycle-accurate simulator for the first supported target architecture. It was also used in the MNEMEE European Union FP7 project [BPS+10] to evaluate the efficiency of the presented tool flow. The structure of the simulated target architecture is visualized in Figure 3.6. Up to four ARM7M single-core processors [ARM13c] can be added to the simulated target architecture. All cores are enriched with small scratchpad memories as well as data and instruction caches connected via a fast local bus. Each processor operates on its own private memory

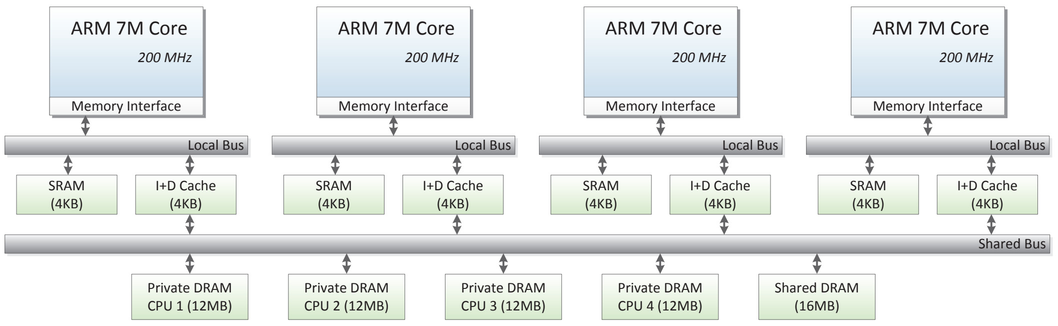

ARM 7M Core
200 MHz
Memory Interface
Local Bus
SRAM (4KB)
I+D Cache (4KB)
Shared Bus
Private DRAM CPU 1 (12MB)
Private DRAM CPU 2 (12MB)
Private DRAM CPU 3 (12MB)
Private DRAM CPU 4 (12MB)
Shared DRAM (16MB)

Figure 3.6: MPARM Platform Configured with four Processing Units

with 12 MB size each which is available through a shared bus. The architecture also contains a 16 MB DRAM which can be accessed by all processors. This memory is especially used for communication between the available cores.

Communication and synchronization are expensive on this architecture since the available processors are only accelerated by level one caches without cache coherency mechanisms. Thus, only the content of the local DRAMs can be loaded into the cache. The shared DRAM has to be excluded since the level one caches cannot recognize if another core has written to the shared memory. Therefore, each access to the shared memory causes read or write operations on the slow DRAM memory. Bus contention can further slow down these accesses. Hence, the granularity of the extracted tasks has to be considered carefully for this target platform.

However, MPARM has a big advantage compared to the other considered target architectures. It is enriched with an energy model called MEMSIM [Kat08] which is based on [WM06]. This energy model is directly integrated into the simulator and is therefore used to evaluate the multi-objective aware parallelization approaches in the remainder of this thesis (cf. Chapters 6 and 8). Both, energy values and access delays for the available memories are set to the default values of MPARM [BBB+05] and MEMSIM [Kat08].

3.3.2 ARM11QuadProc Platform

The ARM11QuadProc platform (cf. Figure 3.7) is the second one used to evaluate the different parallelization approaches developed in the context of this thesis. It is simulated by Synopsis's cycle-accurate CoMET simulator [Syn13a] which is part of the Vast framework. Compared to the MPARM platform, it contains up to four ARM1176 single-core processors [ARM13a] of the ARM11 processor family. The ARM1176 processors are more recent than the earlier employed ARM7M processors and contain separate Tightly Coupled Memories (TCM)[2] for data and instructions directly integrated on the processors' die. Moreover, the level one caches for data and instructions are also directly integrated on the processors' die.

Also here, no cache coherency unit could be added to be able to cache the shared

[2]TCM memories and scratchpad memories are the same.

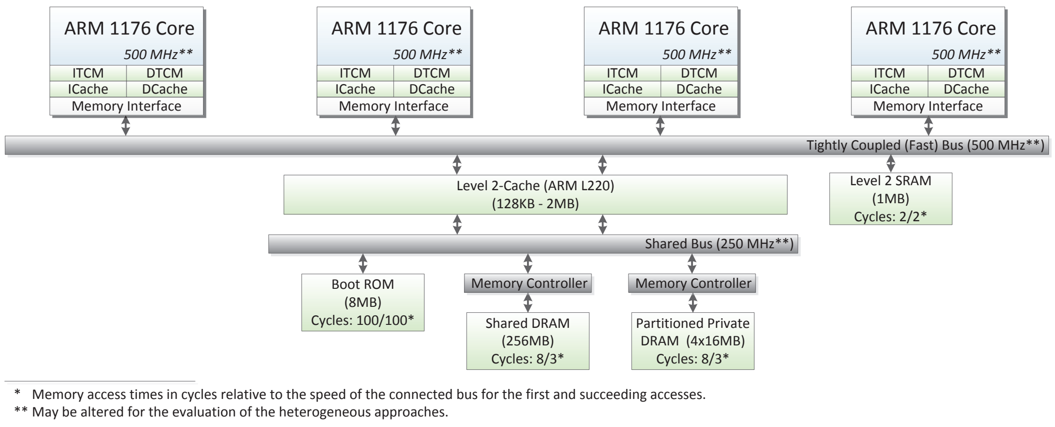

ARM 1176 Core
500 MHz**
ITCM
DTCM
ICache
DCache
Memory Interface
Tightly Coupled (Fast) Bus (500 MHz**)
Level 2-Cache (ARM L220) (128KB - 2MB)
Level 2 SRAM (1MB) Cycles: 2/2*
Shared Bus (250 MHz**)
Boot ROM (8MB) Cycles: 100/100*
Memory Controller
Shared DRAM (256MB) Cycles: 8/3*
Partitioned Private DRAM (4x16MB) Cycles: 8/3*
* Memory access times in cycles relative to the speed of the connected bus for the first and succeeding accesses.
** May be altered for the evaluation of the heterogeneous approaches.

Figure 3.7: ARM11QuadProc Platform Configured with four Processing Units

memory in the level one caches. Instead, a level two cache based on ARM's L220 cache controller is connected via a fast, tightly coupled bus. Since all processors are connected to this level two cache, it can cache all memories behind it, including both private and also shared memories. In contrast to the MPARM platform, the ARM11QuadProc architecture contains only one private memory which is partitioned into parts of equal size for all processors. Both, the private, as well as the shared memories, are connected to the bus by a DRAM controller. Two additional smaller memories are added to this platform, namely a boot ROM memory and a level two SRAM. The boot ROM contains the application code which should be executed by the architecture while the level two SRAM is used to accelerate the employed operating system.

This platform does not provide an energy model so that it cannot be used to evaluate the multi-objective aware approaches presented later in this thesis. However, it is possible to clock the processors with different core frequencies so that the platform can be used to evaluate both, homogeneous and heterogeneous single-objective aware approaches (cf. Chapters 5 and 7).

3.3.3 Arm11MPCore Platform

The previously presented target platforms are multi-processor architectures since they combine several single-core processors. In contrast, the ARM11MPCore platform (cf. Figure 3.8) contains one ARM11MPCore multi-core processor [ARM13b] providing up to four cores in one processor. Thereby, the processors are tighter coupled which often decreases synchronization and communication costs. The ARM11-MPCore platform is also simulated by the cycle-accurate CoMET simulator [Syn13a] of the Vast framework.

Multi-core processors are usually simpler in their structure. The ARM11MPCore processor does, e.g., not contain fast TCM memories. But in contrast to the single-core processors described above, the ARM11MPCore processor employs a cache snoop unit which takes care that the caches of the different cores are automatically

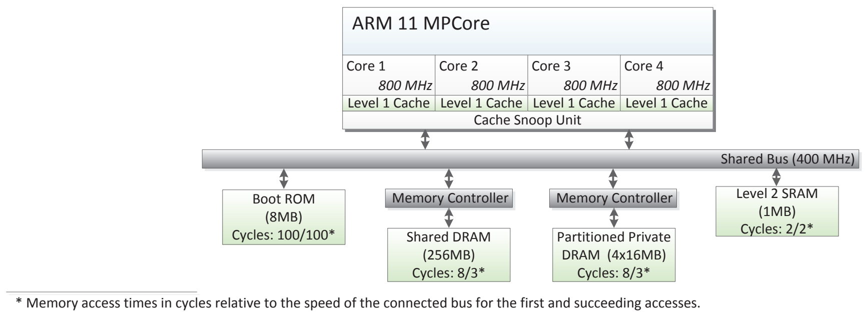

ARM 11 MPCore
Core 1
800 MHz
Level 1 Cache
Core 2
800 MHz
Level 1 Cache
Core 3
800 MHz
Level 1 Cache
Core 4
800 MHz
Level 1 Cache
Cache Snoop Unit
Shared Bus (400 MHz)
Boot ROM (8MB) Cycles: 100/100*
Memory Controller
Shared DRAM (256MB) Cycles: 8/3*
Memory Controller
Partitioned Private DRAM (4x16MB) Cycles: 8/3*
Level 2 SRAM (1MB) Cycles: 2/2*
* Memory access times in cycles relative to the speed of the connected bus for the first and succeeding accesses.

Figure 3.8: ARM11MPCore Platform Configured with four Processing Units

updated if one of the cores writes to one of the available memories. Therefore, also data which is allocated to the shared memory can be cached here, which significantly improves the performance of accesses to such memory locations.

This platform neither provides an energy model nor is it possible to clock the cores with different CPU frequencies. This makes this platform not applicable to evaluate the multi-objective aware parallelization approaches, as well as the heterogeneous ones. Therefore, it is used to evaluate the single-objective aware homogeneous parallelization approaches presented in Chapter 5 only. However, for such approaches the ARM11MPCore platform is representative since the used processor can often be found in embedded multi-core architectures.

3.4 Summary

This chapter presented technical details about the parallelization framework developed in the context of this thesis. With its designed structure, the framework builds a basis to easily integrate the different parallelization approaches presented in the remainder of this thesis into one parallelization framework. Therefore, a basis for a parallelization framework which is optimized for embedded systems is created. This chapter further presented the integration of the parallelization framework into two research tool flows of the MNEMEE and PA4RES projects.

In addition, three target architectures were presented in this chapter which are used to evaluate the approaches presented later in this thesis. They employ three different processors (ARM7M, ARM1176 and ARM11MPCore) in multi-processor and also multi-core architectures. Some of them employ scratchpad memories as well as cache hierarchies or cache snoop units to accelerate memory accesses for parallelized applications. These performance variances have to be considered by the developed parallelization approaches to create solutions which are well tailored towards the considered target architectures.

CHAPTER 4

Parallelization Methodology

Contents

The major challenge that has to be addressed if sequentially written applications should be parallelized in an efficient way is the complexity of the vast solution space. Among others, the parallelization approaches, presented later in this thesis, have to map statements of the application to be parallelized to newly extracted tasks. Since all statements can be mapped to various tasks, an immense amount of solution candidates exists even for small applications. All possible mappings of statements to tasks represent solutions with different objective values regarding execution time, energy consumption, and other objectives. To make things worse, dependencies between statements require communication and synchronization directives if the statements are mapped to different tasks. This forms a series of predecessor and successor relationships between the extracted tasks causing additional delays, which are hard to foresee. However, this has to be considered in the evaluation process, as well. To cap it all, the extracted tasks optimized for one architecture may perform worse on another one since task creation, task execution, and communication costs may vary with respect to the employed processing units and the utilized memory subsystems.

Based on these observations, Sarkar has proven that finding an optimal partition of the statements of an application into concurrently executed tasks is an NP-complete problem [Sar89]. As already shown in the related work Chapter 2, this has led to the development of various inaccurate parallelization methodologies. Some of the previously published approaches disregard the evaluation of the found parallel solution candidates completely and just extract as much parallelism as possible (e.g., [VNS07]). Other approaches employ high-level cost models but rely on simplistic greedy heuristics merging, e.g., the two statements with the highest communication costs into the same task (e.g., [CCS+08]).

The approaches presented in Chapters 5 - 8 of this thesis are designed to provide solutions avoiding these limitations by employing sophisticated parallelization techniques. All presented techniques include high-level cost models to evaluate the benefit of a parallel solution candidate and to balance the extracted tasks automatically. A smart reduction of the solution space is indispensable but should not prevent the extraction of efficient parallelism. Therefore, the presented parallelization techniques of this thesis partition the application in a divide-and-conquer fashion into several hierarchical levels to reduce the number of statements processed at the same time. A suitable structure for the hierarchical segmentation is already given by the hierarchical structure of the application's source code. Often, efficient parallelism can be extracted from parts of the application which are close to each other with respect to the control flow. As a consequence, e.g., a loop's body can be divided into concurrently executed tasks or two function calls which are executed consecutively can be evaluated simultaneously by parallel execution. In contrast, there is, in general, no profit in extracting two tasks from the if- and else-part of a conditional statement since those tasks would never be executed concurrently. This information is implicitly given in the hierarchy of the application's source code and should therefore be used to reduce the search space of the parallelization problem.

Many previously published parallelization approaches employ a flat Program Dependence Graph (PDG) or an alteration of it as intermediate representation. Unfortunately, no hierarchical structures are present in this representation so that the entire program has to be processed simultaneously. This led to the described limitations of currently available parallelization heuristics. To circumvent these limitations, a hierarchical parallelization approach is presented in this thesis. It splits the complex parallelization problem into smaller subproblems, which can be solved efficiently. An intermediate representation called Augmented Hierarchical Task Graph (AHTG) is employed in this work which is capable of dividing the intermediate representation into different hierarchical levels. The AHTG combines control- and data-flow dependencies with annotated performance characteristics in one graph representation. An earlier version of an HTG was presented by Girkar and Polychronopoulos in [GP94] which was used for scheduling optimizations in [Pol91]. The approaches presented in this thesis adapt the HTG to an AHTG and employ it to extract parallelism from sequentially written applications. Therefore, the originally presented HTG is modified in a way that the different hierarchical levels can be processed in isolation. This leads to a significant reduction of the parallelization problem since only a few statements are processed in one step enabling more complex parallelization approaches. The intermediate representation is also designed in a way that each hierarchical sub-graph is acyclic, enabling the evaluation of high-level cost models. The extracted parallelism found on one hierarchical level is further combined with parallelism found on other hierarchical levels if it increases the overall performance. By applying this hierarchical approach, different granularities of extracted parallelism from various parallelization types can easily be combined to find well suited solutions for embedded MPSoCs.

The rest of this chapter is structured as follows: First, the Augmented Hierarchical Task Graph is described and defined in more detail in Section 4.1. The developed hierarchical parallelization approach used by all parallelization techniques of this thesis is presented from a global perspective in Section 4.2. This section also describes the integration of the different parallelization techniques presented in Chapters 5 - 8 into the global parallelization approach.

4.1 Augmented Hierarchical Task Graph (AHTG)

The Augmented Hierarchical Task Graph (AHTG) is used as central intermediate representation within the parallelization framework presented in this thesis. By partitioning the graph structure into different hierarchical levels in a divide-and-conquer fashion, the vast solution space of the parallelization problem can be drastically reduced. This enables complex parallelization approaches which have to search for parallelism only on a small portion of the application at the same time. Each hierarchical level should also be self-contained and acyclic so that it is possible to evaluate the impact of parallelism for each node in isolation.

4.1.1 Structure and Components of the AHTG

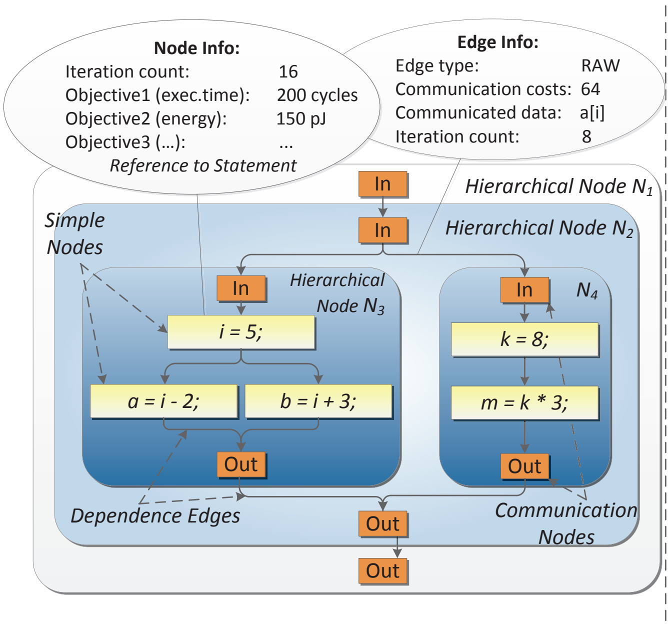
A simplistic example of an AHTG as shown in Figure 4.1 shall be used to introduce the components of the graph. The right-hand side of this figure shows the tree structure of the hierarchy (cf. Figure 4.1(b)) of the graph shown on the left-hand side (cf. Figure 4.1(a)). The example presented in Figure 4.1 is arranged in four hierarchical levels and contains several node types, edges between the nodes, and also cost information added to both, nodes and edges of the graph. These components, as well as the most important properties of the AHTG, are introduced in the following.

As depicted in Figure 4.1(a), the graph contains the following node types:

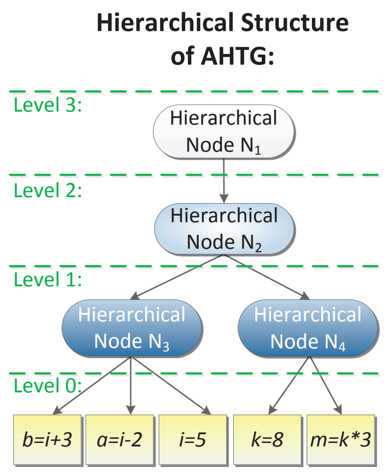
- **Simple nodes:**
  Simple nodes correspond to a basic statement in the original source code without any further hierarchical structures. Such a statement could be, e.g., an assignment (expression) statement like '$a = b$;'. By construction all leaves of the hierarchical structure of the graph (cf. Figure 4.1(b)) are represented by simple nodes.

- **Hierarchical nodes:**
  Hierarchical nodes correspond to statements providing a hierarchical structure, like, e.g., loops or function bodies in the original source code. All hierarchical nodes contain a communication in-node, a communication out-node and an arbitrary number of child nodes. These child nodes can be either simple nodes or additional hierarchical nodes.

- **Communication in-nodes:**
  Communication in-nodes are part of every hierarchical node. Communication from a node not contained in the hierarchical node to any inner node is



(a) Components of AHTG



(b) Hierarchical Structure of AHTG

Figure 4.1: Simplified Example of an Augmented Hierarchical Task Graph (AHTG) Structured in four Hierarchical Levels.

redirected through these communication in-nodes. Moreover, communication in-nodes are single-entry nodes and are hence a predecessor of all other nodes contained in the same hierarchical node.

- **Communication out-nodes:**
  Communication out-nodes are also part of every hierarchical node. Communication from a child node of the hierarchical node to any node not contained in the hierarchical node is redirected through this communication out-node. Moreover, the communication-out nodes are single-exit nodes of the hierarchical nodes and are successors of all other nodes contained in the hierarchical node, accordingly.

By introducing the new communication nodes as single-entry and single-exit points of a hierarchical node and by redirecting the communication between different hierarchical levels over the communication nodes (which was not done in this way by [GP94]), each hierarchical node is self-contained and can be processed in isolation. The communication nodes also redirect communication which points in the opposite direction to the regular control flow and create acyclic sub-graphs for each hierarchical node. Both properties are essential for the employed parallelization algorithms presented later in this thesis. More information on the way edges are redirected is given in the following subsection.

By construction, all nodes of the graph with the exception of the communication nodes directly correlate to statements of the application to be parallelized. Therefore, this relationship is also preserved to easily convey transformations done in the graph representation to the source code and vice versa. Another profitable characteristic of the presented Augmented Hierarchical Task Graph is that the control flow is already expressed by the hierarchical structure of the graph. Control-flow back edges for, e.g., loops are implicitly modeled by the hierarchy of the nodes and must not explicitly be added. Only explicit jump statements like `break`, `return`, or `goto` (for imperative programming languages like C) require additional control flow edges. Besides control flow edges, the AHTG contains Read-after-Write (RAW), Write-after-Read (WAR) and Write-after-Write (WAW) data-dependencies. With this distinction of dependence types, the parallelization algorithms are flexible since not all dependence types automatically enforce a sequentialized execution of the extracted tasks in all situations.

However, information extracted from the graph structure containing nodes and edges itself is insufficient to generate well-balanced tasks from sequentially written applications. The high-level cost models employed in the parallelization approaches of this thesis require additional estimated information about, e.g., execution costs and communication overhead, which has to be inserted if two statements are executed in different tasks. Therefore, the nodes of the graph are augmented with additional cost information (cf. *Node Info* in Figure 4.1(a)):

- **Iteration count:**
  The iteration count of the node is equal to the iteration count of the statement represented by the node and is extracted by the framework presented in Section 3.

- **Objective values:**
  The objective values contain estimated cost information for the optimization objectives considered by the parallelization approaches presented in Chapters 5 - 8. Execution costs in CPU cycles and the energy consumption in pJ are automatically extracted (cf. Section 3.2.3) and annotated to the nodes of the graph. Additional objectives can easily be added if they are required by new parallelization techniques, due to the transparent structure of the AHTG. If the chosen parallelization technique considers heterogeneous architectures (like the ones presented in Chapters 7 and 8), the objectives are added once for each processing unit to enable a consideration of performance variations depending on the executing core.

- **Reference to Statement:**
  The relationship of nodes to the intermediate representation's statement is also stored to be able to adapt changes from the graph to the original application code and vice versa.

It is also essential to possess knowledge about the communication costs, which have to be taken into account if the statements of two nodes are executed in different tasks. Therefore, additional information is also added to the edges of the graph (cf. *Edge Info* in Figure 4.1(a)):

- **Edge type:**
  Each edge belongs to one of the dependence types mentioned earlier (like, e.g., Read-after-Write data dependencies) and is annotated to the edges to be later accessible by the parallelization approaches.

- **Communication costs:**
  The communication delay in CPU cycles, if the source and target node of the edge are executed in different tasks. It is estimated by multiplying the amount of communicated bytes with a platform-dependent communication factor. Other objective values, like, e.g., the energy required to communicate the data, are also annotated to the edges.

- **Communicated data:**
  The symbols or expressions that have to be communicated are also annotated to the edges to be later accessible by the parallelization approaches.

- **Iteration count:**
  The iteration count reflects the number of times the communication takes place.

The Augmented Hierarchical Task Graph as well as the attached node and edge information is extracted automatically from sequential ANSI-C code to enable an automatic parallelization flow. While the extraction of the estimated objective values is presented in the framework chapter (cf. Section 3.2.3), the following subsection focuses on the extraction of the graph structure. A more complex example of an AHTG is also presented in the Appendix in Chapter A.1. The figures depict different screen shots of the graphs generated by the parallelization framework for the *spectral* benchmark [Lee13].

4.1.2 Extraction of the AHTG

Since most embedded applications are written in sequential C, the parallelization framework presented in this thesis focuses on the extraction of parallelism from programs written in this programming language. However, most techniques presented in this chapter may also be adaptable to other programming languages.

Algorithm 1 Extraction Algorithm of the Augmented Hierarchical Task Graph

```
1: function ExtractAhtg(IR $ir$)
2:     $rootNode \leftarrow$ CreateHTGStructure($ir$)
3:     AddDependenceEdges($rootNode$)
4:     AugmentAHTG($rootNode$)
5: end function
```

Algorithm 1 shows the structure of the main function extracting an Augmented Hierarchical Task Graph from given sequential application code. The function ExtractAhtg expects an intermediate representation (IR) of the application's source code as input. The intermediate representation employed here is called ICD-C IR [Inf13]. In a first step, the nodes of the graph are created and arranged in the hierarchical structure by a call to the function CreateHTGStructure in line 2. Afterwards, data dependence and control flow edges for jump statements are created by a call to the function AddDependenceEdges in line 3. The insertion of control flow edges for jump statements follows the same rules as the insertion of data-flow edges and is therefore not considered separately. Finally, the function AugmentAHTG is called in line 4 which augments the created nodes and edges with cost information extracted by the corresponding objective estimation pre-processing tool (cf. Section 3.2.3)[1].

4.1.2.1 Extraction of Nodes

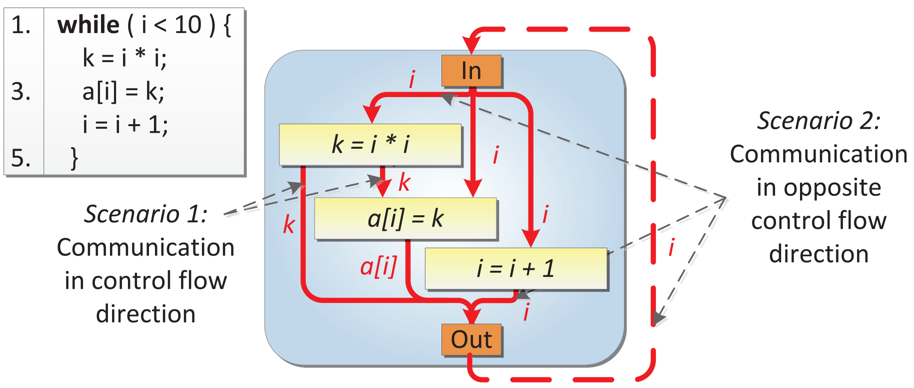
The general structure of the CreateHTGStructure function extracting the nodes of the graph is summarized by the pseudo-code shown in Algorithm 2.

Algorithm 2 Root Structure Creation of the AHTG.

```
1: function CreateHTGStructure(IR $ir$)
2:     $rootNode \leftarrow$ CreateHierarchicalNode()
3:     for $f \in ir.getFunctions()$ do
4:         $fNode \leftarrow$ CreateHierarchicalNode($f$)
5:         $rootNode.addChildNode(fNode)$
6:         $fNode.setParentNode(rootNode)$
7:         CreateNodes($fNode, f$)
8:     end for
9: end function
10:
11: function CreateNodes(HierarchicalNode $parentNode$, Statement $s$)
12:     if $s \in \{ExpStmt, JumpStmt, TargetStmt\}$ then
13:         $node \leftarrow$ CreateSimpleNode($s$)
14:     else
15:         $node \leftarrow$ CreateHierarchicalNode($s$)
16:         for $childStmt \in s.getChildStatements()$ do
17:             CreateNodes($node, childStmt$)
18:         end for
19:     end if
20:     $parentNode.addChildNode(node)$
21:     $node.setParentNode(parentNode)$
22: end function
```

As depicted in line 2, the function first creates a new hierarchical node for the root of the graph representing the entire application. Afterwards, a new hierarchical node is created for each function in line 4. To enable the navigation in both directions, upwards and downwards through the hierarchy of the graph, the function node is added as a child node of the root node and vice versa in lines 5 and 6.

[1]More details on the AugmentAHTG function are omitted due to its simplicity. The function just adds the estimated cost information to the nodes and edges of the graph.



```
while ( i < 10 ) {
   k = i * i;
   a[i] = k;
   i = i + 1;
}
```

Scenario 1: Communication in control flow direction
Scenario 2: Communication in opposite control flow direction
In
k = i * i
a[i] = k
i = i + 1
Out
i
k
a[i]

Figure 4.2: Example for Communication within the same Hierarchical Node (Local Communication).

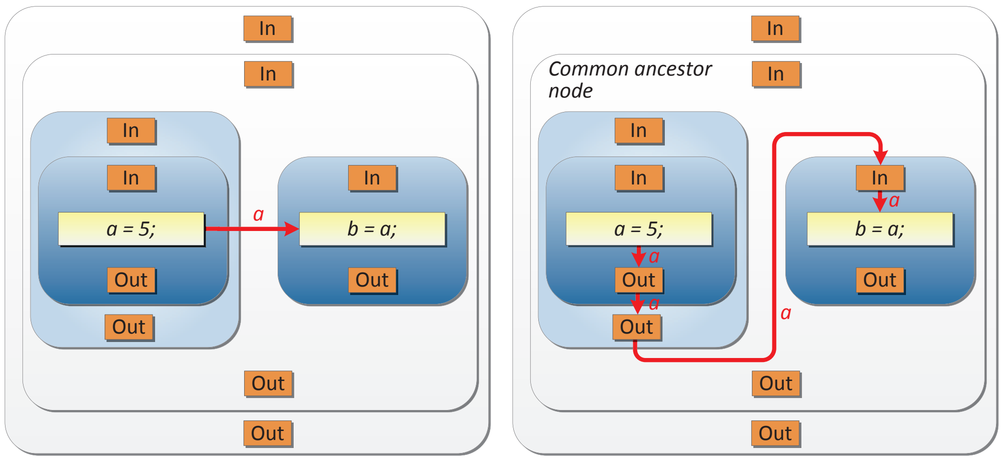
Next, CreateNodes is called for each function, so that child nodes are added for all statements belonging to the corresponding function.

The structure of the CreateNodes function is also depicted in Algorithm 2. It is recursively called for each statement of the current function and creates either a simple or a hierarchical node, depending on the statement's type. If the statement contains further hierarchical structures expressed by compound statements, the function CreateNodes is recursively called until all statements are processed. After the recursive call returns, the hierarchical relationship between the newly created node and its parent node is created in lines 20 and 21. As soon as CreateNodes was called for all functions of the application, nodes are created for all statements, arranged in the desired hierarchical structure. Since the function nodes are added as child nodes of the root node of the graph, all nodes can be reached through the hierarchy starting from the root node.

4.1.2.2 Creation of Dependence Edges

Two properties should be provided by the graph intermediate representation: (a) all sub graphs have to be cycle-free and (b) all communication passes through single entry- and single exit-nodes so that the nodes can be processed in isolation. Therefore, the following four scenarios for adding edges to the graph have to be considered:

Local Communication: Communication between two nodes belonging to the same hierarchical parent node is called local communication and is the simplest form of communication. Figure 4.2 shows an appropriate example containing three nodes which are part of the same hierarchical parent node. For this communication type, two scenarios must be distinguished. In the first one, the dependence edge points in the same direction as the control flow. An example for such an edge can be found for the nodes representing the statements `k = i * i` and `a[i] = k`. The data of variable `k` is computed and communicated in the control flow's direction.

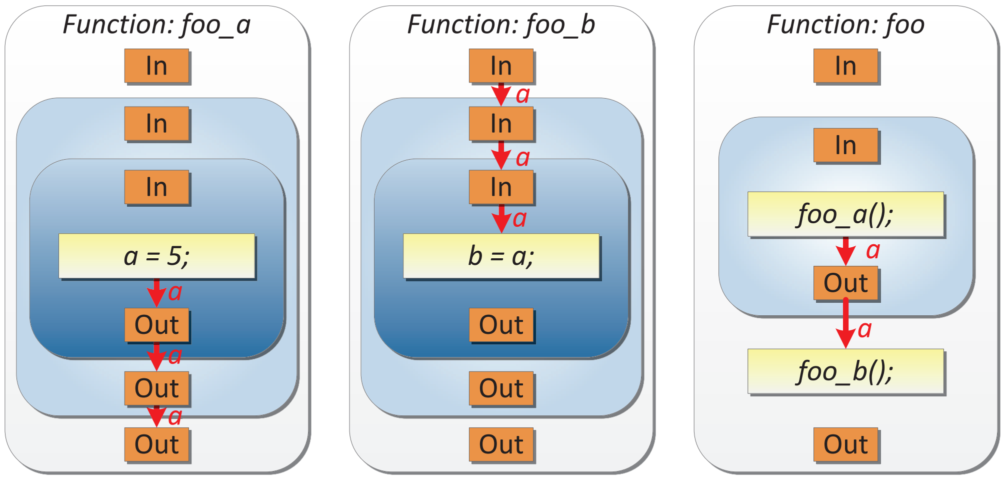


(a) Improper Direct Communication (b) Employed Redirected Communication

Figure 4.3: Communication Example for Statements in the same Function (Global Communication)

To represent this dependency, a new edge is directly added to the graph between both nodes. In the second scenario, source and target nodes still belong to the same hierarchical parent node, but the communication would insert a back-edge in the opposite direction of the control-flow. The statement `i = i + 1`, for example, computes the value of variable `i` for the next iteration of the loop. The new value is not read in the current iteration any more but would insert a cycle if a new edge would directly be added to `k = i * i`. To circumvent this problem, the data of variable `i` is first communicated from the source- to the communication out-node. A second communication edge is then inserted from the communication in-node to the target node. Thus, the data of variable `i` is implicitly communicated from the communication out- to the communication in-node of the hierarchical node. Hence, the data of `i` is available for the next execution of the node, but no cyclic dependencies are added to the graph.

Global Communication: Figure 4.3(a) presents an example of a communication edge which has to be inserted between nodes belonging to different hierarchical nodes (global communication). If the edge would be directly added as depicted in Figure 4.3(a), the single-entry and single-exit property would be violated. In addition, it is not clear if any dependencies between the parent hierarchical nodes would prevent parallel execution on higher hierarchical levels. Therefore, the first common ancestor node (upwards in the hierarchy) of the source and target nodes is determined, like depicted in Figure 4.3(b). The source node communicates the data via communication out-nodes until this first common ancestor node is reached. Afterwards, the data is further communicated via additional communication in-nodes until it reaches the hierarchical level of the target node and finally the node itself. This takes care that the single entry- and single exit-node property is retained, and



(a) `foo_a`: Producing data (b) `foo_b`: Consuming data (c) `foo`: Calling `foo_a`, `foo_b`

Figure 4.4: Communication Example Crossing Function Boundaries (Function Communication)

the influence of dependencies over various hierarchical levels is explicitly expressed. If, for example, the two parent hierarchical nodes shown in Figure 4.3 are loops, both loops cannot be executed in separate tasks fully in parallel because of the dependency contained deeper in the loops' bodies prevents it. This can directly be derived from the structure of the presented AHTG due its communication redirection like shown in Figure 4.3(b), but would not be obvious if the redirection techniques were not applied, like shown in Figure 4.3(a).

Function Communication: The situation gets even more complicated if the source and target node of a dependency are part of different functions, like depicted in the example shown in Figure 4.4. There, function `foo_a` (shown in Figure 4.4(a)) writes to a variable `a` which is consumed by one of the nodes of function `foo_b` (shown in Figure 4.4(b)). Analogously to the previous communication example, the data is first communicated from the source node via several hierarchical levels to the communication out-node of function `foo_a`. Afterwards, the data is communicated through several hierarchical levels from the communication in-node of `foo_b` to the target node. However, the dependency also exists between nodes of the function `foo`, calling `foo_a` and `foo_b` as depicted in Figure 4.4(c). Therefore, new dependency edges have to be added between those nodes as well since the dependence between the nodes of `foo_a` and `foo_b` also sequentializes the execution of both functions on the outer hierarchical level. If additional functions are called between source and target node, all of them are treated like function `foo`.

The three presented examples cover all cases for which edges have to be added to the AHTG. Pseudo-code describing the edge creation algorithms for all cases in more

Algorithm 3 Creation of dependence edges for the AHTG.

```
1:  function AddDependenceEdges(Node $n$)
2:      for $dep \in n.getStmt().getDependencies()$ do
3:          $n2 \leftarrow dep.getTargetNode()$
4:          if $n.getParent() = n2.getParent() \wedge$ not $isBackEdge(dep)$ then
5:              CreateNewEdge($dep, n, n2$) // cf. Figure 4.2
6:          else if $n.getParent() = n2.getParent() \wedge isBackEdge(dep)$ then
7:              // cf. Local Communication (Figure 4.2)
8:              CreateNewEdge($dep, n, n.getParent().getOutNode()$)
9:              CreateNewEdge($dep, n2.getParent().getInNode(), n2$)
10:         else if $n.getFunctionNode() = n2.getFunctionNode()$ then
11:             // cf. Global Communication (Figure 4.3(b))
12:             CreateRedirectedEdge($dep, n, n2$)
13:         else// cf. Function Communication (Figure 4.4)
14:             CreateCallEdge($dep, n, n2$)
15:         end if
16:     end for
17:     if $n \in HierarchicalNode$ then
18:         for $cnode \in n.getChildNodes()$ do
19:             AddDependenceEdges($cnode$)
20:         end for
21:     end if
22: end function
23:
24: function CreateRedirectedEdge(Dependency $dep$, Node $source$, Target $target$)
25:     $cpnode \leftarrow$ GetFirstCommonParent($source, target$)
26:     $node \leftarrow source$
27:     while $node.getParent() \neq cpnode$ do
28:         CreateNewEdge($dep, node.getOutNode(), node.getParent().getOutNode()$)
29:         $node \leftarrow node.getParent()$
30:     end while
31:     $node \leftarrow target$
32:     while $node.getParent() \neq cpnode$ do
33:         CreateNewEdge($dep, node.getParent().getInNode(), node.getInNode()$)
34:         $node \leftarrow node.getParent()$
35:     end while
36: end function
37:
38: function CreateCallEdge(Dependency $dep$, Node $source$, Target $target$)
39:     $node \leftarrow source$
40:     while $node.getParent() \neq source.getFunctionNode()$ do
41:         CreateNewEdge($dep, node.getOutNode(), node.getParent().getOutNode()$)
42:         $node \leftarrow node.getParent()$
43:     end while
44:     $node \leftarrow target$
45:     while $node.getParent() \neq target.getFunctionNode()$ do
46:         CreateNewEdge($dep, node.getParent().getInNode(), node.getInNode()$)
47:         $node \leftarrow node.getParent()$
48:     end while
49:     CreateCallEdges(...) // Continue in call hierarchy...
50: end function
```

detail can be found in Algorithm 3. The function AddDependenceEdges is recursively called for each node of the graph and determines the kind of dependency to call the appropriate function for the edge creation. The function CreateNewEdge creates a new direct edge between two nodes if both nodes belong to the same hierarchical parent node and the new edge does not form a back-edge. If both nodes belong to the same hierarchical parent node but the new edge would form a back-edge, two edges communicating via the communication out and communication in-node are created by two calls to the function CreateNewEdge, respectively. In contrast, if both nodes are part of the same function but belong to different parent hierarchical nodes, the function CreateRedirectedEdge is called creating edges like discussed and shown in Figure 4.3. Finally, if both nodes are part of different functions, CreateCallEdge is called operating like depicted in Figure 4.4. The function `getInNode()` returns the communication in-node for a hierarchical node. Otherwise (e.g., for simple nodes), it returns the node itself since these nodes do not contain communication nodes. The function `getOutNode()` returns the communication out-node for a hierarchical node or the node itself otherwise, respectively.

As soon as the graph is extracted from the source code with all its nodes and dependence edges it gets augmented with additional cost information as shown in Figure 4.1. Based on this intermediate representation, the hierarchical parallelization approach employed in this thesis is presented in the following section.

4.2 Global Hierarchical Parallelization Approach

This section presents the global parallelization approach developed to combine the novel parallelization techniques presented later in this thesis. This section also explains the integration and interaction of the different parallelization techniques, described in Chapters 5 - 8 into the global parallelization approach. The central intermediate representation used is the Augmented Hierarchical Task Graph (AHTG) which was described in detail in the previous section. By exploiting the advantages provided by the AHTG, it is possible to search for parallelism only in a small portion of the application at the same time. Compared to approaches which have to consider the whole application at a time, the global parallelization approach presented in this thesis drastically reduces the size of the vast solution space. This is crucial for the sophisticated parallelization approaches presented later in this thesis.

4.2.1 Overview of the Parallelization Approach

The global structure of the employed parallelization approach is visualized in Figure 4.5. Eight fundamental steps are performed to extract parallelism from sequentially written applications. In detail, these steps behave as follows:

1. **Extract the ICD-C IR and the Augmented Hierarchical Task Graph:** The parallelization framework and its different approaches presented in this

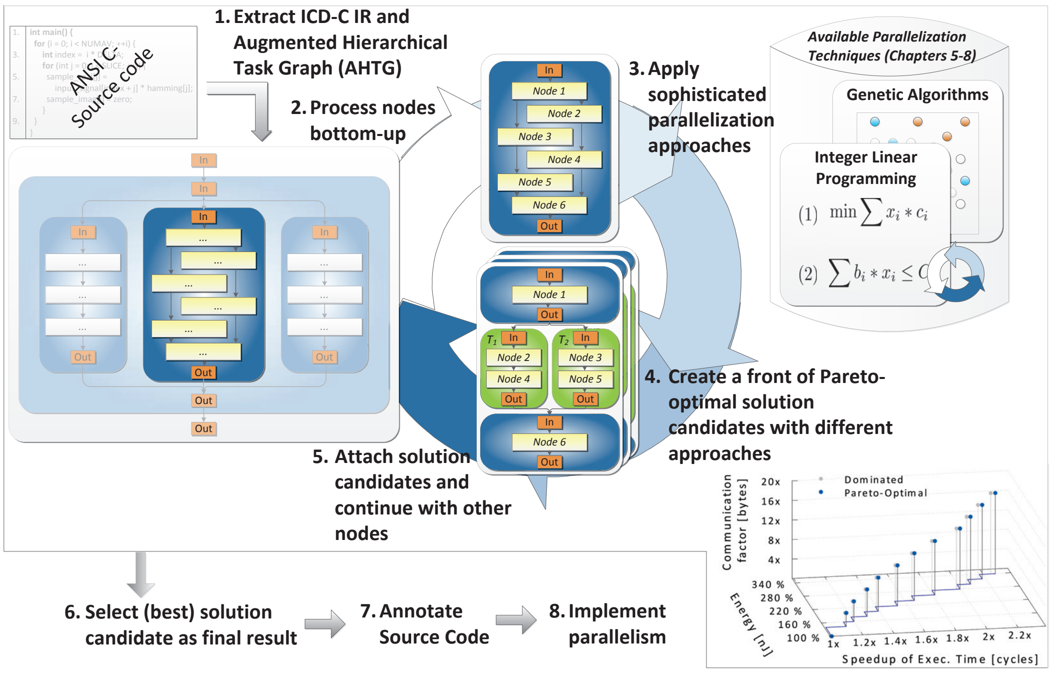

ANSI-C Source code
1. Extract ICD-C IR and Augmented Hierarchical Task Graph (AHTG)
2. Process nodes bottom-up
3. Apply sophisticated parallelization approaches
Available Parallelization Techniques (Chapters 5-8)
Genetic Algorithms
Integer Linear Programming
(1) $\min \sum x_i * c_i$
(2) $\sum b_i * x_i \leq C$
4. Create a front of Pareto-optimal solution candidates with different approaches
5. Attach solution candidates and continue with other nodes
6. Select (best) solution candidate as final result
7. Annotate Source Code
8. Implement parallelism
Dominated
Pareto-Optimal
Communication factor [bytes]
Energy [nJ]
Speedup of Exec. Time [cycles]
In
Out
Node 1
Node 2
Node 3
Node 4
Node 5
Node 6

Figure 4.5: Global Perspective on the presented Parallelization Approach.

thesis are designed to extract parallelism from applications written in sequential ANSI-C code. Therefore, the application's source code is first transformed into a high-level intermediate representation. The source-code intermediate representation employed in this work is called ICD-C IR [Inf13] and has the advantage that its internal structure directly correlates to the structure and components of the application without lowering the source code. This eases the implementation of the extracted parallelism in the final step. Based on the ICD-C IR, the AHTG is extracted as presented in the previous section.

2. **Process nodes bottom-up:**
   Each hierarchical node can be processed in isolation which enables the usage of complex parallelization techniques, due to the hierarchical segmentation of the graph. This advantage is exploited by the presented parallelization approach which extracts parallelism in a depth-first-search manner. Therefore, the algorithm starts with one of the innermost nodes of the graph. By construction, these nodes are either *simple* or *communication nodes* (cf. Section 4.1.1). Since it makes no sense to move a single statement into a task and wait for its completion, only the original sequential solution represented by a so-called solution candidate is created for these nodes. All leaves of the graph are augmented with cost information for the considered objectives, like, e.g., execution time and energy consumption so that this cost information can directly be used as objective values for the created sequential solution candidates. As soon as all child nodes are processed, and a set of parallel solution candidates is created for all of them, the parallelization approach continues to search for parallelism

upwards in the hierarchy. If the algorithm reaches a hierarchical node, one of the parallelization approaches presented in Chapters 5 - 8 or a combination of them is applied to extract parallelism there. Therefore, child nodes can be moved to newly extracted tasks and can be recombined with parallelism which was extracted deeper in the hierarchy (the creation and combination of parallel solution candidates is described in detail in Section 4.2.2). From the perspective of the global parallelization approach, all available parallelization techniques operate in a black-box fashion. The approaches extract parallelism and return objective values for the extracted solution candidates. As a consequence, the global approach and the other techniques do not need to know any details about the extracted solutions since they can rely on these objective values. The following steps 3-5 are now processed for each hierarchical node in isolation in a bottom-up search manner. As soon as all nodes are processed and the root node of the graph is reached, the algorithm continues with step six.

3. **Apply sophisticated parallelization approaches:**
   All child nodes of the node to be parallelized are already processed and a set of parallel solution candidates exists for all of them, due to the bottom-up direction of the parallelization approach. The presented framework of this thesis contains a couple of sophisticated parallelization approaches extracting different kinds of parallelism (task-, loop-, and pipeline-parallelism) in single- and multi-objective aware manners for homogeneous and heterogeneous embedded architectures as presented later in Chapters 5 - 8. Depending on the target platform, the application designer can choose one approach or also a combination of them. All selected parallelization approaches first determine if they are applicable for the specific part of the application which should be processed. For example, the pipeline-parallelization approaches presented in this thesis can only be applied to nodes representing a loop-statement while task-level parallelism can be applied to all hierarchical nodes. If the specific approach is able to extract solutions for the considered node, it is executed once or multiple times with different input parameters (e.g. for the ILP-based approaches) to extract different solution candidates representing parallelized versions of the considered part of the application.

4. **Create a front of Pareto-optimal solution candidates with different approaches:**
   All solution candidates generated by the selected parallelization approaches for the processed node are collected and evaluated for the considered optimization objectives. Afterwards, the front of Pareto-optimal solution candidates is determined so that this front is used as the final solution for the processed node. At this step, no final decision is taken which solution candidate should be used. To be most flexible, all these solutions are offered as solution candidates for this node when the parent node is processed. Thereby, new parallelism can be combined with different solution candidates from the child nodes.

5. **Attach solution candidates and continue with other nodes:**
   The solution candidates of the Pareto-frontier of the node to be parallelized are now attached to the node of the Augmented Hierarchical Task Graph as solution candidates. Afterwards, steps 2-5 can be performed for the next nodes on the same hierarchical level. As soon as all nodes on the same level are processed, the algorithm continues the parallelization process upwards in the hierarchy. Steps 3 and 4 are repeated until the root node of the graph is reached and a set of Pareto-optimal solutions is created containing a combination of solution candidates for all child nodes.

6. **Select (best) solution candidate as final result:**
   If the application designer has chosen a combination of techniques focusing on single-objective aware optimization techniques only, the best solution with respect to this objective is automatically chosen as the final solution candidate. Otherwise, the front of Pareto-optimal solutions is presented by the framework so that the application designer can choose the solution which fits best to a specific application scenario as the final solution.

7. **Annotate Source Code:**
   The presented parallelization framework finally annotates the application's source code according to the selected final solution. Here, the application designer can choose between one of two different annotation types. The first one is based on a separate parallel specification file which maps labeled statements to tasks according to the input specification of the MPA framework [BBW+09]. The second supported code annotation is based on an OpenMP extension, which was specifically developed for usage with embedded devices (cf. PICO in Section 3.1.2).

8. **Implement parallelism:**
   Depending on the chosen output-format, either MPA [BBW+09] or PICO (cf. Section 3.1.2) is chosen for an automatic implementation of the extracted parallelism. Since the application code is only slightly modified by inserted labels or pragma statements, it may also be possible to adapt the generated solutions for other implementation tools easily.

The hierarchical approach also enables an easy integration of additional parallelization approaches in the future. As long as an approach returns a node-to-task mapping for the created solution candidates and evaluation functions for the considered optimization objectives, it can be directly added as one of the available parallelization approaches applied in step 3. The new solution candidates are combined with solution candidates of the existing approaches and – if they optimize at least one objective – they are added to the front of Pareto-optimal solution candidates for the considered node. In this way, an arbitrary number of approaches can easily be combined in a plug-and-play fashion.

The global parallelization algorithm is also visualized in Algorithm 4. It is frequently referenced in the remainder of this thesis to explain the integration of the

Algorithm 4 Pseudo Code of Global Parallelization Algorithm

```
 1: function MAIN(IR $ir$, Platform $pf$, int $maxTasks$)
 2:     $ahtg \leftarrow$ EXTRACTAHTG($ir$) // cf. Algorithm 1
 3:     $solutions \leftarrow$ PARALLELIZEAHTG($ahtg.getRootNode(), pf, maxTasks$)
 4:     IMPLEMENTBESTSOLUTION($ahtg, pf, solutions$)
 5: end function
 6:
 7: function PARALLELIZEAHTG(Node $n$, Platform $pf$, int $maxTasks$)
 8:     // Parallelize bottom-up in hierarchy, first.
 9:     for all $c \in n.getChildNodes()$ do
10:         $cnoderesults \leftarrow$ PARALLELIZEAHTG($c, pf, maxTasks$)
11:         ATTACHSOLUTIONCANDIDATES($c, cnoderesults$)
12:     end for
13:     // Add the sequential solution for all processing units.
14:     $results \leftarrow n.getSequentialSolutions(pf)$
15:     // Apply all activated approaches of Chapters 5-8 and collect results.
16:     for all $approach \in ParallelizationApproaches$ do
17:         if ISENABLED($approach$) $\wedge$ ISAPPLICABLE($approach, n$) then
18:             $r \leftarrow$ PARALLELIZE($approach, n, pf, maxTasks$)
19:             $results \leftarrow results \cup \{r\}$
20:         end if
21:     end for
22:     // Only store Pareto-optimal solution candidates for node $n$.
23:     $optresults \leftarrow$ CREATEPARETOFRONTIER($results$)
24:     // Return all found solutions for current node.
25:     return $optresults$
26: end function
```

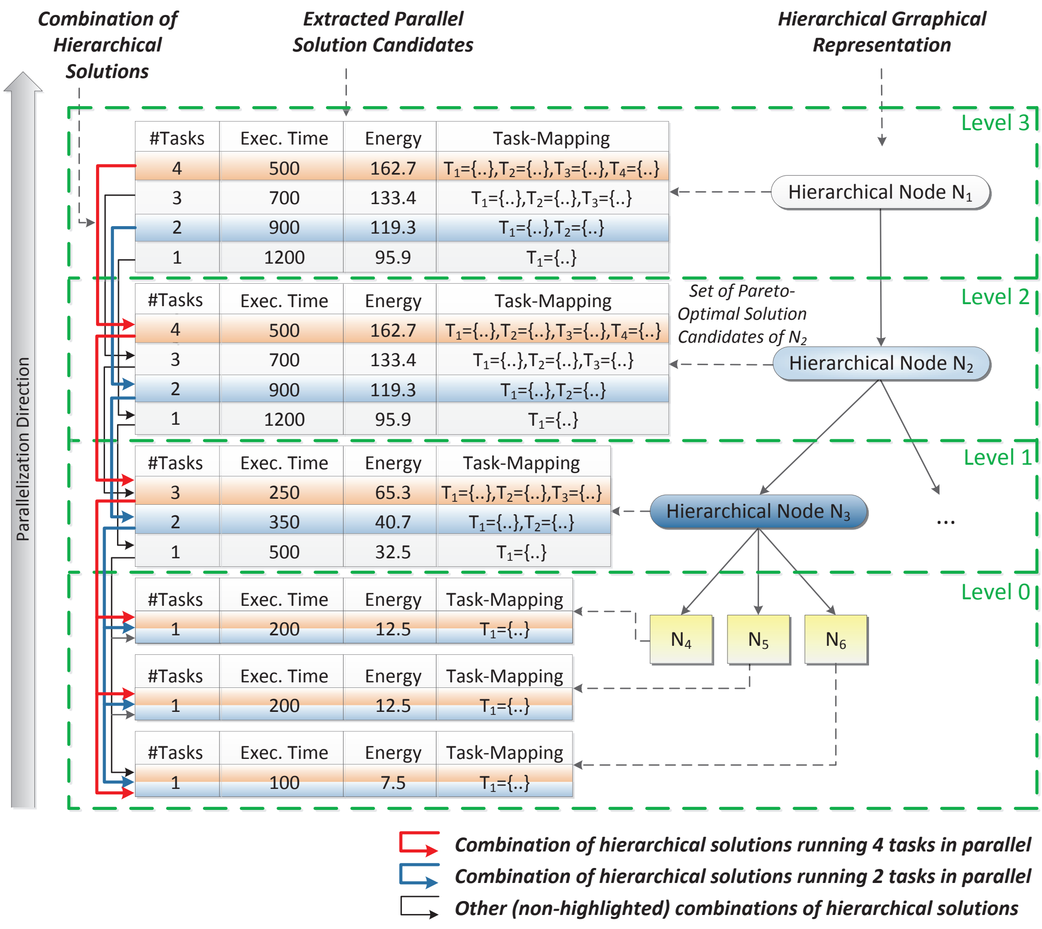
different parallelization approaches into the global parallelization framework. The Augmented Hierarchical Task Graph is extracted in line 2 of the MAIN function. Afterwards, the global parallelization algorithm extracts parallelism in a bottom-up search strategy by calling the function PARALLELIZEAHTG in line 3. The function expects the root node of the Augmented Hierarchical Task Graph, platform information including, e.g., the number and performance characteristics of the available processing units, and an upper bound for the number of concurrently executed tasks, which can be defined by the application designer as arguments. As soon as the function PARALLELIZEAHTG returns, the whole Augmented Hierarchical Task Graph is processed and a set of Pareto-optimal parallel solutions is returned from which one solution is finally implemented in line 4.

The function PARALLELIZEAHTG is called recursively for all child nodes of node n in line 10 first to ensure the bottom-up parallelization methodology. The solution set for node n is initialized with the sequential versions of node n executed on the available processing units in line 14. As a consequence, the parallelization tool flow can always fall back to the sequential version of this node if more efficient parallelism is found upwards in the hierarchy. Afterwards, all enabled and applicable parallelization approaches integrated into the global parallelization tool flow are called one after the other and all generated solution candidates are collected in lines 16-21. Finally, the front of Pareto-optimal solution candidates is determined in line 23 and returned as the result for node n in line 25.

4.2.2 Parallel Solution Candidates

All parallelization techniques presented in this thesis extract one or more so-called parallel solution candidates which are finally combined to a front of Pareto-optimal solutions. Such a front is created for all nodes of the graph. Upwards in the hierarchy, new parallel solution candidates are created and combined with parallel solution candidates of the child nodes deeper in the hierarchy. Each solution candidate contains information about the extracted node-to-task mapping of its direct child nodes as well as cost information for all considered optimization objectives. This cost information is either taken from the augmented cost information of the graph (for sequentially executed nodes) or is estimated by high-level cost models presented later in this thesis. If the considered parallelization approaches are applicable to heterogeneous architectures, they also add information about the determined task-to-processor-type mapping. Since all parallelization techniques annotate cost information to the solution candidates, it is unnecessary for the parallelization techniques upwards in the hierarchy to possess detailed knowledge about the kind of exploited parallelism. This enables a plug-and-play fashion of different parallelization techniques operating as black-boxes in the presented framework.

Figure 4.6 presents a detailed example of the global parallelization approach with respect to the creation and combination of parallel solution candidates. In this example, the focus does not lay on the parallelization approaches and the different extraction and evaluation techniques. The number of extracted tasks, execution times, etc., are only exemplarily chosen and may not necessarily be reasonable. The hierarchical structure of the Augmented Hierarchical Task Graph (AHTG) can also be seen as a tree structure as shown on the right-hand side of this figure. Some of these nodes are omitted to improve readability (further nodes are implied by '...'). The combination of solution candidates through the hierarchy is exemplarily highlighted for two solutions which finally run two and four extracted tasks in parallel (red and blue shapes and edges). As already stated, the parallelization approach starts to extract parallelism using a bottom-up search strategy. The different hierarchical levels are surrounded by dotted shapes and labeled “Level 0” to “Level 3”. As shown in Figure 4.6, the leaves of the graph are located at hierarchical level zero and are invariably simple nodes like, e.g., assignment statements which do not contain any inner structures to be parallelized. Therefore, a parallel set is created for all of them containing only a single sequential solution by taking the objectives values from the augmented information provided by the AHTG. Sequential solutions contain only one (main-)task like shown in the corresponding table depicting the nodes' solution candidates. If the approach extracts parallelism for heterogeneous architectures, more than one sequential solution might be created since the performance of the node's statements may vary on different heterogeneous processing units. For


Combination of Hierarchical Solutions
Extracted Parallel Solution Candidates
Hierarchical Grraphical Representation
Parallelization Direction
Level 3
Hierarchical Node N_1
Set of Pareto-Optimal Solution Candidates of N_2
Level 2
Hierarchical Node N_2
Level 1
Hierarchical Node N_3
...
Level 0
N_4
N_5
N_6
Combination of hierarchical solutions running 4 tasks in parallel
Combination of hierarchical solutions running 2 tasks in parallel
Other (non-highlighted) combinations of hierarchical solutions

#Tasks	Exec. Time	Energy	Task-Mapping
4	500	162.7	T_1={..},T_2={..},T_3={..},T_4={..}
3	700	133.4	T_1={..},T_2={..},T_3={..}
2	900	119.3	T_1={..},T_2={..}
1	1200	95.9	T_1={..}

#Tasks	Exec. Time	Energy	Task-Mapping
4	500	162.7	T_1={..},T_2={..},T_3={..},T_4={..}
3	700	133.4	T_1={..},T_2={..},T_3={..}
2	900	119.3	T_1={..},T_2={..}
1	1200	95.9	T_1={..}

#Tasks	Exec. Time	Energy	Task-Mapping
3	250	65.3	T_1={..},T_2={..},T_3={..}
2	350	40.7	T_1={..},T_2={..}
1	500	32.5	T_1={..}

#Tasks	Exec. Time	Energy	Task-Mapping
1	200	12.5	T_1={..}

#Tasks	Exec. Time	Energy	Task-Mapping
1	200	12.5	T_1={..}

#Tasks	Exec. Time	Energy	Task-Mapping
1	100	7.5	T_1={..}

Figure 4.6: Extraction and Combination of Parallel Solution Candidates.

simplicity reasons, a homogeneous architecture is assumed in Figure 4.6.

As soon as the sequential solution sets are created on the lowest hierarchical level, the parallelization approach moves upwards in the hierarchy to level one and reaches the first hierarchical node N_3. Here, one of the approaches presented in Chapters 5 - 8 or also a combination of them can be used to extract new parallel solution candidates for this node (cf. Algorithm 4 in line 18). Each returned solution candidate is evaluated by high-level cost models and – if it is Pareto-optimal – added to the parallel solution set of node N_3. The different approaches try to extract new parallelism by moving direct child nodes (N_4, N_5, and N_6) of N_3 into concurrently executed tasks and have the possibility to combine these solutions with parallelism which was found deeper in the hierarchy. Therefore, one solution of each child node's solution set (nodes from level zero) is selected for each newly created solution candidate. This combination is depicted by the arrows on the left-hand side of Figure 4.6. Making this decision is trivial on this hierarchical level, since all child nodes (N_4, N_5, and N_6) of N_3 only contain one (sequential) solution. In the given example, three solutions with one, two and three concurrently executed tasks of all extracted solutions remain in the front of Pareto-optimal solutions for node N_3 and are placed in the set of Pareto-optimal solution candidates of this node. Note that hierarchical level one may contain more nodes to be parallelized as indicated by the three dots on the right-hand side of the graph. These nodes have to be processed first before the parallelization approach continues with hierarchical level two.

As soon as all child nodes of hierarchical level one are processed, the algorithm continues with level two which contains hierarchical node N_2. Also here, new tasks can be extracted by parallel executing the child nodes (N_3 and the ones implied by '...'). Furthermore, these tasks can be combined with parallelism which was found deeper in the hierarchy. Therefore, each solution candidate selects the solution candidates of hierarchical level one which should be used like denoted by the edges on the left-hand side of Figure 4.6. In the example shown, the solution with four parallel tasks for node N_2 is based on the solution with three parallel sub-tasks of node N_3 (highlighted by red shapes and edges). In contrast, the solution creating two concurrently executed tasks is based on the solution with two sub-tasks of node N_3 (highlighted by blue shapes and edges). Since the hierarchical solutions from the child nodes' level are also linked to solutions deeper in the hierarchy, a new solution candidate implicitly contains extracted parallelism for all nodes which are processed so far. Besides node N_3, node N_2 also has to choose hierarchical solution candidates for the out-faded node of hierarchical level one ('...'). This step is repeated until the root node of the graph is reached. The four extracted solutions are based on different solution candidates deeper in the hierarchy. One solution, e.g., the highlighted red one implementing four concurrently executed tasks, is finally returned for code annotation and implementation. As can be seen, each solution of the root node implicitly contains one parallel solution candidate for all child nodes.

An approach which directly determines one solution candidate for each node as soon as it is processed would be much easier to implement since each child node would have only one fixed value for each considered objective. Instead, the approach presented here extracts a front of Pareto-optimal solutions for each child node. This makes the extraction step more complicated since the approaches which extract new parallelism also have to decide which hierarchical solutions to select. Depending on the selected solutions the objective values, like, e.g., the execution time of the child nodes, may vary. Nonetheless, the presented approach provides large flexibility on each hierarchical level which is crucial if applications should be efficiently parallelized. By using the proposed approach, the extraction techniques can always fall back to solutions with less or even no parallelism deeper in the hierarchy if more efficient parallelism can be extracted on the current hierarchical level. Vice versa, the algorithms can omit the extraction of new parallelism if, e.g., task-creation and communication-overhead prevent parallel execution on the current hierarchical level and therefore fully or partially rely on parallelism which was extracted deeper in the hierarchy. Of course, also a combination of new and previously extracted parallelism is feasible which highlights the flexibility of the proposed approach.

4.3 Summary

By combining the benefits provided by the Augmented Hierarchical Task Graph (AHTG) with the extraction and combination of Pareto-optimal solution candidates for each node of the graph, a good basis is created enabling complex parallelization techniques, like the ones presented in the following chapters. The AHTG facilitates an infrastructure for an easy combination of multiple parallelization techniques in a black-box fashion extracting different types of parallelism on various granularity levels. Moreover, due to the self-contained, acyclic structure of each hierarchical node, the size of the solution space of the parallelization problem is drastically reduced. This enables complex parallelization techniques which can evaluate the benefit of the extracted parallelism locally.

The front of Pareto-optimal solution candidates created for each node of the graph enables great flexibility on each hierarchical level and allows a combination of new and previously extracted parallelism in a well-balanced fashion. In addition, the structure of the parallelization framework allows an easy integration and combination of new parallelization techniques with existing ones. All techniques can be enabled and disabled in a plug-and-play fashion to extract efficient parallelism for multiple application domains for homogeneous and heterogeneous architectures.

CHAPTER 5

Single-Objective Parallelization for Homogeneous MPSoCs

Contents

This chapter presents the first parallelization approaches developed in the context of this thesis. They are tailored towards the special requirements imposed by embedded MPSoCs. Both approaches, one extracting task-level parallelism and another one extracting pipeline parallelism, focus on the reduction of an application's execution time by partitioning the application into concurrently executed tasks. The parallelization approaches introduced in this chapter are optimized for homogeneous architectures providing multiple processing units with identical performance characteristics. To be able to utilize the targeted MPSoCs efficiently, the presented approaches employ high-level cost models to evaluate and optimize the benefit obtained by parallel execution. The approaches integrate task creation and communication costs which can be configured with respect to the targeted MPSoC architecture. Moreover, processor-specific execution times of individual statements are also integrated into the considered high-level cost models based on high-level simulation (cf. Section 3.2.3). This enables an estimation of the execution time of statements grouped to tasks for various target platforms. The models also support avoiding the creation of tasks for which the benefits achieved by parallel execution are diminished by too expensive communication overhead. To cope with the complexity of the proposed parallelization techniques, the approaches are integrated into the hierarchical parallelization tool flow based on the Augmented Hierarchical Task Graph (AHTG) as described in the previous Chapter 4.

As already discussed, finding an optimal partition of the statements of an application into concurrently executed tasks is an NP-complete problem [Sar89]. As a consequence, no efficient algorithm exists which can be used to solve this problem. But instead of employing inaccurate approximations based on simple heuristics like done in many previously published parallelization approaches, the ones presented in this thesis are based on accurate high-level cost models. One approach that is often used in such a situation is Integer Linear Programming (ILP). Even though ILP has proven to be an excellent optimization technique for many real-world problems like, e.g., partitioning issues, it has not been applied to extract parallelism from sequentially written applications so far. Besides the advantage that ILP defines a clear mathematical problem description, the feasibility of integrating high-level cost models directly in the optimization process seems to be promising to apply it to extract parallelism. Another beneficial property of ILP solvers is that they can determine whether they have found the optimal solution for a given optimization problem. Hence, the optimization process can stop as soon as the solution is found. This is not possible for many heuristic-based optimization techniques, like, e.g., Simulated Annealing (SA) [KGV83]. Such heuristics do not have detailed knowledge about the explored solutions space. Therefore, they only try to generate new solution candidates until a given criterion is met. In addition, ILP solvers guarantee to find the optimal solution – with respect to the applied model – if enough time is given and such a solution exists. Many open-source as well as commercial ILP solvers, like, e.g., lp_solve [BKP13] or IBM's CPLEX [IBM13] exist. Thus, the optimization designer can focus on modeling the ILP system and integrating it into the tool flow and rely on the already existing, highly optimized solvers.

Taking all these things together, ILP is a promising optimization technique which could be applied to extract parallelism from sequentially written applications. The complexity of the created ILPs can be significantly reduced by the hierarchical parallelization approach presented in the previous section. This is substantial if ILPs are to be applied since they are NP-complete in the general case [Mar11]. By using the proposed hierarchical approach, only a small amount of statements are processed at the same time which reduces the ILP systems' problem size to a degree that they can be solved efficiently. Therefore, the approaches presented in this chapter employ ILP to extract parallelism from sequentially written applications. The approaches extract different kinds of parallelism on various granularity levels starting from single-statements over loops to complete function bodies by utilizing high-level cost models for evaluation purposes.

The fundamental concepts of ILP are shortly summarized in Section 5.1, before the first ILP-based parallelization approach extracting task-level parallelism is presented in Section 5.2. Afterwards, Section 5.3 explains the second ILP-based parallelization technique for homogeneous architectures, extracting pipeline parallelism, which is extremely efficient for many embedded applications. Finally, Section 5.4 summarizes the approaches presented in this chapter. Approaches for heterogeneous MPSoCs will be discussed in Chapter 7 onwards.

5.1 Integer Linear Programming

Integer Linear Programming (ILP) is a special form of Linear Programming (LP) forming an optimization technique which is often applied if no efficient algorithms are known to solve a specific optimization problem. By using ILP, a linear optimization function is either minimized or maximized by respecting a set of linear equalities and inequalities. From a geometrical perspective, the ILP system spawns a convex polyhedron and is therefore a special case of a convex optimization. Solving ILP systems is a NP-complete problem [Mar11]. Nevertheless, many small or medium-sized problems can be solved efficiently by commercial as well as open-source solvers.

More formally, the objective function of an ILP has the following form:

$$f\left(x_0, x_1, .., x_n\right) = \sum_{i=0}^{n} a_i * x_i \rightarrow \min \text{ with } a_i \in \mathbb{R}, x_i \in \mathbb{N}_0 \tag{5.1}$$

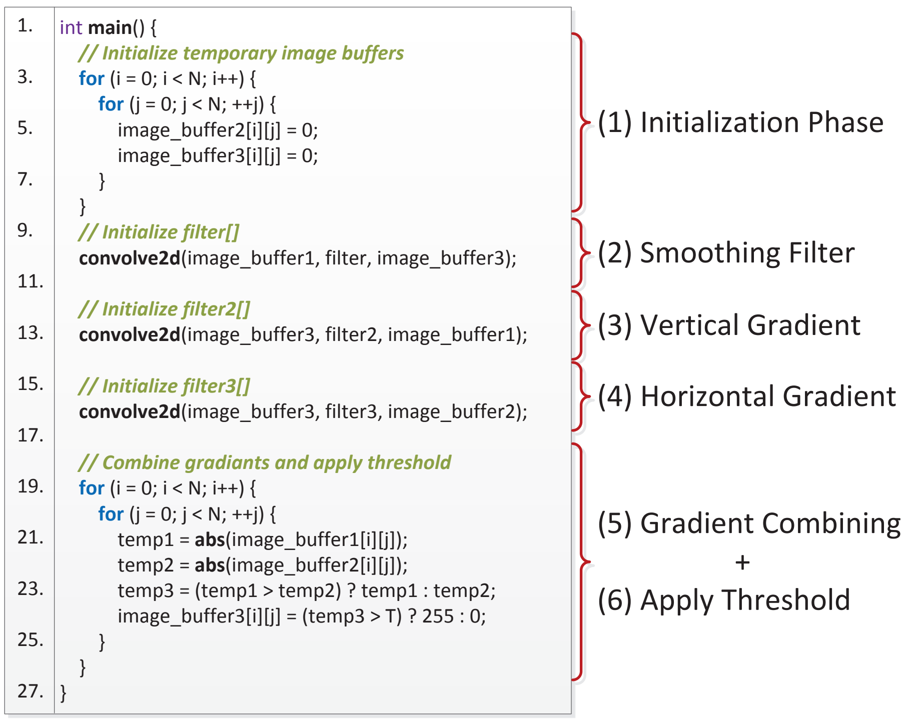
While the set of constraints J is represented by:

$$\forall j \in J : \sum_{i=0}^{n} b_{i,j} * x_i \geq c_j \text{ with } b_{i,j}, c_j \in \mathbb{R}, x_i \in \mathbb{N}_0 \tag{5.2}$$

The minimization of the objective function $f\left(x_0, x_1, .., x_n\right)$ shown in Equation 5.1 can also be changed to a maximization by converting all positive constants a_i to negative ones and vice versa. Constraints shown in Equation 5.2 with '$\geq$' operators can be expressed with '$\leq$' operators in the same way. In addition, a constraint with an '$=$' operator can be substituted by two constraints using a '$\leq$' and a '$\geq$' operator. A subtraction of decision variables x_i is also possible, by a multiplication of their constants with -1. More details on ILP can be found in various books, like [Neu79].

5.2 ILP-based Task-Level Parallelization Approach

One kind of Thread-Level Parallelism which is efficient and often extracted by parallelization frameworks (which do not only extract parallelism from loops of the targeted application) is task-level parallelism. Usually, task-level parallelism divides the statements of the application into coarse-grained, disjunctive tasks operating on preferably independent data sets. In contrast, data-level parallelism tries to duplicate the statements of a loop's body to execute the same task on several processing units. Task-level parallelism often benefits from, e.g., concurrently executed function calls or the parallel execution of several independent loops. The granularity of



```
int main() {
  // Initialize temporary image buffers
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; ++j) {
      image_buffer2[i][j] = 0;
      image_buffer3[i][j] = 0;
    }
  }
  // Initialize filter[]
  convolve2d(image_buffer1, filter, image_buffer3);

  // Initialize filter2[]
  convolve2d(image_buffer3, filter2, image_buffer1);

  // Initialize filter3[]
  convolve2d(image_buffer3, filter3, image_buffer2);

  // Combine gradiants and apply threshold
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; ++j) {
      temp1 = abs(image_buffer1[i][j]);
      temp2 = abs(image_buffer2[i][j]);
      temp3 = (temp1 > temp2) ? temp1 : temp2;
      image_buffer3[i][j] = (temp3 > T) ? 255 : 0;
    }
  }
}
```

Figure 5.1: Task-Level Parallelization Example: Edge Detect Benchmark [Lee13]

the extracted tasks often depends on the targeted application and the parallelization tool itself. Most tools start with basic blocks as their smallest possible parallelization candidates to handle the complexity of the vast solution space. However, due to the employed hierarchical approach, the task-level parallelization technique presented in this section is able to deal with single statements as the most fine-grained task granularity. This ensures great flexibility in the parallelization process.

The rest of this section is structured as follows: First, Section 5.2.1 explains task-level parallelism in more detail and introduces its key properties with a motivating example. Afterwards, Section 5.2.2 shows how the presented parallelization technique is integrated into the global parallelization approach. The parallelization model is explained in Section 5.2.3 before details of the ILP-based extraction technique are presented in Section 5.2.4. Finally, the efficiency of the proposed approach is evaluated in Section 5.2.5.

5.2.1 Motivating Example for Task-Level Parallelism

The example code shown in Figure 5.1 presents the main function of the *edge detect* benchmark from the UTDSP benchmark suite [Lee13]. It detects edges from a 256 level gray-scale image by applying 2D-convolution routines to convolve the image with Sobel operators that expose horizontal and vertical edges. This benchmark was chosen as an example since it is real-world code, which is often applied in embedded systems. In addition, it is useful to present the most important aspects of the presented task-level parallelization approach without being too complex.

As shown on the right-hand side of Figure 5.1, the application can be functionally divided into six phases. The first one is located between lines 2 and 8 and initializes two temporary buffers `image_buffer2` and `image_buffer3` (the input image is stored in `image_buffer1`). The employed hierarchical task graph divides this part of the application into at least three hierarchical levels spawned by the outer loop in line 3, the inner loop in line 4, and the loop body containing the two statements `image_buffer2[i][j]=0` and `image_buffer3[i][j]=0`. The parallelization approach presented in this section tries to extract parallelism in a bottom-up search strategy and has the following options for phase 1:

1. Create two concurrently executed tasks for the statements in line 5 (`image_buffer2[i][j]=0`) and line 6 (`image_buffer3[i][j]=0`) every time the body of the inner loop in line 4 is reached.

2. Extract tasks for the inner loop in line 4 (`for(j=0;j<N;++j)`) executing the iterations of the inner loop in parallel.

3. Extract tasks for the outer loop in line 3 (`for(i=0;i<N;i++)`) executing the iterations of the outer loop in parallel.

In this example, multiple options exist even for the first few lines of the application. Of course, also a combination of parallelism extracted from different hierarchical levels is possible. To be able to determine the best combination of parallelism, high-level cost models are applied. However, since the statements in lines 2-8 only write zero values into temporary buffers, the overhead of task creation and communication may overshadow the benefit of parallel execution. In the worst case, this may lead to a solution which even reduces the performance of the whole application. This should be avoided by the models used.

The last two functional phases of the application (step 5 and step 6 from line 18 up to 26) combine the calculated gradients and apply the specified threshold. They are structured in a similar way to the first phase so that the same three opportunities arise here. A nested loop executes four statements in the inner loop's body. Also here, these statements can be allocated to concurrently executed tasks – as soon as data dependencies do not prevent parallel execution. Moreover, the loops' iterations can also be executed in parallel.

Finally, in stages 2, 3, and 4 several filters are initialized (marked via comments) and a function named `convolve2d` is called three times for different filters and buffers in lines 10, 13, and 16. To keep the example simple, the inner structure of this function is not visualized in Figure 5.1. However, a nested loop is also contained in this function which offers possible parallelism deeper in the hierarchy.

As soon as the whole hierarchical structure of all functional phases is processed by the parallelization approach, the body of the main function starting in line 1 can finally be processed. As described before, both nested loops (in line 3 and line 19) as well as the three calls to the function `convolve2d` offer potential parallelism which

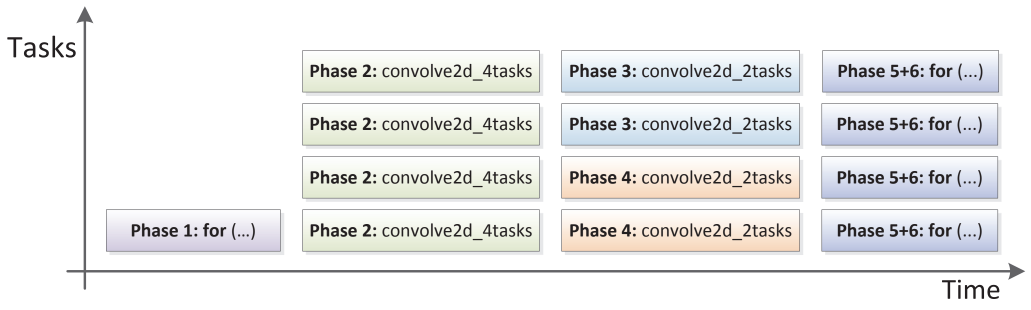

Tasks
Phase 1: for (...)
Phase 2: convolve2d_4tasks
Phase 2: convolve2d_4tasks
Phase 2: convolve2d_4tasks
Phase 2: convolve2d_4tasks
Phase 3: convolve2d_2tasks
Phase 3: convolve2d_2tasks
Phase 4: convolve2d_2tasks
Phase 4: convolve2d_2tasks
Phase 5+6: for (...)
Phase 5+6: for (...)
Phase 5+6: for (...)
Phase 5+6: for (...)
Time

Figure 5.2: Possible Parallel Solution of the Edge Detect Benchmark [Lee13]

was extracted before. With respect to the dependencies between the statements of the function's body, parallelism extracted deeper in the hierarchy may be combined with new parallel sections executing the statements of the function body in parallel.

One possible solution of the parallelized *edge detect* benchmark optimized for a platform with four processing units is visualized in Figure 5.2. The solution is based on the solution candidates discussed above. In this example, the iterations of the outer loop of phase 1 are executed sequentially since the high-level models revealed that the parallel execution of this loop would slow down the overall performance. Afterwards, the second phase is executed containing the first `convolve2d` function which is divided into four concurrently executed tasks inside this function deeper in the hierarchy. Afterwards, new parallelism is implemented directly in the main function's body, which executes phase 3 from lines 12-13 and phase 4 from lines 15-16 in parallel. As a result, the initialization of the filter arrays and the calls to `convolve2d` are executed in parallel. Since only two processing units are allocated so far in this region, each `convolve2d` can be further divided into two additional tasks. Therefore, two implementations of the `convolve2d` function are added to the application's source code with four (`convolve2d_4tasks`) and two tasks (`convolve2d_2tasks`) running in parallel for phase 2, 3, and 4, respectively. Finally, as soon as all tasks executing phase 3 and 4 have finished, the iterations of the nested loop executing phase 5 and 6 are also divided into four concurrently executed tasks. These tasks can either be extracted by one or also by a combination of the proposed parallelization options.

To conclude, this example has shown the following aspects of the presented task-level parallelization approach:

1. Task-Level parallelism can be extracted and combined from different hierarchical levels.

2. The combination of parallelism from different hierarchical levels (e.g., nested loops, function bodies, and function calls) can be beneficial.

3. High-level cost models are necessary to find the best combination of extracted tasks and to avoid too fine-grained solutions, which slow done the application.

4. Specialization of functions with different amounts of tasks can be beneficial.
5. Parallelization tools should be able to tailor solutions to the targeted architecture by, e.g., limiting the number of extracted tasks to the number of available processing units.

5.2.2 Integration into the Global Parallelization Approach

The integration of the presented ILP-based task-level parallelization approach optimized for homogeneous MPSoCs into the global parallelization framework (cf. Section 4.2) is shown in Algorithm 5.

Algorithm 5 Pseudo Code of the ILP-based Task-Level Parallelization Approach

```
1:  // Called bottom-up hierarchically by Algorithm 4 in line 18 on page 56
2:  function ExtractHomTLP(Node $n$, Platform $pf$, int $maxTasks$)
3:      // This function is only applicable to hierarchical nodes.
4:      $solutions \leftarrow \emptyset$
5:      if IsNotHierarchicalNode($n$) then
6:          return $solutions$
7:      end if
8:      // Extract parallelism for hierarchical node n.
9:      // All nodes deeper in the hierarchy are already processed.
10:     $i \leftarrow maxTasks$
11:     while $i >= 2$ do
12:         $result \leftarrow$ HomILPTaskLevelParallelizer($n, pf, i$)
13:         $solutions \leftarrow solutions \cup \{result\}$
14:         $i \leftarrow$ NumberOfTasks($result$) $- 1$
15:     end while
16:     return $solutions$
17: end function
```

The function ExtractHomTLP is executed by the global parallelization algorithm (cf. Algorithm 4) as soon as all child nodes deeper in the hierarchy are processed. As arguments, the function expects the node to parallelize n, platform specific information pf containing, e.g., the performance characteristics and the number of available processing units, and an upper bound of extractable tasks $maxTasks$. The variable $maxTasks$ is set to the number of available processing units of the targeted architecture by default and can be customized by the application designer. Thereby, the parallelization process does not generate more tasks than processing units are available by default. This reduces additional scheduling overhead at runtime which is desirable for embedded systems.

In lines 5-7 the algorithm determines whether the currently processed node n is a non-hierarchical node since only hierarchical ones are processed by the ILP-based homogeneous task-level parallelization approach. It does not make sense to move one node to a separate task and wait for its completion. As a consequence, non-hierarchical nodes are skipped in the parallelization process (lines 5-7). ILP-based optimization techniques always return the best solution candidate – with respect to the applied optimization model – as their only solution. If only one solution would be generated for each node in the graph, the parallelization process upwards in the hierarchy would be limited in its further possibilities. Therefore, the ILP-based parallelization technique described in Section 5.2.4 is executed multiple times in lines 10-15 to generate several solution candidates with at maximum $maxTasks$ tasks down to 2 tasks. Thus, for architectures providing, e.g., four processing units the ILP-based parallelization approach is called up to three times in line 12 with an upper bound of extractable tasks set to 4, 3, and 2. However, due to the optimality of the solutions returned by the ILP solver, some of the iterations can often be skipped. If the parallelization process returns a solution with 2 tasks, even if it was able to generate up to 4 tasks, the iterations with 3 and 2 maximum tasks are skipped since they would create the same solution. All generated solution candidates are finally collected in line 13 and returned as solution candidates for the processed node n in line 16. Upwards in the hierarchy, the generated solution candidates of node n can later be combined with new parallelism for the parent node.

5.2.3 Parallelization Model

This section presents the concepts of the task-level parallelization approach called in line 12 of Algorithm 5 (HomILPTaskLevelParallelizer) before the ILP formulations used are discussed in the next Section. As its target objective, the task-level parallelization approach presented in this chapter tries to minimize the *critical path* (the most expensive one) within a hierarchical node n. In particular, this means that the ILP-based approach aims at the minimization of the costs of the path from the hierarchical node's communication in- to its communication out-node by moving some of its child nodes to concurrently executed tasks. All child nodes are already processed at this point, due to the bottom-up parallelization approach of the framework. Therefore, a set of parallel solution candidates with different execution times depending on the number of extracted tasks exists for all child nodes. The ILP solver is now able to reduce the longest execution path for node n by moving its child nodes to newly created tasks and by selecting hierarchical solution candidates with different granularities for each child node. The computation of the critical path is based on Sarkar [Sar91a]. To be able to adapt this approach to different hardware platforms, a task creation overhead which is added for each created task and a communication overhead which is multiplied by the amount of communicated bytes between different tasks can be specified. By changing these parameters, the user is also able to steer the granularity of the extracted parallelism.

The ILP-based parallelization approach divides the hierarchical node into three sections (cf. Figure 5.3(b)). All statements of the first section belong to the main task and are executed sequentially on the processor which started the execution of the hierarchical node. The second one is the so called parallel section, where different tasks can be executed concurrently. The main task waits until all tasks of this parallel section have finished their work and all data is communicated back. The last section is again a sequential section which belongs to the main task, where

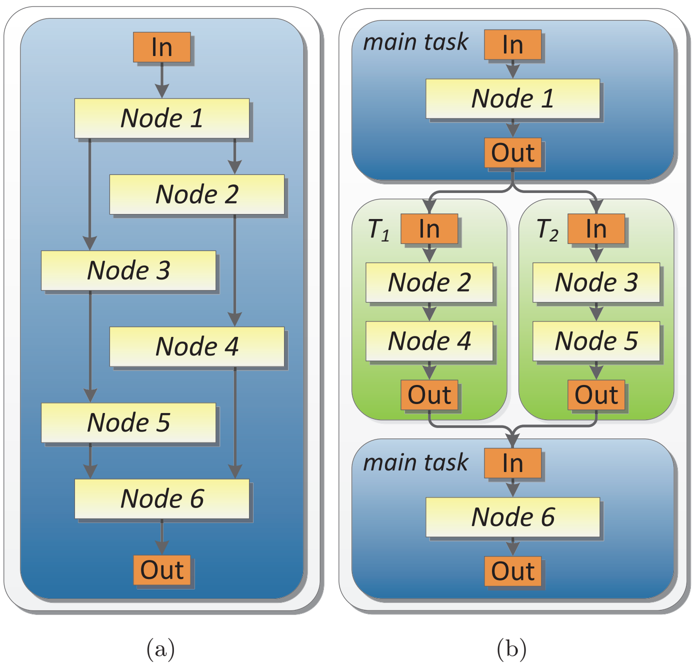


Figure 5.3: Task-Level Parallelization Example

statements are executed without explicit data communication. This kind of segmentation is called fork-join execution model [DM98] and is equal to the one used by MPA [BBW+09] and OpenMP [DM98]. In general, it is often useful to execute some statements before and after a parallel section sequentially to avoid high communication costs for less computationally intensive parts of the application.

An example for such a partitioning is shown in Figure 5.3. The input for the parallelization step is given as a hierarchical node of the AHTG to be processed on the left-hand side in Figure 5.3(a). The sequential, hierarchical node, which should be parallelized, contains six child nodes, its communication in- and communication out-node and several data dependencies, which may produce communication, if the statements are executed in different tasks. The hierarchical node to be processed may be anywhere in the hierarchy and the child nodes can either be simple nodes or hierarchical ones. The right-hand side shows a possible result of the ILP-based approach in Figure 5.3(b). Here, the solver decided to map *Node 1* to the sequential task that is executed in front of the parallel section. Therefore, no data has to be communicated to execute *Node 1*. *Node 2* and *Node 4* are then moved to a newly created task T_1. The second task T_2, which is concurrently executed to T_1, contains *Node 3* and *Node 5*. *Node 6* is finally executed after the two tasks have been synchronized. As a result, two tasks, each one containing two child nodes, are executed in parallel, which may reduce the overall execution time of the hierarchical node. Further reductions may be achieved by choosing non-sequential parallel solution candidates for the child nodes. Especially in this case, the created tasks should be balanced so that all tasks within the parallel section finish nearly

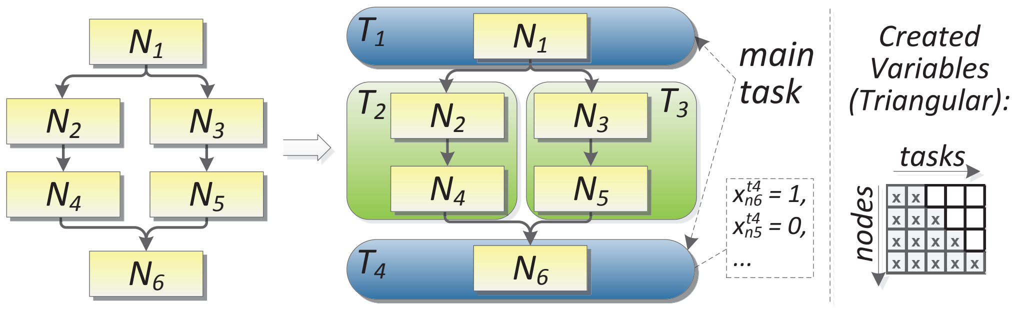


Figure 5.4: Node in Task Constraint

at the same time. Obviously, the well-known Amdahl's law [Amd67] provides an upper bound on the speedup which can be achieved in this way.

5.2.4 ILP-based Parallelization Approach

To cope with the complexity of the parallelization problem, the hierarchical nodes of the AHTG are processed in isolation with the global divide-and-conquer-based parallelization technique. Instead of creating one large ILP, smaller ones for each hierarchical node are created and solved. The extracted solutions are later re-combined with parallelism on the parent hierarchical level. This drastically reduces the search space of the optimization problem so that the ILPs can be used and solved efficiently. The rest of this section defines the ILP formulation used to extract task-level parallelism in the form presented in the previous section. The ILP-based parallelization approach covers the following four main goals:

I) Map statements of direct child nodes into newly extracted, disjunctive tasks to reduce the overall execution time by parallel execution.

II) Combine newly extracted tasks with tasks which were extracted deeper in the hierarchy, if such a solution increases the overall performance (Parallel Solution Candidate Mapping).

III) Keep track of dependencies which may change if child nodes representing statements are moved from one task to another one.

IV) Minimize the overall execution time by taking task creation and communication overhead as well as task execution costs into account.

In the following, decision variables are written in lower case letters, sets start with a capital letter and constants consist of exclusively capital letters. Indices n and o are used for child nodes of the node to be parallelized, t and u represent indices for tasks while hierarchical solution candidates of the child nodes use s as index. Graphical representations for most equations are also given in Figures 5.4 - 5.10 which visualize the decision variables and constraints used.

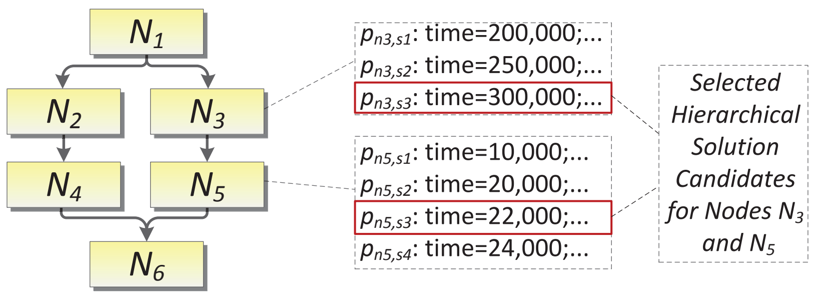


Figure 5.5: Parallel Solution Candidate Constraint

5.2.4.1 Node in Task Constraint

Goal (I) of the approach is a mapping of child nodes to newly extracted, concurrently executed tasks. Therefore, a decision variable x_n^t is defined in Equation 5.3 which denotes whether child node n is mapped to task t. This mapping is also visualized on the left-hand side of Figure 5.4. The tasks with the lowest number (T_1) and the highest one (T_n) belong to the sequentially executed main task. All other tasks are part of the parallel section and are executed concurrently if dependencies do not prevent parallel execution.

$$x_n^t = \begin{cases} 1, & \text{if node } n \text{ is mapped to task } t \\ 0, & \text{otherwise} \end{cases} \tag{5.3}$$

The constraint in Equation 5.4 takes care that every node is mapped to exactly one of the given tasks.

$$\forall n \in Nodes : \sum_{t \in Tasks} x_n^t = 1 \tag{5.4}$$

To reduce the number of created condition variables and constraints, the nodes are first topologically sorted with respect to their dependencies. Afterwards, the first node can only be assigned to the main task or the first task of the parallel section. The second node can further be assigned to the main, the first or the second task of the parallel section and so on. In this way, the variables are created in a triangle form which does not limit the solution quality but reduces the amount of created variables (cf. right-hand side of Figure 5.4).

5.2.4.2 Parallel Solution Candidate Constraint

As explained in the previous section, all child nodes are already processed by the ILP-based parallelization approach since the extraction algorithm parallelizes the application in a bottom-up manner. As a result, all profitable parallel solution candidates were collected in a so-called parallel set for each child node (cf. Figure 5.5). Each set contains at least a sequential solution for each child node and also parallelized versions if they increase the overall performance. Now, the algorithm has to choose one solution candidate for each child node which may contain

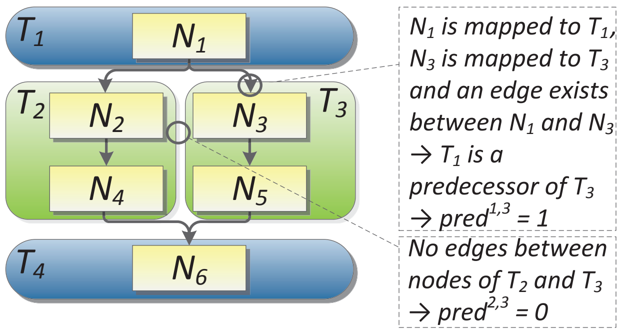

T_1
N_1
T_2
N_2
N_3
T_3
N_4
N_5
T_4
N_6
N_1 is mapped to T_1, N_3 is mapped to T_3 and an edge exists between N_1 and N_3 → T_1 is a predecessor of T_3 → $pred^{1,3} = 1$
No edges between nodes of T_2 and T_3 → $pred^{2,3} = 0$

Figure 5.6: Predecessor Constraint

tasks which were extracted deeper in the hierarchy. In this way, newly extracted tasks are combined with tasks which were found earlier like claimed by Goal (II). Each solution candidate has a different execution time depending on the number of extracted tasks. Equation 5.5 defines variable $p_{n,s}$ which evaluates to 1 if parallel solution s of node n is chosen for n.

$$p_{n,s} = \begin{cases} 1, & \text{if parallel solution } s \text{ of child node } n \text{ is chosen} \\ 0, & \text{otherwise} \end{cases} \tag{5.5}$$

Equation 5.6 takes care that exactly one hierarchical parallel solution is chosen for each child node n.

$$\forall n \in Nodes : \sum_{s \in Solutions_n} p_{n,s} = 1 \tag{5.6}$$

The definition and the way the hierarchical solution candidates are selected were slightly modified compared to the original publication [CMM10]. In the original version, the solution candidates of all child nodes were crosswise combined before the ILP was executed. As a consequence, the solver only had to choose one combination of solution candidates. Hence, depending on the number of extracted solution candidates per child node, a large number of combinations had to be created even if only one of them was chosen and further used. In the new version presented here, the ILP directly selects one solution candidate per child node so that this step can be skipped. This form also enables better opportunities for heterogeneous architectures, like required by the heterogeneous parallelization approaches presented later in this thesis.

5.2.4.3 Predecessor Constraint

Parallel execution is often prohibited by data- or control-flow dependencies which create a predecessor and successor relationship between the extracted tasks. Unfortunately, the dependencies between the different newly extracted tasks are based on the dependencies between the child nodes. Thus, if a dependence edge between two nodes exists and both nodes are mapped to different tasks, the succeeding one has to wait until its predecessor has finished its execution so that the required data can be communicated (cf. nodes N_1 and N_3 in tasks T_1 and T_3 in Figure 5.6). The situation gets even worse if one node is re-allocated from one to another task by the ILP solver since dependencies between the newly extracted tasks may also change. If, e.g., node N_4 in Figure 5.6 would be re-allocated from task T_2 to task T_3, the dependence edge between nodes N_2 and N_4 would enforce the serial execution of the tasks T_2 and T_3. This behavior and the resulting execution order of tasks in the parallel section have to be explicitly modeled since the critical (or most expensive) execution path within the hierarchical node to be parallelized should be used as optimization objective. Techniques determining such paths within ILPs are often used in literature, like, e.g., in [LMD94] and [FSS11]. However, these techniques model paths with static dependencies. Here, the paths dynamically depend on the node-to-task mapping making the problem more complex. Equation 5.7 defines decision variable $pred^{t,u}$ which evaluates to 1 if task t is a direct predecessor of task u.

$$pred^{t,u} = \begin{cases} 1, & \text{if task } t \text{ is a direct predecessor of task } u \\ 0, & \text{otherwise} \end{cases} \tag{5.7}$$

The relation between the newly extracted tasks must be expressed in the ILP as claimed by Goal (III) and depends on the node-to-task mapping, modeled by decision variable x_n^t (cf. Equation 5.3). If a dependence edge from node n to node o exists ($EDGE_{n,o} = 1$) and both nodes are mapped to different tasks t and u, then task t is a direct predecessor of u, like denoted in Equation 5.8.

$$\begin{aligned} &\forall t, u \in Tasks : \forall n, o \in Nodes : t \neq u : n \neq o : EDGE_{n,o} = 1 : \\ &\qquad pred^{t,u} \geq x_n^t \wedge x_o^u \end{aligned} \tag{5.8}$$

If task t is a predecessor of task u, then u has to wait until t has finished its execution since u has to consume data produced by task t. The $\wedge$ operator used in Equation 5.8 is not part of regular ILP formulations. Nevertheless, it can be substituted by a new variable and three inserted constraints as shown in the Appendix in Section A.2.1.

The predecessor variable $pred^{t,u}$ is created for all possible task combinations. From a technical perspective, it should be mentioned that the constant $EDGE_{n,o}$ is known when the ILP is created. Therefore, constraints are only generated if a direct edge between n and o exists. Two further exceptions are related to the calculation of the path information: the sequential block of the main task right before the parallel section (cf. Figure 5.3) is a predecessor of all other tasks. Analogously, the sequential block after the parallel section is a successor of all other tasks or vice versa: All other tasks are predecessors of the sequential task after the parallel section.

5.2.4.4 Task Execution Costs Constraint

The presented ILP-based parallelization approach should be able to balance the extracted tasks automatically so that all tasks finish nearly at the same time before

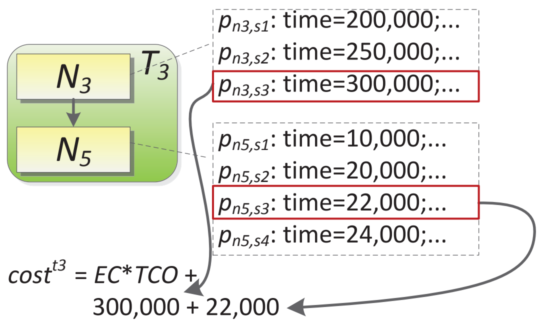


Figure 5.7: Task Execution Costs Constraint

the parallel section is left. Therefore, execution costs are estimated for all tasks to weight the different execution paths extracted in the next step. Equation 5.9 defines variable $cost^t$ which represents the execution costs of task t.

$$\forall t \in Tasks : cost^t = EC * TCO + \sum_n \sum_s (x_n^t \wedge p_{n,s}) * COSTS_{n,s} \tag{5.9}$$

The execution costs $cost^t$ of task t consist of a configurable task creation overhead TCO multiplied by the execution count EC of the node to be parallelized. This overhead is increased by the execution costs[1] $COSTS_{n,s}$ of all nodes n which are mapped to task t depending on the chosen parallel solution candidate $p_{n,s}$ (cf. Figure 5.7). By adjusting the constant task creation overhead TCO, the approach can be ported to various target platforms since this overhead directly influences the granularity of the extracted tasks. It should also be mentioned here that $cost^t$ is part of the objective function so that it is automatically minimized by the ILP.

The execution costs belonging to the two parts of the sequential main task are calculated in a slightly different way. Because they are executed on the same processing unit as the previous nodes, the task creation overhead $EC * TCO$ of Equation 5.9 can be ignored for them so that only the execution costs of the nodes are summed up.

5.2.4.5 Path Cost Constraint

Based on the knowledge of the predecessor relationships and the execution costs of each task, it is now possible to define the accumulated costs of all possible execution paths within the hierarchical node to be parallelized as stated by Goal (IV). Each path cost variable $accumcost^t$ for task t contains execution and communication costs for task t itself and all previous tasks on the path starting with the sequential task in front of the parallel section (cf. Figure 5.8). The order in which the data will be communicated between the extracted tasks is not known at this time. This decision is later taken by one of the parallelization implementation tools. Therefore, a best-case and a worst-case scenario are shown here which can be selected by the user to

[1]The variables $COSTS_{n,s}$ were calculated deeper in the hierarchy and are therefore constants for the parallel solution candidates.

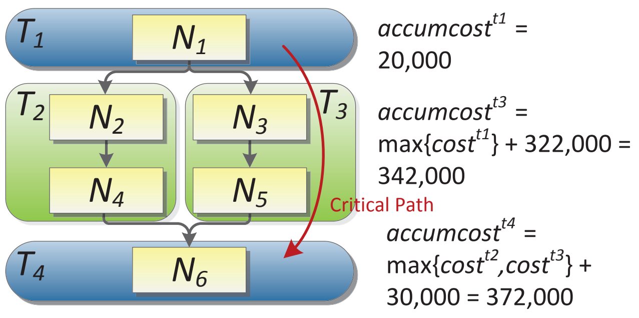

T_1
N_1
T_2
N_2
N_3
T_3
N_4
N_5
T_4
N_6
Critical Path
accumcost^{t1} = 20,000
accumcost^{t3} = max{cost^{t1}} + 322,000 = 342,000
accumcost^{t4} = max{cost^{t2},cost^{t3}} + 30,000 = 372,000

Figure 5.8: Path Cost Constraint

calculate the path costs. However, experience has shown that the best-case scenario estimates too optimistic execution costs for at least the target platforms considered in this thesis. As a consequence, the worst-case formulation was chosen to generate the results presented later in this chapter.

Equation 5.10 defines the accumulated cost information $accumcost^t$ for task t estimated by the best-case communication model. Here, the accumulated path costs are equal to the execution costs $cost^t$ of task t itself increased by execution costs of the most expensive predecessor task u and the communication costs $commcost^{u,t}$ between both tasks. Thus, it is assumed that the data is directly communicated from task u to t as soon as u has finished its execution. The communication costs are determined by multiplying the amount of communicated bytes by a configurable communication factor to adjust the approach to different target platforms. The precondition $pred^{u,t} = 1$ can be ensured by subtracting a constant from the right-hand side of the equation whose value is greater than the sum of all other possible values if the precondition is not met like shown in the Appendix in Section A.2.2.

$$\begin{gathered}\forall t, u \in Tasks : pred^{u,t} = 1 \Rightarrow t \neq u : \\ accumcost^t \geq cost^t + accumcost^u + commcost^{u,t}\end{gathered} \tag{5.10}$$

The worst-case scenario presented in Equation 5.11 assumes that task t has to wait for its input data until all its predecessor tasks have communicated all data to their successor tasks, even if this data is not consumed by t. In contrast to the best-case model, the communication costs $commcost^u$ contain all communication costs of task u even if the data is not communicated to task t to express that the required data is communicated at the end of u's communication phase.

$$\begin{gathered}\forall t, u \in Tasks : pred^{u,t} = 1 \Rightarrow t \neq u : \\ accumcost^t \geq cost^t + accumcost^u + commcost^u\end{gathered} \tag{5.11}$$

The accumulated path costs are also part of the objective function and are automatically minimized by the ILP solver.

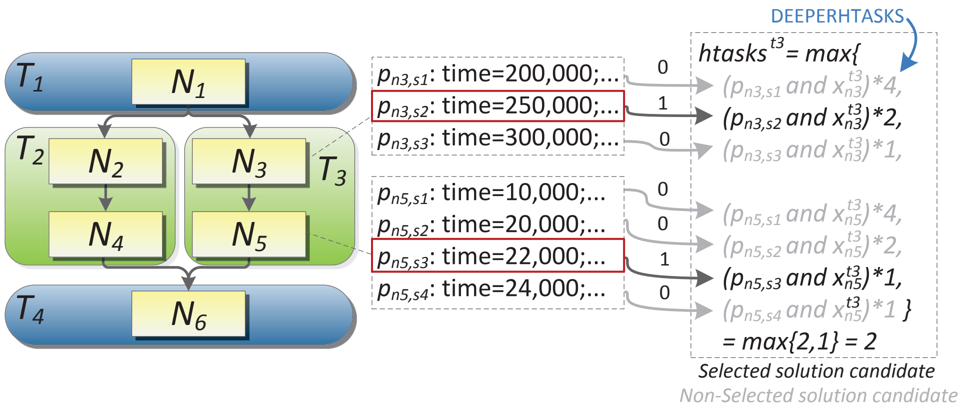


Figure 5.9: Number of Hierarchical Tasks Constraint

5.2.4.6 Number of Hierarchical Tasks Constraint

The avoidance of additional runtime overhead is important for resource-restricted embedded devices. Therefore, the application designer should be able to restrict the number of parallel executed tasks in order to avoid scheduling overhead at runtime. Each child node may contain a number of concurrently executed tasks deeper in the hierarchy, depending on the chosen parallel solution candidate. To be able to restrict the overall number of concurrently executed tasks, the amount of tasks executed deeper in the hierarchy $htasks^t$ has to be calculated for all newly extracted tasks t, based on the node-to-task mapping. Since all nodes within the same task are executed sequentially, the hierarchical tasks of these nodes will not overlap. Thus, the highest amount of concurrently executed hierarchical tasks at a time is as large as the maximum number of hierarchical tasks of all nodes mapped to the new task (cf. Figure 5.9). This is expressed by the constraint defined in Equation 5.12.

$$\begin{aligned}\forall t \in Tasks : \forall n \in Nodes : \forall s \in Solutions_n :\\ htasks^t \geq (x_n^t \wedge p_{n,s}) * DEEPERHTASKS_{n,s}\end{aligned} \tag{5.12}$$

The number of hierarchical tasks $DEEPERHTASKS_{n,s}$ for node n depends on the chosen parallel solution candidate $p_{n,s}$ and is a constant since it is known at the time when the ILP is created for the hierarchical node to be parallelized.

5.2.4.7 Max. Number of Concurrently Executed Tasks Constraint

To limit the number of concurrently executed tasks, a new decision variable $taskused^t$ is introduced in Equation 5.13 which defines if task t is used.

$$taskused^t = \begin{cases} 1, & \text{if task } t \text{ is used} \\ 0, & \text{otherwise} \end{cases} \tag{5.13}$$

Task t is used if it contains at least one node like ensured by Equation 5.14 (cf.

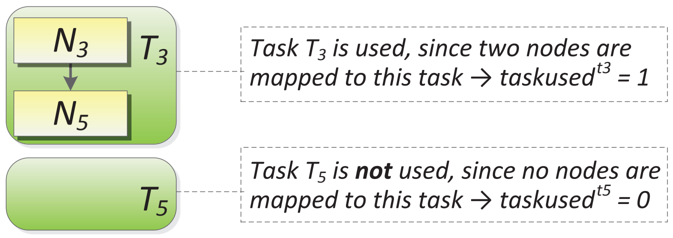

N3
N5
T3
T5
Task T3 is used, since two nodes are mapped to this task → taskused^t3 = 1
Task T5 is not used, since no nodes are mapped to this task → taskused^t5 = 0

Figure 5.10: Max. Number of Concurrently Executed Tasks Constraint

Figure 5.10).

$$\forall t \in Tasks : \forall n \in Nodes : taskused^t \geq x_n^t \tag{5.14}$$

The number of concurrently executed tasks is now equal to the sum of newly created tasks increased by the number of hierarchical tasks. Here, only the parallel tasks are taken into account since the executed tasks in the sequential parts are already synchronized before the parallel section is entered. Equation 5.15 takes care that the number of newly created and combined hierarchical tasks does not exceed the given upper bound $MAXTASKS$ of concurrently executed tasks.

$$MAXTASKS \geq \sum_{t \in Tasks} (taskused^t + htasks^t) \tag{5.15}$$

The ILP formulation can further be optimized here by excluding solutions with the same objective value, which only differ in the allocated tasks (cf. [LMM+97] and Section 9.2). This could be achieved by constraints like $taskused^t \geq taskused^{t+1}$ and may be integrated into the system in the future.

5.2.4.8 Cycle-Free Constraint

To avoid deadlocks and paths with infinite costs, the approach has to take care that the paths within the node to be parallelized are cycle-free. Therefore, all direct child nodes are topologically sorted by their dependencies and an ascending, unique id is generated for both, nodes (*nodeid*) and tasks (*taskid*). W.l.o.g., the generated task graph is cycle-free if the *taskid* of node n is greater or equal to the *taskid*s of all nodes o with a smaller *nodeid*. This is shown by Equation 5.16.

$$\forall n, o \in Nodes : nodeid_n \geq nodeid_o : taskid_n \geq taskid_o \tag{5.16}$$

5.2.4.9 Objective Function

Based on all defined decision variables and constraints, it is now possible to construct the objective function. As mentioned before, the critical or most expensive path from the communication in- to the communication out-node should be minimized. The costs of this path are stored in the value of the variable *accumcost* of the sequential out task t_{seqout} (the last one which is the second part of the main task) because it is the successor of all other nodes in the created task graph. Therefore, it should be minimized by the ILP solver as defined in Equation 5.17.

$$exectime = accumcost^{t_{seqout}} \rightarrow \min \tag{5.17}$$

The value of the objective function is equivalent to the execution time required to process the parallelized hierarchical node. It is hence returned together with the node-to-task mapping and the chosen hierarchical solution candidates as result of the parallelization step.

Even though the definition of the presented ILP-based parallelization technique is complex, it can be solved quickly for the considered real-world benchmarks as shown in the next section. In most cases, the hierarchical task graph reduces the amount of child nodes from five up to fifteen nodes on each hierarchical level. Empirical results have shown that most of the ILPs can be solved in less than a second using the commercial ILP solver CPLEX [IBM13].

5.2.5 Experimental Results

To evaluate the efficiency of the presented ILP-based task-level parallelization approach for embedded homogeneous multi-core architectures, results for eleven embedded benchmarks are presented. Most of them belong to the UTDSP benchmark suite [Lee13] containing representative real-world embedded applications. In addition, other meaningful embedded applications like, e.g., a *jpeg2000* encoder and an implementation of the so-called *boundary value problem* from a physics application domain were evaluated.

Three different target platforms were used to further highlight portability to different target architectures. The first one is the Arm11MPCore platform (cf. Section 3.3.3) containing one ARM11MPCore multi-core processor [ARM13b] providing four ARM cores. The second one is the ARM11QuadProc multi-processor architecture (cf. Section 3.3.2) equipped with four ARM1176 single-core processors [ARM13a]. The last considered platform is the MPARM platform (cf. Section 3.3.1) containing four ARM7 single-core processors [ARM13c]. While the first two platforms are evaluated by the instruction-accurate simulator contained in the Virtualizer tool suite [Syn13b], the last one is simulated by the cycle-accurate MPARM simulator [BBB+05].

In order to execute the output of the parallelization framework on the simulators, some additional work had to be done. The MPA tool [BBW+09] is used to implement the extracted parallelism specified by the presented parallelization approach. MPA requires a special run time library (RTLIB), which had to be ported to the different platforms. RTEMS (Real-Time Executive for Multiprocessor Systems) [RTE13] is used as real-time operating system and had to be adapted to the different target platforms as well. To build a bridge between the operating system and RTLIB, a library called R2G (RTEMS and RTLIB Glued Together) [Hei10] is employed. These middleware adaptations were mostly done by Andreas Heinig.

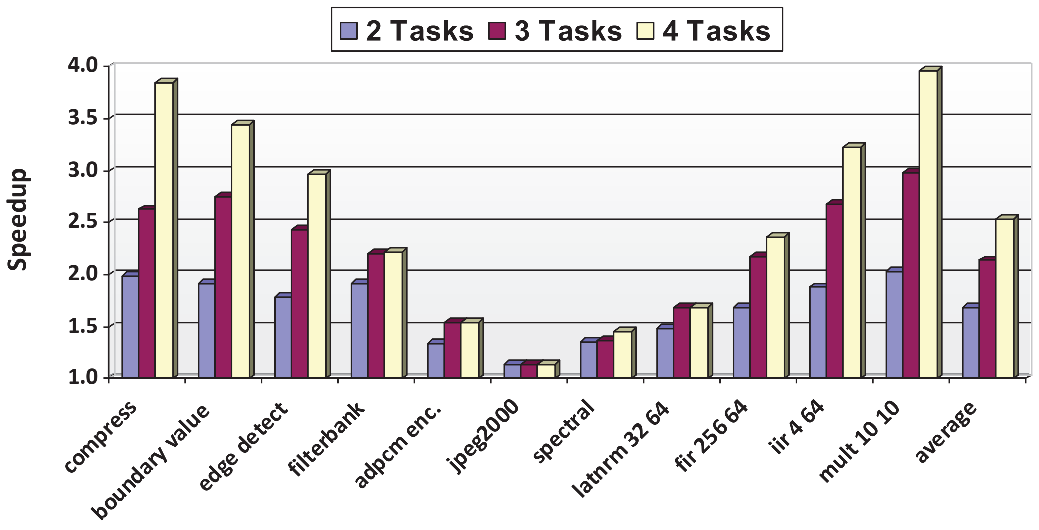


Figure 5.11: Speedup of Homogeneous Task-Level Parallelization Approach (Arm11-MPCore Platform)

The results presented in Figures 5.11 - 5.13 include the execution time of the application without the initialization phase of the operating system and the run-time library because this overhead affects both the sequential and the parallelized version of the application. Hence, the results focus on the execution time of the application itself. The parallelization approach was limited to two, three and four tasks as an upper boundary of concurrently executed tasks by the $maxTasks$ variable to evaluate the extracted speedup for different configurations of the considered target platforms. The baseline of all diagrams is the sequential execution time of the applications on one of the available cores. The original publication in [CMM10] contains only results for the MPARM platform so that two additional architectures are evaluated here. Furthermore, this section presents new results for more benchmarks compared to [CMM10]. Some of the results may differ from the originally extracted ones since many parts of the parallelization tool flow changed in the last years including, e.g., the parallelization tool itself but also the operating system, the simulators and other parts of the middleware which all have a big influence on the extracted speedup.

As can be seen in Figures 5.11 - 5.13, the speedup scales well with the given amount of extracted tasks for most of the benchmarks. The extracted speedups of the different target platforms are further comparable and differ only marginally for the different platforms. It was, e.g., possible to accelerate the *compress* benchmark by nearly 2.0×, 2.6×, and 3.9× for two, three and four tasks running in parallel on the Arm11MPCore platform, respectively. For the ARM11QuadProc and MPARM platforms marginally lower speedups of nearly 2.0×, 2.6× and 3.8× were reached which shows that the approach is able to extract reasonable speedups for several target platforms. Other applications like the *edge detect* benchmark, require more

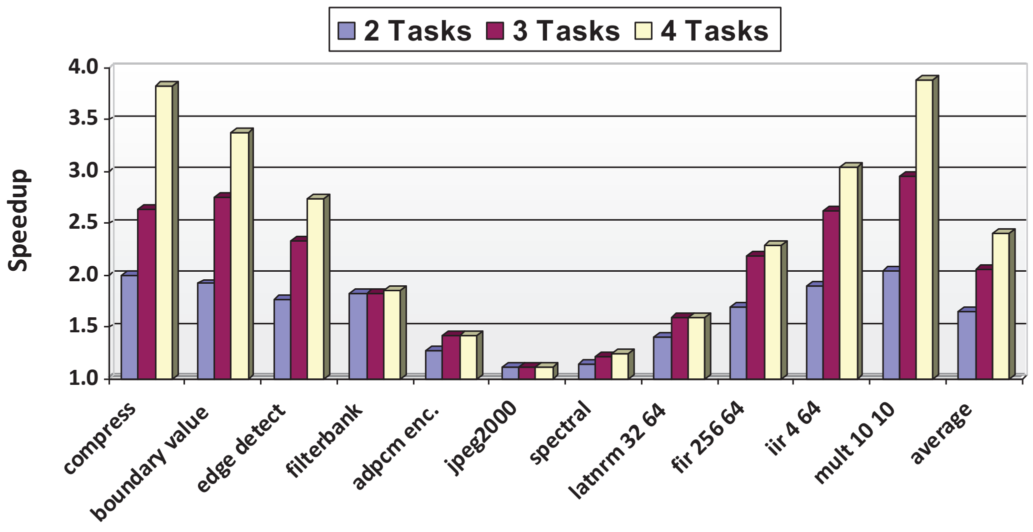


Figure 5.12: Speedup of Homogeneous Task-Level Parallelization Approach (Arm11-QuadProcessor Platform)

communication between the extracted tasks so that the speedup is not as high as observed for the *compress* benchmark. But also here, speedups of nearly 1.8×, 2.4×, and 3.0× were reached for the Arm11MPCore platform while 1.8×, 2.3×, and 2.7× speedups were extracted for the ARM11QuadProc platform. The MPARM platform performed slightly faster than the ARM11QuadProc platform with speedups of 1.8×, 2.4×, and 2.9×, respectively. Some of the benchmarks like the *boundary value problem* show larger differences between the considered target platforms but most results are comparable for all evaluated platforms.

However, in contrast to benchmarks like *compress*, *edge detect*, and *mult* for which the task-level parallelization approach was able to extract efficient parallelism, some benchmarks perform not so well. The benchmarks *jpeg2000* and *spectral*, for example, contain loops with loop-carried dependencies, which can hardly be handled by task-level parallelism. Therefore, additional parallelization methods (like the pipeline parallelization approach presented in the next section) are required to be well applicable for benchmarks from various application domains. But even though some of the benchmarks perform not so well, an average speedup of 1.7×, 2.1×, and 2.6× could be achieved for the Arm11MPCore platform, respectively. Moreover, also for the ARM11QuadProc architecture, speedups of 1.7×, 2.1×, and 2.4× could be reached. The MPARM platform is, on average, the slowest one with speedups of 1.6×, 2.0×, and 2.4×, respectively. Since these average speedups also contain results for which task-level parallelism was not well applicable, the results show that this first approach is a good basis for other parallelization techniques presented later in this thesis. In addition, efficient solutions could be extracted for more than 50% of all considered benchmarks with speedups of up to 3.9× for the considered platforms providing four processing units.

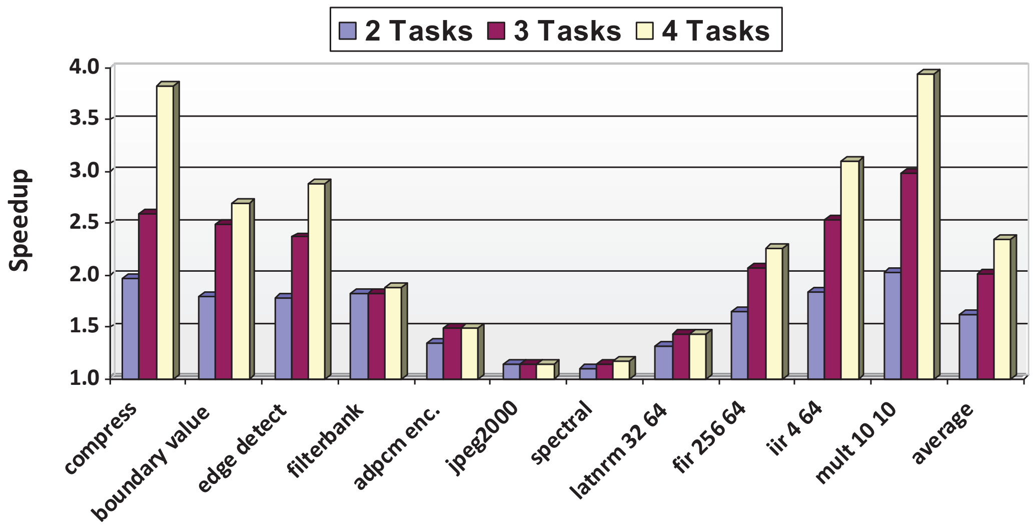


Figure 5.13: Speedup of Homogeneous Task-Level Parallelization Approach (MPARM Platform)

5.2.5.1 Exemplary Description of Extracted Parallelism

In order to present the conceptual possibilities of the extracted parallel solution candidates exploited by the presented task-level parallelization approach, the structural results for two, three, and four concurrently executed tasks for the *edge detect* benchmark are visualized in Figure 5.14. This application provides a good trade-off between computational work and simplicity in its structure and was also used as a motivating example at the beginning of this section.

The structure of the sequential application shown in Figure 5.14(a) consists of six phases. After a short initialization phase, three filters are applied to the original image in order to extract horizontal and vertical edges from the input image. All three filters are applied by the same function `convolve2d` which is called with different input filters. The result of the smoothing filter (the first call) is used as input for the second and third function calls. The two filtered images are then combined and compared to a predefined threshold. This is done in a loop which is iterating over the image dimensions. The first difference which can be observed in the solutions presented in Figures 5.14(b) - 5.14(d) compared to the parallelization example shown in Figure 5.2 is that phases five and six (gradient combining and threshold application) are executed sequentially for all created solutions since the employed high-level models revealed that the parallel execution of this part slows down the application due to high task-creation and communication costs.

The result of the parallelization tool, limiting the number of concurrently executed tasks to two, is visualized in Figure 5.14(b). There, the `convolve2d` function is parallelized using two tasks to compute the result of the smoothing filter. Instead of using this parallelization method also for the two succeeding filters, which seems to be the most obvious solution, the parallelization tool decided to duplicate the

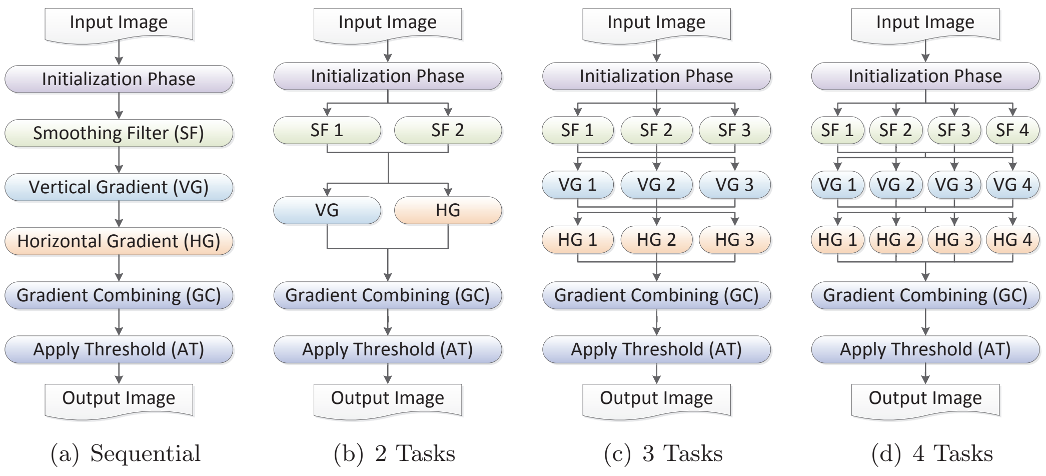


Figure 5.14: Parallelization of the Edge Detect Benchmark

`convolve2d` function. One version of this function implements two concurrently executed tasks while the second one is only executed sequentially by the main task. The latter sequential version is then used for the horizontal and vertical filters because the tool detected that maximum benefit is obtained by parallelizing both function calls instead of just parallelizing the functions itself. A combination of a parallel execution of both functions with two inner tasks would exceed the maximum task boundary. For this solution, nearly the entire application is executed in parallel while the number of concurrently executed tasks is limited to two. This results in the presented speedup of nearly 1.8× for the Arm11MPCore platform.

The results for the three and four core versions are visualized in Figures 5.14(c) and 5.14(d). In contrast to the two core version, the parallel result differs in the implementation of the vertical and horizontal function calls. The version with two concurrent function calls to the `convolve2d` function is no longer used since this would only lead to two concurrently executed tasks. Therefore, both function calls are executed one after the other with three and four concurrently executed tasks in each function, respectively. The achieved speedup with three concurrently executed tasks is 2.4× while the measured speedup of four tasks is 3× for the Arm11MPCore platform. It could also be observed that the speedup further increases for growing input image sizes because the contribution of the initialization phase to the overall execution time decreases. Besides these results, this example also demonstrates the described trade-off technique, deliberating which parallelism is most suitable for the given architectural configuration.

5.2.5.2 Optimization Time & ILP Statistics

Further statistics including the execution times which were required to extract the presented results by the ILP-based parallelization approach and additional ILP-based statistics are presented in Table 5.1. All results are based on the extraction

	Execution Times of Approach in MM:SS[2]				ILP Statistics (for 4 Tasks)		
Benchmark	Preproc.	2 Tasks	3 Tasks	4 Tasks	#ILPs	#Var	#Constr
adpcm enc.	00:08	00:02	00:02	00:02	23	4,425	10,178
bound. value	00:08	00:02	00:03	00:06	7	1,517	4,145
compress	00:49	00:03	00:20	00:21	40	9,649	22,806
edge detect	00:28	00:06	00:06	00:08	49	9,601	20,558
filterbank	00:55	00:01	00:01	00:09	6	2,046	5,023
fir 256 64	00:08	00:01	00:01	00:01	10	874	1,534
iir 4 64	00:06	00:01	00:01	00:02	10	2,611	8,101
jpeg2000	00:58	00:16	00:23	00:33	40	12,822	36,381
latnrm 32 64	00:05	00:01	00:01	00:01	12	1,578	2,964
mult 10 10	00:10	00:01	00:01	00:01	6	636	1,521
spectral	00:10	00:01	00:07	00:22	33	8,674	21,232
average	00:22	00:03	00:06	00:09	21	4,948	12,222

Table 5.1: Statistics of Homogeneous ILP-based Task-Level Parallelization Approach for the Arm11MPCore Platform.

of task-level parallelism for the Arm11MPCore platform. Statistics for the other architectures are comparable and therefore omitted. The execution time is divided into the necessary preprocessing steps (including code optimization, execution time estimation and data-dependency analysis) and the extraction algorithm itself. For the latter one, the times for extracting up to two, three, and four tasks are given. CPLEX [IBM13] was used as ILP solver. As shown in Table 5.1, most time is required to execute the preprocessing steps (22 seconds on average). The results for two, three, and four tasks could be extracted in 3, 6, and 9 seconds in the average case, respectively. Many benchmarks could be processed in less than a second which shows that the complex ILP system can be solved efficiently, even though ILP is NP-complete in the general case. This is only possible due to the hierarchical segmentation provided by the Augmented Hierarchical Task Graph.

The ILP statistics shown on the right-hand side of Table 5.1 are based on the solutions running four tasks in parallel. The first column (#ILPs) shows the number of created and solved ILP systems while the second and third ones present the number of created variables (#Var) and constraints (#Constr) summed up over all ILP systems. For the *compress* benchmark, for example, 40 ILP systems were created containing all in all nearly 10k decision variables and 23k constraints. The whole benchmark was parallelized in 21 seconds and the ILPs were solved in less than a second. The same behavior could also be observed for the average case, where 21 ILP systems were solved in 9 seconds which also highlights the efficiency of the proposed approach.

[2]Measured on an AMD Opteron core running at 2.4 GHz

5.3 ILP-based Pipeline Parallelization Approach

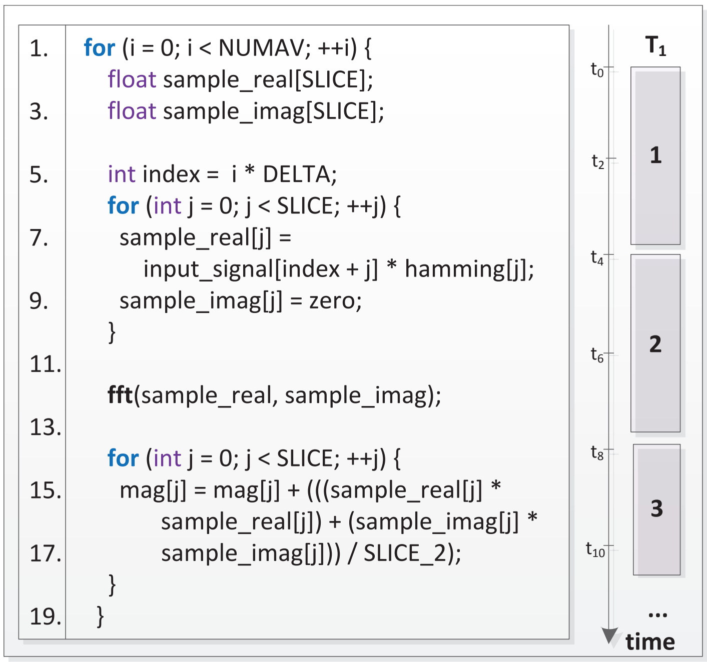
The evaluation of the task-level parallelization technique presented in the previous section has shown that this kind of parallelism is well applicable for embedded applications from multiple application domains. Reasonable speedups could be extracted for most of the considered benchmarks. However, the approach was not able to parallelize some of them efficiently, like the *filterbank*, *jpeg2000*, and *spectral* benchmarks. The reason is that many embedded applications, especially in the domain of networking services, voice- and image processing as well as multimedia tasks like video decoding, are structured in a pipelined manner. All these applications have in common that most of their parallelism is hidden in loops containing different pipelining-based jobs. Several filters are, for example, applied to a block of a source image and each filter depends on the previous one. This prevents the extraction of task-level parallelism. Often, some of these filters are also responsible for dependencies between different processed blocks. This also prohibits ordinary data-level parallelism like DoAll parallelism which just executes all loop iterations in parallel. In contrast, pipeline parallelism can often lead to efficient parallel solutions in such cases. Since ILP was successfully employed to extract task-level parallelism in the previous section, it is also used here to extract pipeline parallelism from sequentially written applications.

The rest of this section is structured as follows: First, Section 5.3.1 explains pipeline parallelism in more detail and introduces its key properties with a motivating example. Afterwards, Section 5.3.2 presents the Program Dependence Graph (PDG) which is used as intermediate representation to extract pipeline parallelism. The integration of the pipeline parallelization technique into the global parallelization approach is discussed in Section 5.3.3 before the model used is explained in Section 5.3.4. Details of the ILP-based extraction technique are then presented in Section 5.3.5 before the efficiency of the proposed approach is evaluated in Section 5.3.6.

5.3.1 Motivating Example for Pipeline Parallelism

The extraction of coarse-grained task-level parallelism is an efficient technique to parallelize large independent blocks of an application such as function calls which can be executed concurrently. However, those approaches lack the possibility to extract parallelism from loops, especially from those with loop-carried dependencies. Therefore, this section introduces pipeline parallelism which can be extracted from flat or nested loops of sequentially written applications. The *spectral* benchmark, which is part of the UTDSP benchmark suite [Lee13], is used as a real-world motivating example. This application calculates the power spectrum of an input sample of speech. It was chosen as an example since it has a representative structure for pipeline-based embedded applications without being too complex.

The example code shown in Figures 5.15 - 5.17 represents the main computation loop of the *spectral* benchmark. On the left-hand side of each figure, the applica-



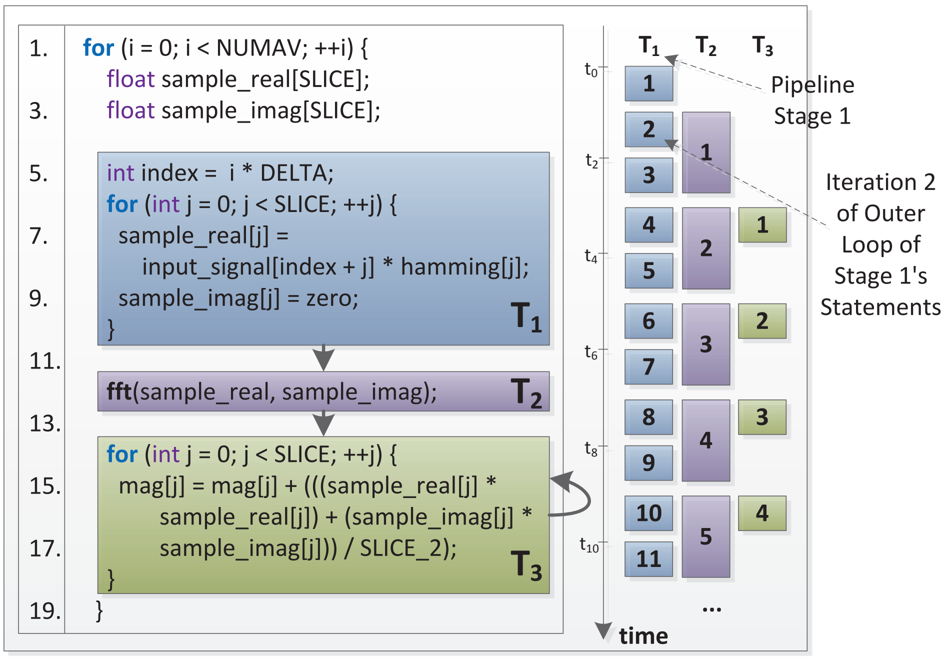
```
1.   for (i = 0; i < NUMAV; ++i) {
       float sample_real[SLICE];
3.     float sample_imag[SLICE];

5.     int index =  i * DELTA;
       for (int j = 0; j < SLICE; ++j) {
7.       sample_real[j] =
             input_signal[index + j] * hamming[j];
9.       sample_imag[j] = zero;
       }
11.
       fft(sample_real, sample_imag);
13.
       for (int j = 0; j < SLICE; ++j) {
15.      mag[j] = mag[j] + (((sample_real[j] *
              sample_real[j]) + (sample_imag[j] *
17.           sample_imag[j])) / SLICE_2);
       }
19.  }
```

Figure 5.15: Spectral Benchmark [Lee13]: Sequential Execution

tion's source code and the way how the statements are partitioned into concurrently executed tasks are shown. The right-hand side shows the time at which the iterations of the tasks are executed. In this example, it is assumed that the execution of the first inner loop and the calculation of the index (lines 5-10) requires one time slot, the execution of one FFT call (line 12) takes two time slots and the execution of the second inner loop (lines 14-18) needs one time slot to be completed. Thus, one iteration of the whole outer loop takes four time slots like shown on the right-hand side of Figure 5.15, representing the sequential execution of the application. The second inner loop starting in line 14 reads elements of the *mag* array which were written in the previous iteration of the outer loop (starting at line 1). This creates a loop-carried dependency between different iterations of the outer loop which prevents parallelization approaches from executing the whole outer loop in parallel. Even though many types of parallelism are useless in such a situation, pipeline parallelism can successfully be applied here by splitting a loop horizontally and vertically into concurrently executed tasks. While horizontal splits divide the statements of the loop body into disjunctive pipeline stages which are executed in a pipelined manner, vertical splits further partition the different loop iterations of the created stages into additional sub-tasks.

An example performing horizontal splits applied to the *spectral* application is depicted in Figure 5.16. Here, the body of the outer loop is divided into three pipeline stages T_1, T_2, and T_3. The benefit of such a parallelization is that each pipeline stage can start the next iteration of the outer loop, executing its assigned statements as soon as it has communicated its result to the next pipeline stage



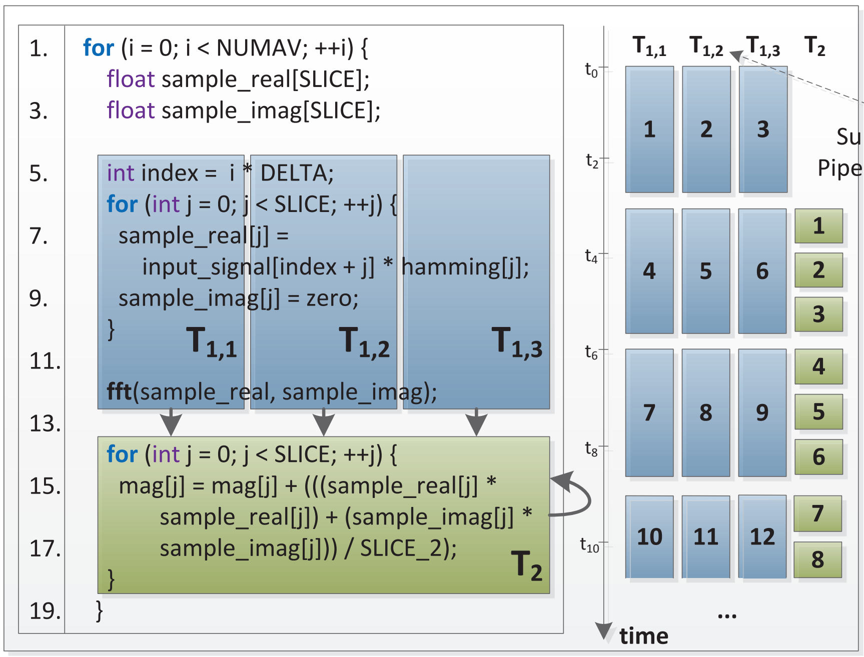
```
1.  for (i = 0; i < NUMAV; ++i) {
      float sample_real[SLICE];
3.    float sample_imag[SLICE];

5.    int index =  i * DELTA;
      for (int j = 0; j < SLICE; ++j) {
7.     sample_real[j] =
          input_signal[index + j] * hamming[j];
9.     sample_imag[j] = zero;
      }
11.
      fft(sample_real, sample_imag);
13.
      for (int j = 0; j < SLICE; ++j) {
15.    mag[j] = mag[j] + (((sample_real[j] *
              sample_real[j]) + (sample_imag[j] *
17.           sample_imag[j])) / SLICE_2);
      }
19. }
```

Figure 5.16: Spectral Benchmark [Lee13]: Horizontal Loop Splits

waiting for its output. In this way, a pipeline of calculations is created whose execution time is shown on the right-hand side of Figure 5.16. As soon as the first iteration of pipeline stage T_1 has completed its work, the required data is sent to pipeline stage T_2. Now, T_1's second and third iteration can be executed in parallel to the first iteration of pipeline stage T_2. After completion of its first iteration, T_2 sends its data to stage T_3 so that all three pipeline stages execute their work in an interleaved, pipelined manner.

As can be seen, this example is not well balanced since pipeline stage T_3 has to wait for data of stage T_2 after each iteration. To circumvent this problem, different loop iterations of the outer loop (line 1) of the created pipeline stages may be executed in parallel if data dependencies do not prevent parallel execution. Such sub-tasks of the extracted pipeline stages are generated by so-called vertical splits. The extracted parallelism shown in Figure 5.17 combines horizontal and vertical splits and is much more efficient than the solution which only generates pipelined tasks as shown in Figure 5.16. Pipeline stages T_1 and T_2 of the example shown in Figure 5.16 are combined into one pipeline stage (T_1) to provide a more computationally intensive pipeline stage. This stage is now vertically divided into three sub-tasks $T_{1,1}$, $T_{1,2}$, and $T_{1,3}$. These tasks execute the first, second and third iteration of the outer loop (line 1) of the statements assigned to this pipeline stage in parallel. As soon as all sub-tasks have communicated their data to the consuming pipeline stage T_2, iterations four, five and six are executed in parallel to the first three iterations of pipeline stage T_2 and so on. By combining both kinds of splits, the workload is well balanced.



```
for (i = 0; i < NUMAV; ++i) {
  float sample_real[SLICE];
  float sample_imag[SLICE];

  int index =  i * DELTA;
  for (int j = 0; j < SLICE; ++j) {
    sample_real[j] =
      input_signal[index + j] * hamming[j];
    sample_imag[j] = zero;
  }

  fft(sample_real, sample_imag);

  for (int j = 0; j < SLICE; ++j) {
    mag[j] = mag[j] + (((sample_real[j] *
        sample_real[j]) + (sample_imag[j] *
        sample_imag[j])) / SLICE_2);
  }
}
```

Figure 5.17: Spectral Benchmark [Lee13]: Horizontal and Vertical Loop Splits

By combining horizontal and vertical splits, the pipeline parallelization approach presented in this section is able to extract efficient parallelism from loops, even if loop-carried dependencies prevent traditional data-level parallelization techniques. The ILP-based approach provides both kinds of splits simultaneously so that it will not end up in a local optimum which may be the case if the vertical splits would rely on horizontal splits extracted in a separate phase. The example also shows that it is important to take the execution time of the tasks into account to create a well-balanced task structure.

5.3.2 Augmented Program Dependence Graph

Besides all advantages of the Augmented Hierarchical Task Graph (AHTG), it has become evident that the AHTG is not the best intermediate representation to extract parallelism from loops – especially from nested ones. As described in Chapter 4.1, the AHTG hides data flow edges pointing in the opposite direction of the control flow (back-edges). This is one of the key aspects used to separate the different hierarchical levels as exploited by the global parallelization approach of the presented framework. However, these back-edges are important for loop-level and pipeline parallelism, since, among others, they represent data which is communicated between different iterations of the loop(-nest) to be parallelized. In addition, an implicit synchronization barrier is assumed at the end of each hierarchical node, which is not a problem for task-level parallelism. Contrarily, it drastically limits the speedup obtainable by pipeline parallelism.

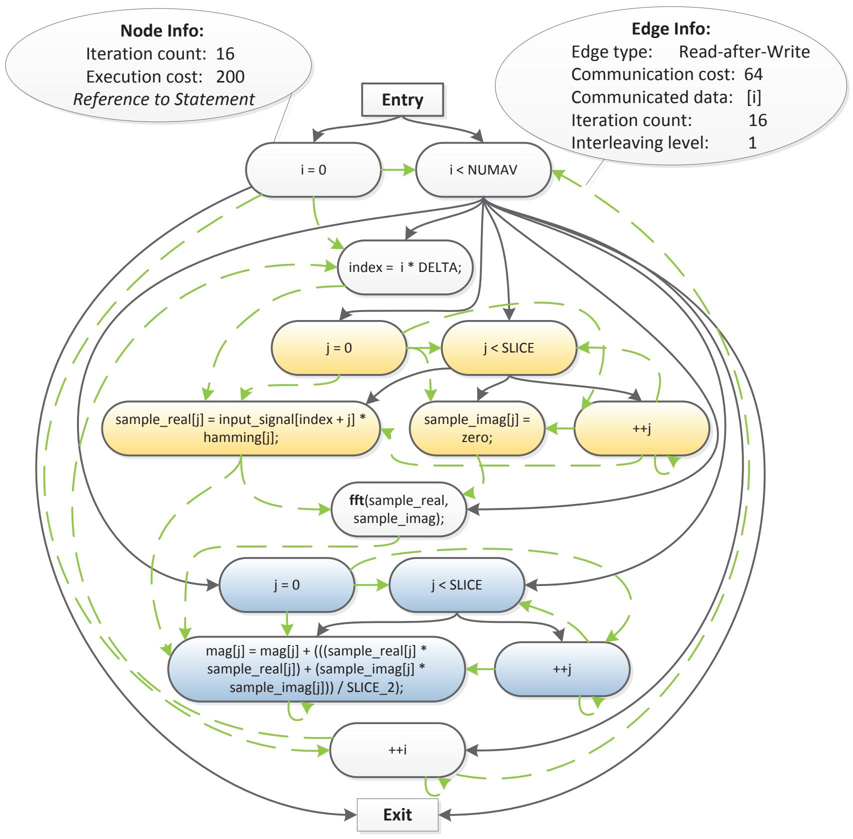


Figure 5.18: Program Dependence Graph Example - Spectral Benchmark

Therefore, a so-called Program Dependence Graph (PDG) [FOW87] is extracted and augmented with cost information for a loop(-nest) as soon as a hierarchical node representing a loop is reached in the parallelization process. The employed augmented PDG combines control-[3] and data-flow dependencies in one graph representation like also done by the AHTG. In contrast to the AHTG, the augmented PDG is a flat graph without hierarchical levels and communication redirection. Each statement of the loop to be parallelized is represented by a node in the PDG and data- as well as control-flow edges are directly added between the nodes.

An example of an augmented PDG is given in Figure 5.18. The graph represents the nested main computational loop of the *spectral* benchmark from Figure 5.15. Obviously, the graph contains one *entry* node, one *exit* node and several other nodes representing statements of the application. The nodes are connected by directed edges describing dependencies that have to be taken into account if the nodes are mapped to concurrently executed tasks. Solid black edges represent control flow dependencies while dashed green edges visualize data dependencies. In order to extract efficient parallelism from sequential applications, the created tasks have to be balanced as shown by the first presented parallelization approach. For this reason,

[3]Denoting that the execution of a statement must precede the execution of another statement.

each node of the PDG is also augmented with the iteration count and execution costs of the statement represented by the node (cf. *Node Info* in Figure 5.18). It is also essential to have information about the communication costs which have to be taken into account if the statements of the nodes are executed in separate tasks. Therefore, the edge type, communication costs, the communicated data, the iteration count as well as an interleaving level – describing the minimal amount of loop iterations which can be executed before the data is consumed at the target node – are annotated to the data dependence edges (cf. *Edge Info* in Figure 5.18). Additional information like the estimated energy consumption or different objective values depending on the executing processing unit of a heterogeneous architecture will be added by other approaches presented later in this thesis.

Since the PDG was not conceived by the author of this thesis, the author would like to refer to [FOW87] for more details and techniques how to extract the graph from a given application's source code.

5.3.3 Integration into the Global Parallelization Approach

The integration of the ILP-based pipeline parallelization approach optimized for homogeneous MPSoCs into the global parallelization framework (cf. Section 4.2) is shown in Algorithm 6.

Algorithm 6 Pseudo Code of the ILP-based Pipeline Parallelization Approach

```
1: // Called bottom-up hierarchically by Algorithm 4 in line 18 on page 56
2: function ExtractHomPipeline(Node $n$, Platform $pf$, int $maxTasks$)
3:     // This function is only applicable to loops.
4:     $solutions \leftarrow \emptyset$
5:     if not $n.getStmt().isLoopStmt()$ then
6:         return $solutions$
7:     end if
8:     // Create an augmented PDG for the loop(-nest).
9:     $loopPDG \leftarrow$ ConstructPDG($n.getStmt()$)
10:    // Extract pipeline parallelism from the loop's PDG.
11:    $i \leftarrow maxTasks$
12:    while $i >= 2$ do
13:        $result \leftarrow$ HomILPPipelineParallelizer($loopPDG, pf, i$)
14:        $solutions \leftarrow solutions \cup \{result\}$
15:        $i \leftarrow$ NumberOfTasks($result$) $- 1$
16:    end while
17:    return $solutions$
18: end function
```

The function ExtractHomPipeline is executed by the global parallelization algorithm (cf. Algorithm 4) as soon as all child nodes deeper in the hierarchy are processed. As arguments, the function expects the node n to be parallelized, platform specific information pf containing, e.g., the performance characteristics and the number of available processing units, and an upper bound of extractable tasks *maxTasks*. The variable *maxTasks* is set to the number of available processing units of the targeted architecture by default and can be customized by the application designer. In this way, the parallelization process does not generate more tasks than processing units are available by default.

Pipeline parallelism can, in general, be extracted only from nested or flat loops of sequentially written applications. Therefore, the algorithm first determines whether the currently processed node represents a loop statement in lines 5-7. If this is the case, it creates an augmented PDG containing only nodes for the statements which are part of the (nested) loop, like explained in Section 5.3.2. If the statement is a non-loop statement, the algorithm returns an empty solution set in line 6 so that other parallelization approaches should be used to extract parallelism for node n.

ILP solvers return the best solution as their only result. Since the child solutions should be re-combined with new parallelism on the parent hierarchical level, several solution candidates with a different number of concurrently executed tasks should be extracted for each node. Therefore, the function HomILPPipelineParallelizer is called multiple times to extract solutions with, e.g., two, three, four, up to *maxTasks* concurrently executed tasks in line 13. All generated solution candidates are finally collected in line 14 and returned as solution candidates for the processed loop node n in line 17. The pipeline-based solution candidates extracted for node n can further be combined with solution candidates from other parallelization techniques in the global parallelization approach. Afterwards, the extraction step is continued with nodes on the same and later parent hierarchical levels until the root node of the application's AHTG is reached.

5.3.4 Parallelization Model

The function HomILPPipelineParallelizer of Algorithm 6 is called for each loop in isolation with a given augmented PDG containing only those nodes which are part of the loop to be parallelized. Furthermore, an upper bound of extractable tasks and some platform-specific information are given as arguments in line 13 of Algorithm 5. The pipeline parallelization approach tries to split the statements of the loop's body horizontally into disjunctive pipeline stages which are executed in a pipelined manner. Vertical splits can also be applied to each pipeline stage which move different iterations of a stage into concurrently executed tasks (cf. example in Section 5.3.1).

The ILP-based pipeline parallelization approach is able to balance the extracted tasks fully automatically. Therefore, a high-level cost model is used to evaluate the performance of the possible parallel solution candidates. This model considers dependencies between the extracted pipeline stages and also between their different iterations. To accomplish this, the different iterations of the loop are virtually unrolled for evaluation purposes within the employed model[4]. Based on this unrolled augmented PDG, the most expensive execution path (*critical path*), starting from

[4]This iteration-unrolling is only done in the employed model and, of course, not in the application's source code.

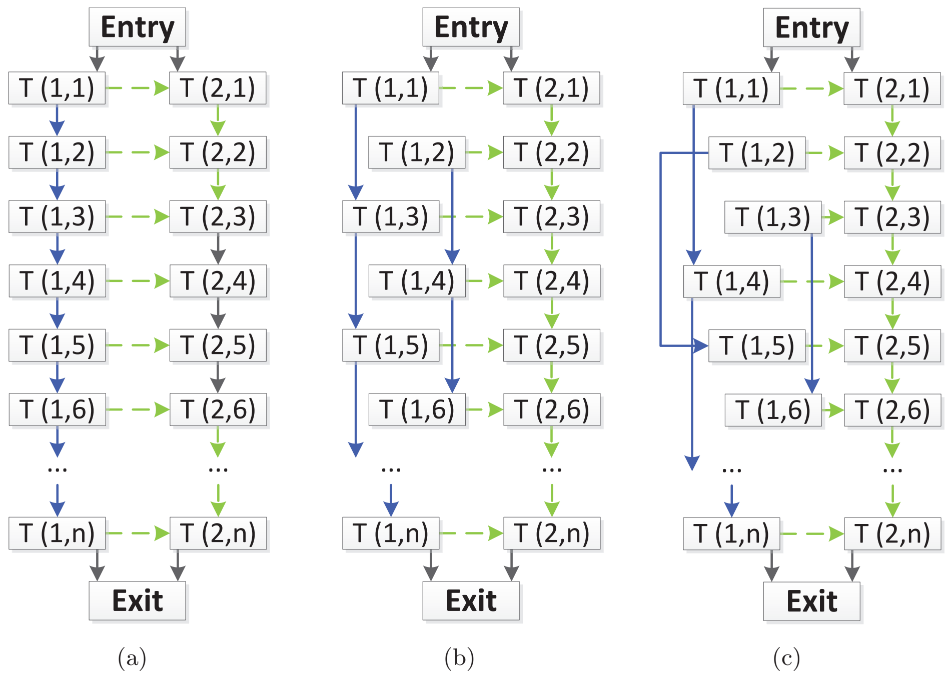


Figure 5.19: Loop Dependencies for Vertical Splits

the *entry* to the *exit* node of the loop's PDG is minimized by splitting the loop horizontally and vertically into different tasks.

An example for such an unrolled PDG containing two extracted pipeline stages T_1 and T_2 is depicted in Figure 5.19(a-c). $T(x, y)$ represents iteration y of pipeline stage x. The example is based on the *spectral* benchmark shown in Figure 5.17. Pipeline stage 2 consumes data of stage 1 and a loop-carried data dependency exists between the different iterations of pipeline stage 2 like denoted by the dashed green edges. In contrast, pipeline stage 1 is free of loop-carried data dependencies. Besides loop-carried data dependencies, the amount of vertical splits also adds virtual dependencies between the different iterations of the pipeline stages' execution since iterations belonging to the same task can only be executed one after another. If, e.g., a pipeline stage is not divided into several concurrently executed tasks, all iterations are processed sequentially by one task. In contrast, if a pipeline stage is, e.g., split once, iterations 1 and 2 as well as 3 and 4 etc. can be executed in parallel. Since iterations 1, 3, 5 etc. are mapped to the same sub-task, iteration 3 depends on iteration 1 while iteration 5 depends on iteration 3 and so on.

Figure 5.19(a) shows the different iterations of the two pipeline stages without vertical splits. All iterations of pipeline stage 2 have to be executed sequentially due to the loop-carried data dependency with an interleaving level of 1. Even though no loop-carried data dependencies exist for pipeline stage 1, a dependence edge has to be inserted between all iterations of stage 1 since all of them are mapped to the same sub-task due to missing vertical splits (denoted by the solid blue arrows).

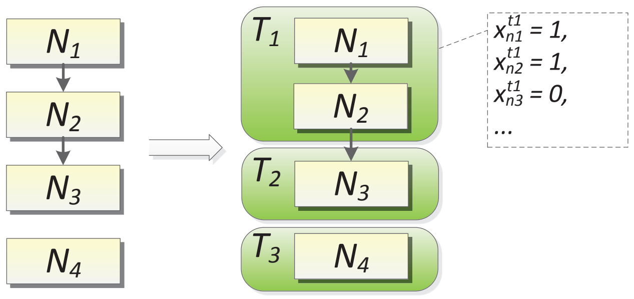


Figure 5.20: Horizontal Split Constraint (Pipeline Stage Extraction)

Figure 5.19(b) shows how the dependencies change if pipeline stage 1 is vertically split once. Iterations $\{1, 3, 5, ..\}$ and $\{2, 4, 6, ..\}$ of pipeline stage 1 can be executed in parallel in that case. Further parallel execution can be achieved if pipeline stage one is split twice into three concurrently executed sub-tasks like depicted in Figure 5.19(c). Here, the iterations are grouped into tasks executing iterations $\{1, 4, 7, ..\}$, $\{2, 5, 8, ..\}$, and $\{3, 6, 9, ..\}$ in parallel. Of course, it does not make sense to split pipeline stage 2 into concurrently executed tasks since the loop-carried data dependency would sequentialize their execution.

All cases and their corresponding dependencies, shown in Figure 5.19(a-c), have to be taken into account by the ILP-based parallelization technique presented in the following section. In addition, the dashed green arrows of Figure 5.19, describing data dependencies, depend on the node-to-task mapping. Thus, the dependencies between the tasks may also change if one statement is moved from one pipeline stage to another one.

5.3.5 ILP-based Parallelization Approach

This section defines the ILP formulations for the pipeline-based parallelization approach described above. The approach is executed for each loop in isolation and covers four main goals:

I) Extract different pipeline stages by mapping statements of the loop's body into disjunctive stages (cf. Figure 5.16 on page 86 from the motivating example).

II) Divide pipeline stages into sub-tasks which execute different iterations of the stages in parallel (see Figure 5.17 on page 87 from the motivating example).

III) Keep track of dependencies which may change if statements are moved from one pipeline stage to another one or if iterations of pipeline stages are mapped to different sub-tasks.

IV) Minimize execution costs by considering task creation, communication, and task execution costs.

In the following, decision variables are written in lower case letters, sets start with a capital letter and constants contain exclusively capital letters. Indices n

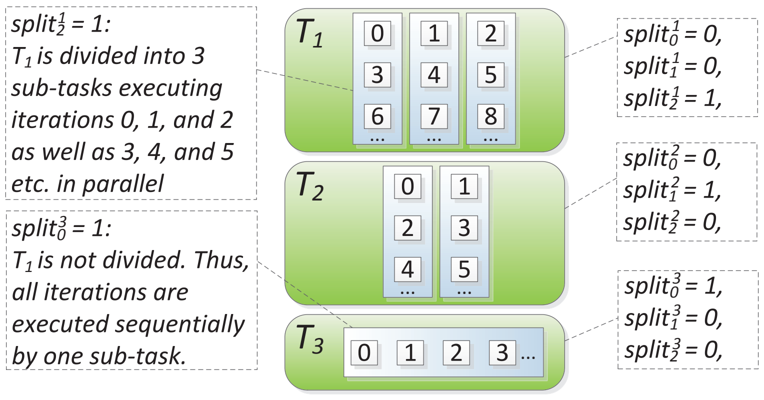

split$_2^1$ = 1:
T$_1$ is divided into 3 sub-tasks executing iterations 0, 1, and 2 as well as 3, 4, and 5 etc. in parallel
split$_0^3$ = 1:
T$_1$ is not divided. Thus, all iterations are executed sequentially by one sub-task.
T$_1$
0 1 2 3 4 5 6 7 8 ...
T$_2$
0 1 2 3 4 5 ...
T$_3$
0 1 2 3 ...
split$_0^1$ = 0, split$_1^1$ = 0, split$_2^1$ = 1,
split$_0^2$ = 0, split$_1^2$ = 1, split$_2^2$ = 0,
split$_0^3$ = 1, split$_1^3$ = 0, split$_2^3$ = 0,

Figure 5.21: Vertical Split Constraint (Split Pipeline Iterations)

and o are used for nodes of the PDG, i and j are used for iterations of the loop to be parallelized, t and u represent indices for pipeline stages while s is used for concurrently executed sub-tasks of a pipeline stage. Graphical representations for most equations are also given in Figures 5.20 - 5.24.

5.3.5.1 Horizontal Split Constraint (Pipeline Stage Extraction)

Goal (I) of the ILP-based pipeline parallelization approach for homogeneous MP-SoCs is a mapping of PDG nodes to pipeline stages. This is the first method applied by this approach to extract parallelism and is comparable to the horizontal splits presented in the motivating example. This is ensured by decision variable x_n^t like defined in Equation 5.18 and visualized in Figure 5.20. It evaluates to 1 if node n is mapped to pipeline stage t.

$$x_n^t = \begin{cases} 1, & \text{if node } n \text{ is mapped to pipeline stage } t \\ 0, & \text{otherwise} \end{cases} \tag{5.18}$$

The constraint defined in Equation 5.19 ensures that every child node (representing statements of the loop to be parallelized) is mapped to exactly one pipeline stage.

$$\forall n \in Nodes: \sum_{t \in Stages} x_n^t = 1 \tag{5.19}$$

5.3.5.2 Vertical Split Constraint (Split Pipeline Iterations)

Besides horizontal splits, further parallelism can be extracted by vertical splits like shown in the motivating example. These vertical splits allocate the different iterations of a pipeline stage in concurrently executed tasks (Goal (II)). This is expressed by decision variable $split_s^t$ which has the value of 1 if pipeline stage t is split s times. In the example of Figure 5.21, $split_2^1$ has the value of 1 which means that pipeline stage t_1 is divided twice into 3 concurrently executed sub-tasks. Hence, iterations

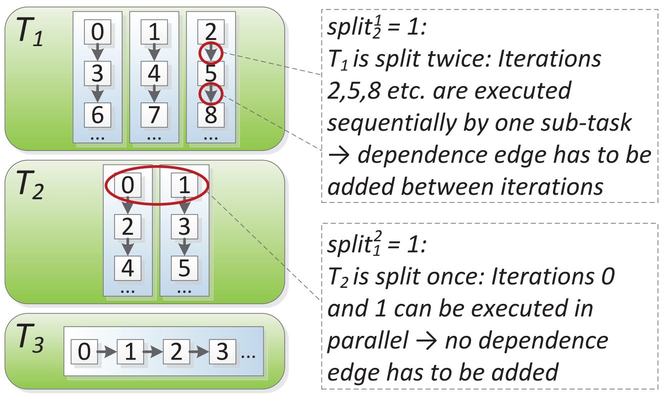

T1
0 3 6 ...
1 4 7 ...
2 5 8 ...
T2
0 2 4 ...
1 3 5 ...
T3
0 1 2 3 ...
split$^1_2$ = 1:
T1 is split twice: Iterations 2,5,8 etc. are executed sequentially by one sub-task → dependence edge has to be added between iterations
split$^2_1$ = 1:
T2 is split once: Iterations 0 and 1 can be executed in parallel → no dependence edge has to be added

Figure 5.22: Stage Split Dependencies

0, 1, and 2 as well as iterations 3, 4, and 5, etc. are executed in parallel. In contrast, $split^3_0$ is also set to 1 which means that pipeline stage t_3 is not divided into sub-tasks so that all iterations of pipeline stage 3 are executed sequentially by one task. Equation 5.20 defines decision variable $split^t_s$.

$$split^t_s = \begin{cases} 1, & \text{if task } t \text{ is split } s \text{ times} \\ 0, & \text{otherwise} \end{cases} \tag{5.20}$$

The ILP solver has to determine the best amount of splits for each pipeline stage t used, like ensured by Equation 5.21.

$$\forall t \in Stages : \sum_{s \in \{0..MAXTASKS\}} split^t_s = 1 \tag{5.21}$$

5.3.5.3 Predecessor Constraint

To minimize the critical or most expensive path from the *entry* to the *exit* node of the PDG, the ILP formulation has to be extended by path information. As described in Section 5.3.4, the path costs are estimated by virtually unrolling the iterations of the loop. As a consequence, predecessor variables have to be created for each iteration combination of two pipeline stages (Goal (III)). Equation 5.22 defines decision variable $pred^{t,u}_{i,j}$ which is added for each pipeline stage t in iteration i and stage u in iteration j.

$$pred^{t,u}_{i,j} = \begin{cases} 1, & \text{if stage } t \text{ in iteration } i \text{ is a direct} \\ & \text{predecessor of stage } u \text{ in iteration } j \\ 0, & \text{otherwise} \end{cases} \tag{5.22}$$

Two situations may lead to a predecessor relationship between two pipeline stages in their different iterations. The first one is based on the amount of vertical splits per pipeline stage (Stage Split Dependencies) while the second one depends on data and control flow dependencies (Data and Control Flow Dependencies) caused by horizontal splits. Therefore, both kinds of dependencies are defined in the following.

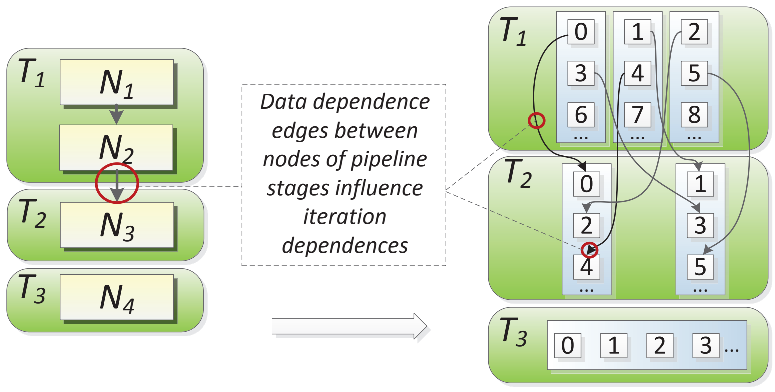

T1
N1
N2
T2
N3
T3
N4
Data dependence edges between nodes of pipeline stages influence iteration dependences
T1
0 1 2
3 4 5
6 7 8
T2
0 1
2 3
4 5
T3
0 1 2 3

Figure 5.23: Data and Control Flow Dependencies

Stage Split Dependencies: Figure 5.22 shows the same example of allocated pipeline stages and their iterations to sub-tasks as presented in Figure 5.21. Here, new dependence edges are added between some iterations of the pipeline stages since all iterations allocated to the same sub-task are executed one after another. Since pipeline stage t_1 is divided twice into three sub-tasks in Figure 5.22, iterations 0, 1, and 2 as well as 3, 4, and 5 etc. can be executed concurrently – if no data dependencies prevent the parallel execution. Thus, no split dependence edges have to be added between iterations 0, 1, and 2. However, iterations 0 and 3 as well as 1 and 4 are executed by the same sub-task. Since each sub-task can execute only one iteration at a time, iteration 0 of pipeline stage t_1 must be executed before iteration 3 can be started. Therefore, a dependence edge is added between all iterations mapped to the same sub-task depending on how often the pipeline stages are split.

This is ensured by Equation 5.23 adding constraints for all iteration combinations of pipeline stage t which do not exceed the upper boundary of the maximum extractable tasks $MAXTASKS$. Depending on how often pipeline stage t is split ($split^t_s$), iteration i of stage t is a predecessor of iteration j of stage t. The number of loop iterations NI can be determined by static loop analyzers, like, e.g. [Cor08], [LCF+09], or the analyzer integrated in the employed ICD-C high-level IR [Inf13].

$$\begin{gathered}\forall t \in Stages : \forall i \in \{0, .., NI\text{-}1\} : \\ \forall j \in \{i+1, .., NI\text{-}1\} : (j - i \leq MAXTASKS) : \\ pred^{t,t}_{i,j} \geq split^t_{j-i-1}\end{gathered} \tag{5.23}$$

Data and Control Flow Dependencies: Data and control flow dependencies between pipeline stages form the second kind of dependency which may sequentialize the execution of the different pipeline stages' iterations. In contrast to the Stage Split Dependencies which are based on the $split^t_s$ variables, the data and control flow dependencies are based on the dependencies between the nodes mapped to the different pipeline stages. An example is shown in Figure 5.23. The node-to-pipeline stage mapping is shown on the left-hand side. As can be seen, a dependence edge between node n_2 and node n_3 exists, which requires communication between pipeline stage t_1 and pipeline stage t_2. This communication takes place at the end of each iteration. Each edge of the PDG contains additional information containing, e.g., the interleaving level denoting how many iterations can be executed between source and target node before the data is required. An interleaving level of 0 would, e.g., indicate that the data is consumed in the same iteration. An interleaving level of 1 would describe that the data is required in the next iteration, etc. The example on the right-hand side of Figure 5.23 assumes an interleaving level of 0 for the dependence edge. Dependence edges have to be added between the corresponding iterations of both pipeline stages, due to the edge between pipeline stage t_1 and pipeline stage t_2. Therefore, iteration 0 of pipeline stage t_1 is a predecessor of iteration 0 of stage t_2. In addition, iteration 1 of pipeline stage t_1 is a predecessor of iteration 1 of stage t_2. Such edges have to be added between all iterations of both stages[5].

The creation of these dependencies is ensured by Equation 5.24:

$$\begin{gathered}\forall t, u \in Stages : \forall i \in \{0, .., NI\text{-}1\} : \forall j \in \{i, .., NI\text{-}1\} : \\ \forall n, m \in Nodes : n \neq m : EDGE_{n,m,j-i} = 1 : \\ pred^{t,u}_{i,j} \geq x^t_n \wedge x^u_m \end{gathered} \tag{5.24}$$

The predecessor variable $pred^{t,u}_{i,j}$ is created for all possible pipeline stage and loop iteration combinations. In this way, for all combinations of nodes, it is checked if node n is part of stage t while node m has to be part of stage u. If this is true and a directed edge from n to m exists with an interleaving level of $j - i$, denoted by $EDGE_{n,m,j-i}$, pipeline stage u depends on t for the iterations i and j[6].

5.3.5.4 Execution Costs of Pipeline Stage Constraint

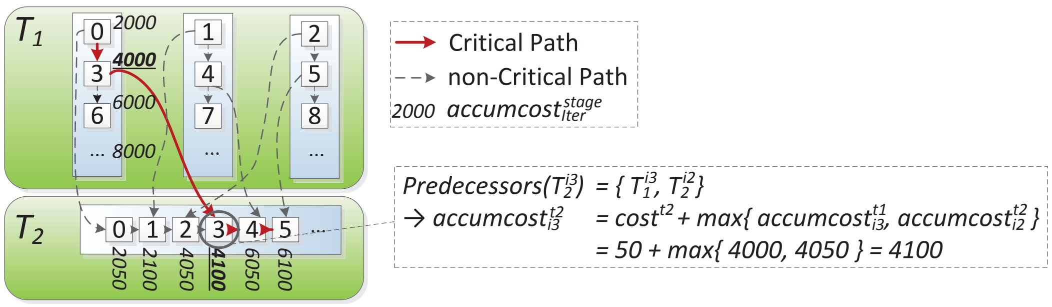
The predecessor relationship enables to describe paths with respect to dependencies. Since it is particularly important to take execution and communication costs into account to create well-balanced tasks, the augmented cost information of the PDG (cf. Section 5.3.2) has to be integrated into the ILP formulation. Therefore, the overall execution costs of each node are distributed in equal parts over the different iterations of the loop. This saves a couple of decision variables since the ILP does not have to distinguish between different execution costs of pipeline stages in different iterations.

$$\forall t \in Stages : cost^t \geq \sum_{n \in Nodes} x^t_n * (COST_n / NI) \tag{5.25}$$

Equation 5.25 sets the lower bound of the costs for one iteration of pipeline stage t to at least the sum of costs $COST_n$ of each node n, which is part of stage

[5]If the interleaving level of the dependence edge would be, e.g., 1, iteration 0 of t_1 would be a predecessor of iteration 1 of t_2, etc.

[6]The usage of the $\wedge$ operator and preconditions within ILP formulations is shown in the Appendix in Section A.2



T1
T2
0 1 2 3 4 5 6 7 8 ...
2000
4000
6000
8000
2050 2100 4050 4100 6050 6100
Critical Path
non-Critical Path
2000 accumcost^stage_Iter
Predecessors(T^i3_2) = { T^i3_1, T^i2_2 }
→ accumcost^t2_i3 = cost^t2 + max{ accumcost^t1_i3, accumcost^t2_i2 }
= 50 + max{ 4000, 4050 } = 4100

Figure 5.24: Path Cost Constraint

t, divided by the number of loop iterations. The variable $cost^t$ is also part of the objective function so that it is automatically minimized by the ILP solver if stage t is part of the critical path. The pipeline parallelization approach does not combine hierarchical solution candidates (as done by the task-level parallelization approach presented in the previous section) since it is hard to handle a different amount of inner tasks for different iterations of the loop. However, this limitation is based on a technical simplification and can of course be improved by future work.

5.3.5.5 Path Cost Constraint

Based on the knowledge of the execution costs of all pipeline stage iterations, it is now possible to describe the accumulated costs of the possible paths like depicted in Figure 5.24[7] (Goal (IV)). The order in which data is communicated between two iterations of the pipeline stages is not known at this time. Since the worst-case scenario has performed best for the presented ILP-based task-level parallelization approach, a comparable one should be used here as well. The employed model assumes that a task t has to wait for its data until all its predecessor tasks have communicated all data to their successor tasks, even if this data is not consumed by t. The ILP formulation of this worst-case scenario based path calculation is shown in Equation 5.26.

$$\forall t, u \in Stages : \forall i \in \{0, .., NI\text{-}1\} : \forall j \in \{i, .., NI\text{-}1\} : pred^{u,t}_{i,j} = 1 :$$
$$accumcost^t_j \geq cost^t + accumcost^u_i + commcost^u \tag{5.26}$$

Equation 5.26 ensures that the path costs $accumcost^t_j$ for pipeline stage t in iteration j are at least as large as the costs $cost^t$ for the execution of one iteration of stage t itself and the path costs of its most expensive predecessor $accumcost^u_i$, including all communication costs $commcost^u$ of pipeline stage u.

5.3.5.6 Maximum Number of Extracted Tasks Constraint

To give the user the possibility to limit the number of concurrently executed tasks, a new decision variable $stageused^t$ is introduced in Equation 5.27, which reflects

[7]To keep the example in the figure more comprehensive, communication costs are omitted.

whether pipeline stage t is used.

$$stageused^t = \begin{cases} 1, & \text{if pipeline stage } t \text{ is used} \\ 0, & \text{otherwise} \end{cases} \tag{5.27}$$

Pipeline stage t is used if it contains at least one node, which is defined by Equation 5.28.

$$\forall t \in Stages : \forall n \in Nodes : stageused^t \geq x_n^t \tag{5.28}$$

The number of concurrently executed tasks is equal to the sum of pipeline stages used (for each stage used, the variable *stageused*^{t} evaluates to 1, otherwise to 0), increased by the number of vertical splits.

$$numtasks \geq \sum_{t \in Stages} (stageused^t + \sum_{s \in \{0..MAXTASKS\}} s * split_s^t) \tag{5.29}$$

If, e.g., pipeline stage 1 is used and it is split 3 times, 4 tasks are created (see Equation 5.29). Equation 5.30 ensures that the number of created tasks does not exceed the given upper bound *MAXTASKS* of concurrently executed tasks.

$$MAXTASKS \geq numtasks \tag{5.30}$$

The ILP formulation can further be optimized here by excluding solutions with the same objective value, which only differ in the allocated sub-tasks (cf. [LMM+97] and Section 9.2). This could be achieved by adding constraints like $stageused^t \geq stageused^{t+1}$ and may be integrated into the system in the future.

5.3.5.7 Cycle-Free Constraint

To avoid deadlocks and paths with infinite costs, the approach has to take care that the paths are cycle-free. Therefore, all direct child nodes are topologically sorted by their dependencies and an ascending, unique id is generated for both, nodes and pipeline stages. W.l.o.g., the generated graph is cycle-free if the *taskid* of node n is greater or equal to the *taskid*s of all nodes o with a smaller *topsort* id. This is ensured by Equation 5.31.

$$\forall n, o \in Nodes : topsort_n \geq topsort_o : taskid_n \geq taskid_o \tag{5.31}$$

5.3.5.8 Objective Function

With all decision variables and constraints defined, it is now possible to describe the objective function. As mentioned before, the most expensive execution path from the *entry* to the *exit* node of the loop's PDG should be minimized. Hence, additional constraints which statically set the *entry* node to be a predecessor of all tasks are added. The *exit* node will be a successor of all tasks, respectively. With

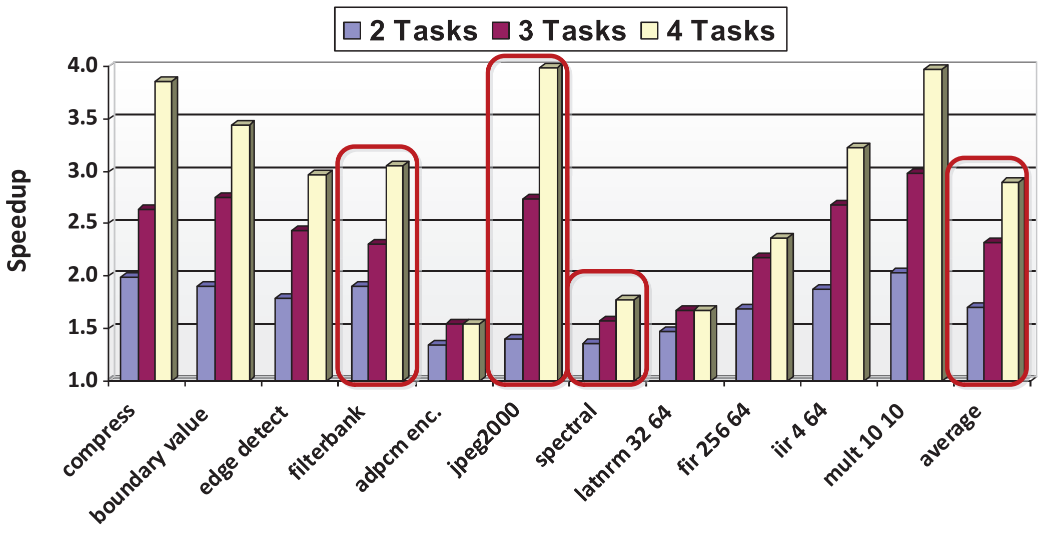


Figure 5.25: Speedup of Homogeneous Task-Level and Pipeline Parallelization Approach (Arm11MPCore Platform)

the help of the additional constraints, it is easy to create the objective function, like shown in Equation 5.32.

$$exectime = numtasks * TASKOVERHEAD + accumcost_{exit} \rightarrow \min \quad (5.32)$$

Since the creation of tasks also increases the execution time, a constant task creation overhead, multiplied by the number of created tasks, is added. This task creation overhead can be defined in the platform description together with a communication cost factor. By defining these platform-dependent parameters, the cost model of the ILP can easily be adapted to different architectures. The value of the objective function is equivalent to the execution time of the parallelized loop. It is hence returned together with the node-to-task mapping as the result of the parallelization step.

5.3.6 Experimental Results

Instead of presenting results for the pipeline parallelization approach only, a combination with the previously presented task-level parallelization approach of Section 5.2 is given here. Therefore, in addition to the approach itself, a combination of two approaches of the same framework presented in this thesis is also evaluated. Results for the pipeline parallelization approach in isolation can be found in the original publication in [CHM+11]. Some of the results may differ from the ones presented in [CHM+11] since the operating system, the simulators as well as the tool flow were changed between both measurements. To ensure comparability to the task-level parallelization approach evaluated in Section 5.2.5, the evaluation setup including the employed middleware (RTEMS operating system, etc.), the target platforms and also the set of benchmarks was chosen to be identical.

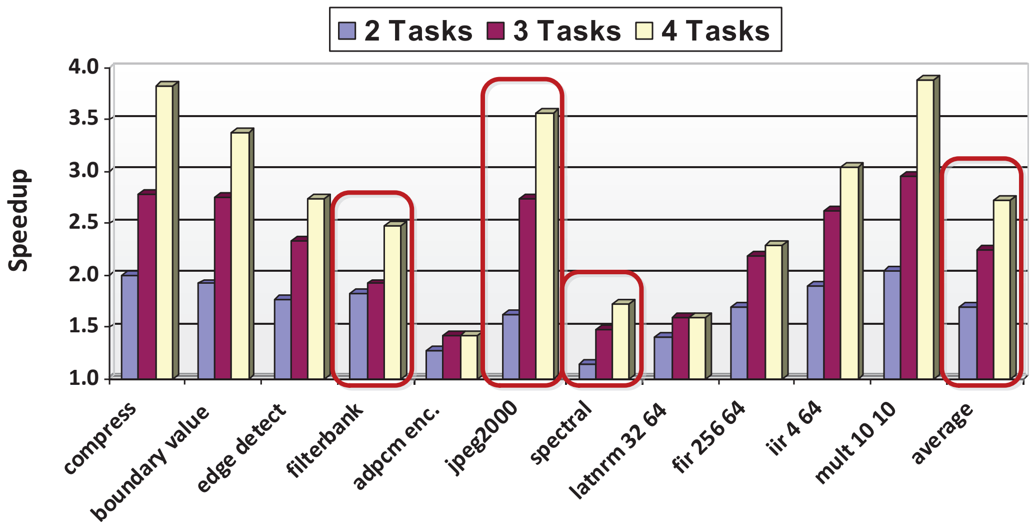


Figure 5.26: Speedup of Homogeneous Task-Level and Pipeline Parallelization Approach (Arm11QuadProcessor Platform)

The results for the three considered target platforms are presented in Figures 5.25 - 5.27. As can be seen, the speedup of most evaluated benchmarks scales well for the given amount of processing units (two, three and four) for all three target platforms. The benchmarks which profit most from the new pipeline parallelization approach are highlighted by red boxes and outperform the solutions extracted earlier by the task-level parallelization approach. More than one quarter of the evaluated benchmarks (3 of 11) and also the average speedup were drastically improved. The *jpeg2000* encoder, for example, reached a speedup of 1.4×, 2.8×, and 3.9× for two, three, and four concurrently executed tasks, respectively, for the Arm11MPCore platform as shown in Figure 5.25. In contrast, the task-level parallelization approach only reached an acceleration of around 1.1× for two up to four cores. For the two other platforms, speedups of 1.6×, 2.7×, and 3.6× (for the Arm11QuadProcessor) as well as 1.3×, 2.3×, and 3.2× (for MPARM) could be observed, respectively. Here, the speedups also scale well, even though the performance increase is less impressive than for the Arm11MPCore platform. One reason is that both other platforms miss a cache coherency unit so that the level one cache is not able to cache shared data which makes communication more expensive. Nevertheless, the speedups also scale well for the number of available cores for these platforms.

Besides the *jpeg2000 encoder*, the *filterbank* and *spectral* benchmarks also profit from the presented pipeline parallelization approach. However, the speedups reached for two, three, and four cores for the *spectral* benchmark are 1.4×, 1.6×, and 1.8× for the Arm11MPCore platform, respectively. This is also a higher performance increase than observed by the task-level parallelization approach (1.4×, 1.4×, and 1.5×) but is far away from the theoretical speedup limit provided by the target platform. The reason is that the parallelizable parts of the application are less computational

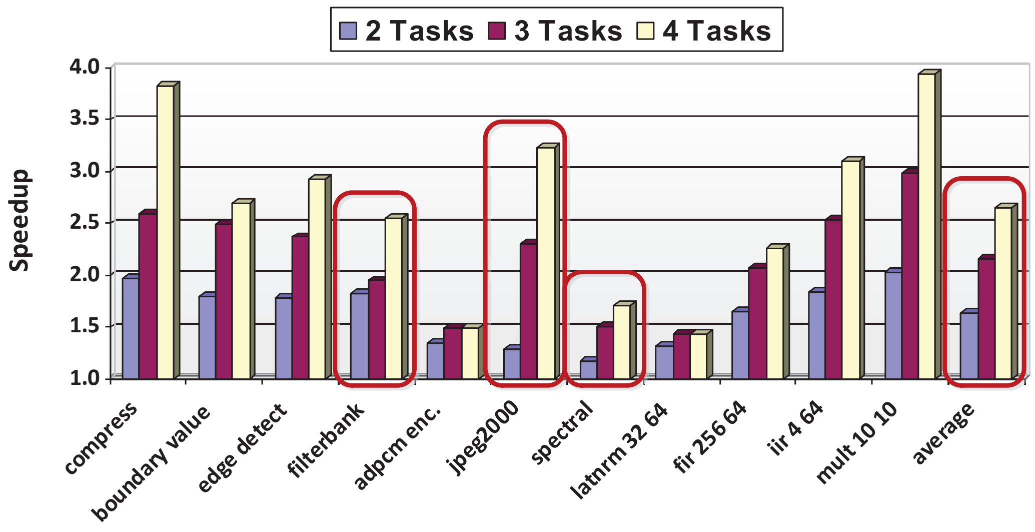


Figure 5.27: Speedup of Homogeneous Task-Level and Pipeline Parallelization Approach (MPARM Platform)

intensive and also require more communication. Thus, not all benchmarks can reach speedups between 3×-4× for a four core platform. The *spectral* benchmark also shows another interesting pattern. While the solutions for the four and three concurrently executed tasks were extracted by the pipeline parallelization approach, the version with two concurrently executed tasks was generated by the task-level parallelization approach. This also shows that a combination of both approaches is beneficial. The two core version of the *edge detect* benchmark could also profit from this combination. The loops of the function applying the filters (cf. Figure 5.1 for more details) were parallelized by the pipeline parallelization approach while concurrently executed function calls were extracted by the task-level parallelization approach. It should be mentioned here that the pipeline parallelization approach is also able to extract simple DoAll parallelism which executes independent iterations of a loop in parallel. This is just a corner case of pipeline parallelism with one replicated pipeline stage including all statements of the loop.

For the Arm11MPCore platform, an average speedup of 1.7×, 2.3×, and 2.9× could be observed, respectively. In contrast, speedups of 1.7×, 2.1×, and 2.6× could be reached with the task-level parallelization approach in isolation. This also shows that a combination of both approaches increases the performance. For the other two platforms, an average speedup of 1.7×, 2.3×, and 2.7× (for the Arm11QuadProcessor) as well as 1.6×, 2.2×, and 2.7× (for the MPARM) could be reached, respectively. The speedups for most evaluated benchmarks are further increased by more complex input samples as evaluated in [CHM+11]. However, the original input files of the application were used as input here.

With respect to the kind of extracted parallelism, it should be mentioned that the *spectral* benchmark was parallelized by the pipeline parallelization approach as

	Execution Times of Approach in MM:SS[8]				ILP Statistics (for 4 Tasks)		
Benchmark	Preproc.	2 Tasks	3 Tasks	4 Tasks	#ILPs	#Var	#Constr
adpcm enc.	00:09	00:02	00:03	00:04	39	7,095	15,218
bound. value	00:13	00:04	00:08	00:30	22	17,148	35,507
compress	00:49	00:07	02:07	03:40	175	51,534	117,236
edge detect	00:31	00:36	01:43	02:29	66	31,857	66,749
filterbank	00:53	00:02	00:16	04:15	16	10,068	21,483
fir 256 64	00:07	00:02	00:04	00:09	18	12,003	23,874
iir 4 64	00:07	00:14	00:57	01:45	13	33,616	97,285
jpeg2000	00:58	00:21	00:51	02:12	59	25,698	62,291
latnrm 32 64	00:06	00:01	00:32	00:44	15	6,210	15,380
mult 10 10	00:10	00:05	00:18	01:48	35	30,598	58,372
spectral	00:10	00:04	00:12	01:08	58	33,222	68,005
average	00:23	00:08	00:39	01:42	47	23,550	52,855

Table 5.2: Statistics for a Combination of the Homogeneous Task-Level and Pipeline Parallelization Approach for the Arm11MPCore Platform.

described in the motivating example in Section 5.3.1. The parallelized version for a platform with two cores is equal to the one visualized in Figure 5.16 with the difference that pipeline stages T_1 and T_2 are merged into one stage. The parallelization for three and four cores is equivalent to the one shown in Figure 5.17. T_1 is only split twice for the three core-version due to the limited number of cores.

5.3.6.1 Optimization Time & ILP Statistics

Further statistics including the execution time which was required to extract the presented results by a combination of the ILP-based task-level and pipeline parallelization approaches and additional ILP-based statistics are presented in Table 5.2. All results are based on the extraction of both kinds of parallelism for the Arm11MPCore platform. The execution time is divided into the necessary preprocessing steps (including code optimization, execution time estimation and data-dependency analysis) and the extraction algorithm itself. For the latter one, the times for extracting up to two, three, and four tasks are given. CPLEX [IBM13] was used as ILP solver for both approaches. As a result, the setup is equal to the one used to generate the results for the task-level parallelization approach in isolation (cf. Section 5.2.5).

Compared to the evaluation shown in Section 5.2.5, it can be seen that the execution time which was necessary to extract the final solutions increased from 3, 6, and 9 seconds (for two, three, and four parallel tasks) to 8, 39, and 102 seconds in the average case (without the time necessary to perform the preprocessing steps). The higher execution times are reasonable since two approaches are executed in a combined manner, here. Moreover, the construction of the ILP systems used to extract pipeline parallelism is more complex since the considered loops are virtually unrolled to model the timing behavior for each loop iteration in isolation. However, an average execution time between 8 and 102 seconds for two up to four cores is

[8]Measured on an AMD Opteron core running at 2.4 GHz

still fast especially for the achieved solution quality.

The ILP statistics shown on the right-hand side of Table 5.2 are based on the solutions running four tasks in parallel and also show the values of the combined approaches. The first column (#ILPs) shows the number of created and solved ILP systems while the second and third ones present the number of created variables (#Var) and constraints (#Constr) summed up over all ILP systems. For the *compress* benchmark, for example, 175 ILP systems were created and solved containing all in all nearly 52k decision variables and 118k constraints. Also here, the amount of solved ILPs and the number of created decision variables was increased by the combination of both approaches. In contrast, the task-level parallelization approach created only 40 ILPs with 10k variables and 22k constraints for this benchmark. This also explains the increased execution times. In the average case, both approaches created 47 ILP systems for the considered benchmarks with 24k variables and 53k constraints which could be solved in 102 seconds. Hence, one ILP system could be solved in 2.5 seconds on average which is still fast.

5.4 Summary

This chapter presented the first two parallelization approaches developed in the context of this thesis. Both of them were optimized for homogeneous MPSoCs and are based on Integer Linear Programming (ILP). Complex ILP systems could be created and solved efficiently to extract tasks from sequentially written applications, due to the hierarchical divide-and-conquer based approach of the Augmented Hierarchical Task Graph (AHTG) (cf. Chapter 4). In addition, both approaches were also evaluated in a combined fashion which highlights the flexibility of the employed approach. Speedups which are close to the theoretical limits could be observed for some of the considered benchmarks, like, e.g., the matrix multiplication with $2\times$, $2.9\times$, and $3.9\times$ for two, three, and four cores, respectively. The combined approach reached an average speedup of $1.7\times$, $2.3\times$, and $2.9\times$ over all evaluated real-world benchmarks which highlights the efficiency of the extracted results of the proposed approaches. In detail, the following goals could be achieved:

1. Creation and integration of a sophisticated task-level parallelization approach.

2. Creation and integration of a sophisticated pipeline parallelization approach.

3. Combination of task-level, data-level[9], and pipeline parallelization approaches which can be executed separately or in a combined fashion.

4. Creation of parallelization techniques combining task creation, communication and execution costs in high-level models to balance the extracted tasks fully automatically.

[9]DoAll parallelism can be extracted as a special case from the pipeline parallelization approach.

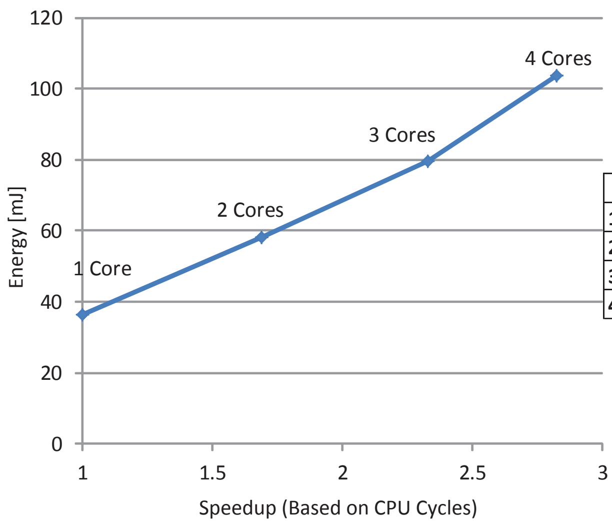
5. Development of approaches optimized for and evaluated on simulated embedded systems.
6. Possibility to limit the number of extracted tasks to ease offline scheduling.
7. Highly efficient solutions can be extracted with speedups of up to 2×, 2.9×, and 3.9× for two, three, and four cores, respectively.

CHAPTER 6

Multi-Objective aware Parallelization for Homogeneous MPSoCs

Contents

The previous chapter presented efficient ILP-based parallelization approaches which are able to extract significant speedups for several real-world embedded benchmarks on different homogeneous target architectures. However, for embedded systems the extraction of high speedups often is not the only optimization objective. Many embedded devices have, e.g., only a very limited amount of computational power, small memories, and are obtaining their energy from a small battery. As a consequence, multiple objectives have to be considered at the same time to map applications onto an embedded MPSoC in an efficient way. Certainly, a lot of execution time can be saved if multiple cores of an MPSoC are executing an application fully in parallel like shown in the previous chapter. But, in general, this requires more energy since all cores must be supplied with power at the maximum voltage level if Dynamic Voltage Scaling (DVS) and Dynamic Voltage and Frequency Scaling (DVFS) are not applied. Figure 6.1, for example, shows measurements for



	Time [Cycles]	Energy [mJ]	Speedup
1 Core	125,256,949	36.433	1.00
2 Cores	74,172,007	58.095	1.69
3 Cores	53,804,696	79.607	2.33
4 Cores	44,389,652	103.655	2.82

Figure 6.1: Speedup vs. Energy Consumption of the parallelized Mult Benchmark [Lee13] simulated on one up to four Cores of the MPARM Platform [BBB+05]

the parallelized matrix multiplication benchmark (*mult*) [Lee13] performed on the MPARM simulator [BBB+05] with the integrated MEMSIM energy model [Kat08], which is based on [WM06] (for more details cf. Section 3.3.1). The application was parallelized with one (sequential execution), two, three, and four concurrently executed tasks, respectively. The simulated platform was configured to provide one, two, three, and four processing units appropriate to the number of extracted tasks. As can be seen, the sequential version of the application executed on a platform equipped with only one ARM 7 core consumed the lowest amount of energy (36 mJ). The version executing four tasks on four ARM 7 cores in parallel reached a speedup of 2.8×, but the consumed energy increased to 104 mJ, which is 185% higher than the sequential version executed on a single-core architecture[1]. If the application designer knows that a certain amount of speedup is sufficient for the execution of the considered application, it may be beneficial to reduce the amount of parallelism in order to put some of the cores into idle mode or switch to a platform providing fewer cores. In this way, a large amount of energy could be saved since fewer cores have to be supplied with power which can significantly increase the battery's runtime. This shows that these trade-offs (which can be extended for several other objectives as well) are important for parallelization techniques tailored towards embedded MPSoCs.

Among others, three major steps have to be performed to map sequentially written applications to multi-processor architectures efficiently. First, the considered application has to be divided into concurrently executed tasks as done by the approaches presented in this thesis. Afterwards, these tasks have to be mapped to

[1]The extracted speedup is lower than the speedup extracted for the previous approaches since the MEMSIM model does not provide caches. This has a negative influence on the parallelized performance since more communication takes place over the shared bus.

the provided processing units of the targeted architecture before the different tasks are finally scheduled in their execution order. Researchers working in the domain of mapping and scheduling optimizations already realized that optimization strategies tailored towards resource-restricted embedded MPSoCs have to consider multiple objectives at the same time, like, e.g., shown in [MTK+11], [TBH+07], [NTS+08], and [HSK+08]. Unfortunately, those techniques rely on the input of the parallelization step. As a consequence, if a parallelization approach only focuses on maximizing the speedup as its only optimization objective, a solution consuming a large amount of energy might be returned. Mapping and scheduling strategies can therefore only try to optimize other objectives in a severely limited search space since they rely on the parallelized application provided as input. In contrast, if a parallelization technique would also consider multiple optimization objectives at the same time, highly optimized tasks for various optimization objectives could be extracted. This also incorporates better optimization potential for the succeeding mapping and scheduling steps. Up to now, this was not well considered by previously published parallelization approaches as discussed in the related work overview in Section 2.4.

Therefore, to the best of the author's knowledge, this chapter presents the first parallelization approaches extracting multi-objective aware parallelism that is highly optimized for homogeneous embedded MPSoCs. However, the size of the solution space is already a problem for single-objective aware parallelization approaches. When considering additional objectives such as energy consumption, the number of possible solution candidates explodes, which renders traditional sophisticated optimization approaches infeasible due to the excessive growth in required computing time to find feasible solutions. This circumstance and the property of Integer Linear Programming (ILP) that solvers return only one solution for the (single) considered objective makes the approaches presented in the previous chapter less attractive for multi-objective aware parallelization strategies. Instead, Genetic Algorithms (GAs) [Mit98], which are a special form of Evolutionary Algorithms [Ash10], have been a popular technique in terms of multi-objective aware optimizations in the last decades. Several researchers have shown in various publications that GAs are well applicable for multi-objective aware optimization techniques even if the optimization is operating on a vast solution space. The combination of these properties makes GAs very attractive for extracting parallelism in a multi-objective aware manner like exploited by the parallelization approaches presented in this chapter. The evaluation of the approaches presented in the previous chapter has revealed that a combination of different parallelization techniques is important to be applicable for applications from multiple application domains. Therefore, a task-level and a pipeline parallelization approach extracting multi-objective aware parallelism are presented here. Moreover, both approaches can be executed in a combined manner. To prune the vast solution space of the parallelization problem, the Augmented Hierarchical Task Graph (AHTG) is also employed here as the central intermediate representation combined with the hierarchical parallelization approach presented in Chapter 4.

The rest of this chapter is structured as follows: The fundamental concepts of Ge-

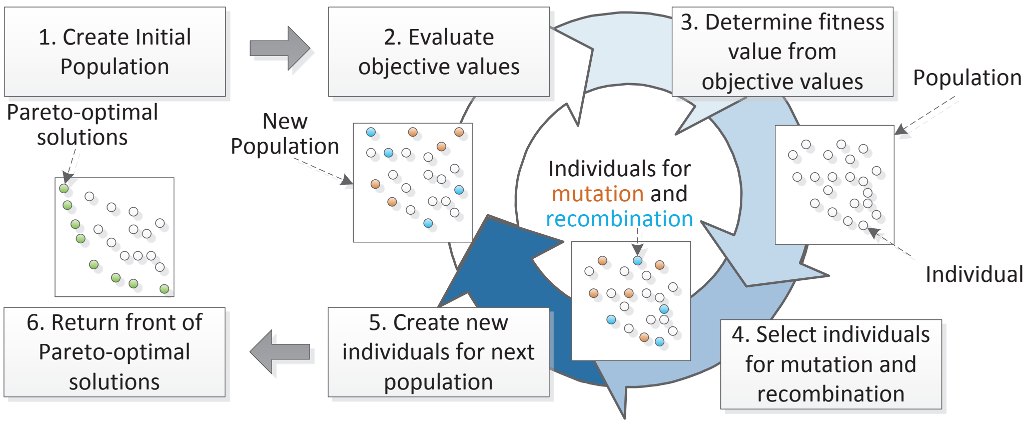

1. Create Initial Population
2. Evaluate objective values
3. Determine fitness value from objective values
Population
Individual
4. Select individuals for mutation and recombination
Individuals for mutation and recombination
5. Create new individuals for next population
New Population
6. Return front of Pareto-optimal solutions
Pareto-optimal solutions

Figure 6.2: General Structure of Genetic Algorithms

netic Algorithms (GAs) are shortly summarized in Section 6.1, before the first GA-based parallelization approach extracting task-level parallelism in a multi-objective aware manner is presented in Section 6.2. Afterwards, Section 6.3 presents the second GA-based parallelization technique for homogeneous architectures, extracting pipeline parallelism while considering multiple objectives. Finally, Section 6.4 summarizes the approaches presented in this chapter. Multi-objective aware parallelization approaches targeting heterogeneous MPSoCs are discussed later in Chapter 8.

6.1 Genetic Algorithms

Optimization is a common goal which is not only aspired by computer architects. One of the oldest participants in the optimization cycle is nature with all its living organisms. It continually adapts the current organisms to changing living conditions. Only the fittest individuals survive and create new offspring by combining their DNA with other individuals. Too weak or less flexible individuals are devoured by other creatures so that they get forgotten over time.

This natural optimization procedure was used as an inspiration for Genetic Algorithms (GAs) (cf., e.g., [Sch75]). GAs start with an initial *population* containing *individuals* representing different solution candidates of the targeted optimization problem. Each individual contains one or more *chromosomes* (strings of DNA) that identify the specific characteristics of each individual. Each chromosome is further divided into an array of *genes* storing values describing the different characteristics of the individuals. These values are also called *alleles*.

The general structure of GAs is depicted in Figure 6.2. As soon as an initial population is created, e.g., randomly (step 1), all individuals are evaluated for the considered objectives (step 2). Afterwards, fitness values based on the objective values are determined (step 3) describing the quality of each individual. With respect to the fitness values, some individuals are selected (step 4) for mutation and recombination to create new promising individuals for the succeeding population (step 5).

The individuals of the new population are now evaluated once again so that steps 2-5 are repeated until a given stopping criterion is met. Finally, the front of Pareto-optimal individuals representing solution candidates is determined and returned as the final solution of the GA (step 6). More information on Genetic Algorithms can, e.g., be found in [Mit98]. The additional steps are also further explained in the following sections describing the components of the GA-based parallelization approaches.

The approaches presented in this thesis employ the PISA framework [BLT+03] with SPEA2 [ZLT01] for selection and variation purposes (steps 3 and 4). Thus, only steps 1, 2, 5, and 6 of Figure 6.2 have to be designed and implemented by the presented approaches. The calculation of the fitness values, as well as the selection of individuals, for recombination and mutation is performed by SPEA2.

6.2 GA-based Task-Level Parallelization Approach

Section 5.2 has revealed that task-level parallelism is well suitable to accelerate sequentially written embedded applications for homogeneous multi-core architectures efficiently. Therefore, this kind of parallelism should also be extracted by the multi-objective aware techniques presented in this Chapter. The approach presented here also operates on the Augmented Hierarchical Task Graph with the global divide-and-conquer-based parallelization approach to extract and combine parallelism with different granularities (cf. Chapter 4). Hence, small groups of statements, different loop(-nest)s and also function calls may be executed in parallel (more details on task-level parallelism with a motivating example is given in Section 5.2.1). The kind of extracted parallelism with the employed fork-join model is equivalent to the one presented in Section 5.2.3. Each hierarchical node is processed in isolation instead of extracting parallelism for the whole application simultaneously. On each hierarchical level, new tasks can be extracted and further combined with parallelism which was extracted deeper in the hierarchy. However, in contrast to the ILP-based approach presented in the previous chapter, the GA-based one described here is multi-objective aware. As a result, the approach of this section is able to generate solutions with low energy consumption, high speedups, low communication overhead, or useful trade-offs between these three objectives instead of just returning the solution with the highest speedup at the expense of other objectives.

The rest of this section is structured as follows: First, Section 6.2.1 explains the integration of the presented parallelization technique into the global parallelization approach. Afterwards, the employed chromosome structure is explained in Section 6.2.2 before the evaluation functions are defined in Section 6.2.3. Finally, the developed mutation and cross-over functions are presented in Section 6.2.4 before the proposed approach is evaluated in Section 6.2.5.

6.2.1 Integration into the Global Parallelization Approach

The integration of the multi-objective aware GA-based task-level parallelization approach optimized for homogeneous MPSoCs into the global parallelization framework (cf. Section 4.2) is shown in Algorithm 7.

Algorithm 7 Pseudo Code of the GA-based Task-Level Parallelization Approach

```
 1: // Called bottom-up hierarchically by Algorithm 4 in line 18 on page 56
 2: function ExtractHomGATLP(Node $n$, Platform $pf$, int $maxTasks$)
 3:     // This function is only applicable to hierarchical nodes.
 4:     if IsNotHierarchicalNode($n$) then
 5:         return $\emptyset$
 6:     end if
 7:     // Extract parallelism for hierarchical node n.
 8:     // All nodes deeper in the hierarchy are already processed.
 9:     $initPopul \leftarrow$ CreateInitialPopulation($n, maxTasks$)
10:     $finalPopul \leftarrow$ HomGATaskLevelParallelizer($n, initPopul, pf, maxTasks$)
11:     $front \leftarrow$ ExtractParetoFrontier($finalPopul$)
12:     return $front$
13: end function
```

The function ExtractHomGATLP is executed by the global parallelization algorithm (cf. Algorithm 4) as soon as all child nodes deeper in the hierarchy are processed. As arguments, the function expects the node n to be parallelized, platform specific information pf containing, e.g., the performance characteristics and the number of available processing units, and an upper bound of extractable tasks $maxTasks$. The value of the variable $maxTasks$ is set to the number of available processing units of the targeted architecture by default and can be customized by the application designer. In this way, the parallelization process does not generate more tasks than there are processing units available by default.

In lines 4-6 the algorithm determines whether the currently processed node n is a non-hierarchical node since only hierarchical ones are processed by the GA-based homogeneous task-level parallelization approach, similar to the ILP-based one. If the node is a hierarchical one, an initial population is created in line 9, first. This initial population contains different individuals representing parallel solution candidates which are randomly generated. In addition, this initial population also contains the sequential solution of the node to be parallelized so that steps upwards in the hierarchy can always fall back to this solution if more efficient parallelism can be extracted there. Afterwards, the GA-based parallelization approach is started in line 10 creating new populations with new solution candidates until a given stopping criterion is met. This stopping criterion is given by a maximum number of created populations, which is dynamically determined based on the number of direct child nodes. Thus, hierarchical nodes with a small number of child nodes are processed faster since their solution space is smaller than the solution space of hierarchical nodes containing more child nodes. As soon as the final population generated by the Genetic Algorithm is returned, containing the best solution candidates found,

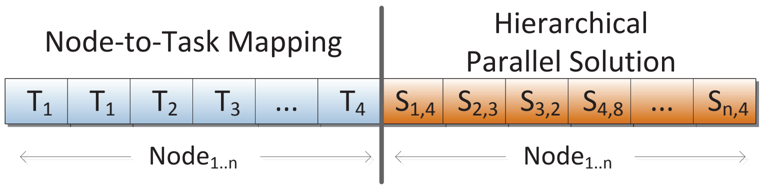


Figure 6.3: Individual's Chromosome Structure

the front of Pareto-optimal solutions is determined and returned as the solution for the processed node in lines 11-12. These solution candidates can further be combined with solution candidates of other parallelization techniques inside the global parallelization method shown in Algorithm 4. In contrast to the ILP-based approaches, the GA-based one is not executed multiple times for different maximum task boundaries. These different solutions are naturally part of the final population returned by the GA.

6.2.2 Chromosome Structure

The first challenge in using Genetic Algorithms is to map the values of the solution space to genes of the individuals' chromosomes in such a way that they can be altered and evaluated efficiently. All child nodes deeper in the hierarchy are already processed due to the bottom-up direction of the global parallelization algorithm, so that a Pareto-front of different solutions, which may contain additional tasks, is attached to each child node. The challenge of parallelizing a hierarchical node is to find an efficient node-to-task mapping for all direct child nodes and to select one of the parallel solutions for each child node. Thereby, the parallelization algorithm is able to extract new parallelism at the current level of the hierarchy which can be combined with parallelism found deeper in the hierarchy. Two goals have to be covered by the GA-based task-level parallelization approach:

I) Statements of direct child nodes have to be mapped to newly extracted, disjunctive tasks to reduce the overall execution time by parallel execution.

II) Newly extracted tasks can be combined with tasks which were extracted deeper in the hierarchy if such a solution increases the overall performance.

These goals have to be integrated into the chromosome structure like depicted in Figure 6.3. The chromosome structure consists of two parts. The first one maps direct child nodes N_i to newly extracted tasks T_j (cf. Goal (I)) while the second one selects one hierarchical solution $S_{i,k}$[2] for each direct child node N_i (cf. Goal II). The genes' values in Figure 6.3, for example, specify that nodes N_1 and N_2 are mapped to task T_1 while node N_3 is mapped to task T_2. Additionally, the hierarchical parallel solution $S_{1,4}$ is chosen for node N_1, parallel solution $S_{2,3}$ is chosen for node N_2, etc. The individuals' chromosomes are only twice as large as the number of direct child

[2] k is just an index to distinguish between the different solutions of node N_i.

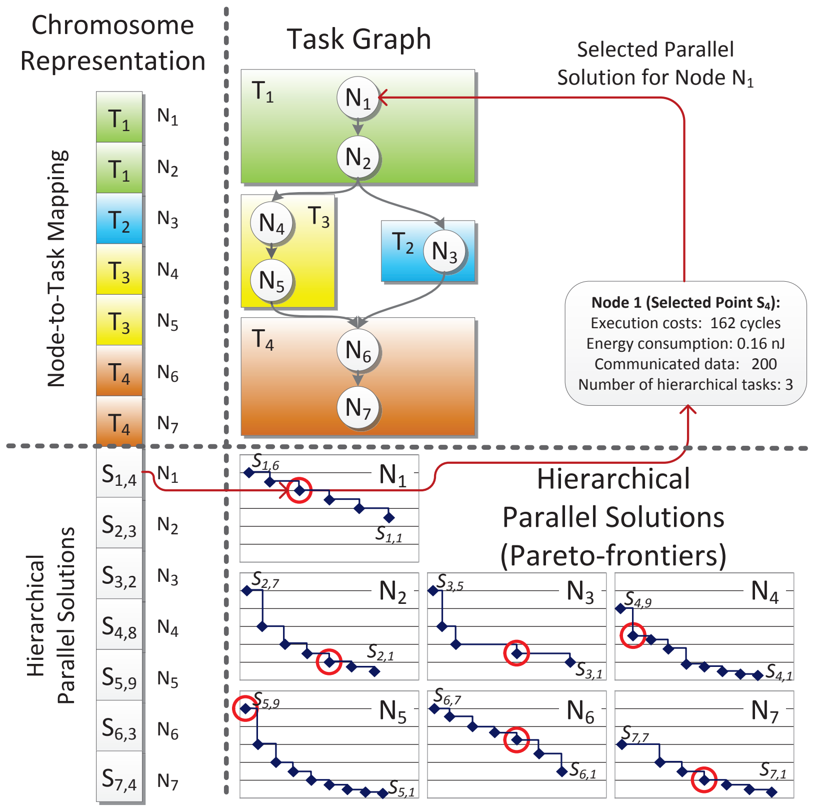

Chromosome Representation
Task Graph
Selected Parallel Solution for Node N1
Node-to-Task Mapping
T1 N1
T1 N2
T2 N3
T3 N4
T3 N5
T4 N6
T4 N7
Node 1 (Selected Point S4):
Execution costs: 162 cycles
Energy consumption: 0.16 nJ
Communicated data: 200
Number of hierarchical tasks: 3
Hierarchical Parallel Solutions
S1,4 N1
S2,3 N2
S3,2 N3
S4,8 N4
S5,9 N5
S6,3 N6
S7,4 N7
Hierarchical Parallel Solutions (Pareto-frontiers)

Figure 6.4: Impact of Chromosome Configuration on the Hierarch. Node to be Parallelized

nodes so that they can be encoded efficiently by a small array of integer values. This enables the creation of a large number of solution candidates with low memory consumption.

The impact of the chromosome's configuration is visualized in Figure 6.4. The example parallelizes a hierarchical node with seven child nodes which can be mapped to up to four newly created tasks. The figure shows the genes' values on the left-hand side and their impact on the evaluation on the right-hand side. The upper part of the figure shows the task graph representation of the node to be parallelized according to the node-to-task mapping defined on the left-hand side. As can be seen, nodes N_1 and N_2 belong to task T_1 while node N_3 belongs to task T_2. Edges between the created tasks depend on the node-to-task mapping. Here, a dependence edge between nodes N_2 and N_3 exists which implies a dependence edge between tasks T_1 and T_2. Thus, the execution of task T_2 has to wait for completion of T_1 since data has to be communicated between both tasks before T_2 can start with its execution. These task execution orders, as well as inserted communication costs, should be considered by the evaluation functions for the applied objectives (cf. Section 6.2.3). The evaluation functions must also be aware of changing paths within the task graph structure. If, for example, node N_5 would be moved from task T_3 to task T_2 by mutation or recombination (cross-over), a new dependency between T_3 and T_2

would arise which has to be taken into account for the evaluation of the objective values.

The second part of the chromosome representation shown in Figure 6.4 selects hierarchical parallel solutions which were extracted deeper in the hierarchy for all child nodes. For each of them, a front of Pareto-optimal solutions exists. The contained solutions were evaluated by the high-level cost functions presented in the next section. The approach has to choose exactly one solution candidate from each child node's Pareto-frontier providing different objective values for the corresponding node. A solution with more extracted parallelism may, for example, reduce the overall execution time at the cost of the system's energy consumption. This part of the chromosome's structure also influences the evaluation of the objective values. In the example shown, hierarchical solution $S_{1,4}$ is selected for node N_1 while $S_{2,3}$ is selected for N_2, etc.

By combining both parts of the chromosome structure, all decisions that have to be taken to extract efficient task-level parallelism while considering multiple objectives at a time are modeled (cf. Goal (I) and (II)). The next section presents the high-level models used to evaluate the quality of different individuals based on the genes' values.

6.2.3 Objective Evaluation

The second functionality that has to be provided for GA-based optimizations consists of evaluation models. These models are used to determine the solutions' quality for the considered objectives (cf. step 2 in Figure 6.2). The employed selection and variation framework (e.g. SPEA2) uses these values to compute a fitness value (cf. step 3 in Figure 6.2) to select the most promising solution candidates for mutation and recombination (cf. step 4 in Figure 6.2). In the current version of the parallelization approach, three objectives are considered. They cover the execution time, which reflects the speedup of the application, the energy consumption of executing the application on the embedded device, and the communication overhead, which gives a hint on the bus load of the parallelized application. It is, of course, also possible to take more objectives into account in future research.

Since the quality of the final Pareto-optimal solutions returned by a Genetic Algorithm primarily depends on the population sizes used and the number of generated populations, the configuration of an individual's chromosome must be evaluated efficiently. Simulating each solution candidate on the target platform is not an option since it would be too time consuming. Instead, the employed genetic parallelization algorithm uses high-level models to evaluate the different objectives. The models employed here do not have to be highly precise. Instead, they should provide a good trade-off between fast evaluation and enough accuracy to check whether a solution candidate may increase the application's overall performance. Otherwise, it would not be possible to evaluate a large number of generated solution candidates which would result in a reduced solution quality of a GA-based approach. Of course, the quality of the models used is later verified by simulation in Section 6.2.5.

6.2.3.1 Objective 1: Execution Time

The evaluation of the objective value for the execution time of a parallelized node's gene values is similar to the modeled execution time of the ILP formulations of the task-level parallelization approach presented in Section 5.2.4. Consequently, the objective value is equal to the execution time of the longest (most critical) path through the sub-graph of the parallelized hierarchical node. In the example shown in Figure 6.4, the longest execution path is either $N_1 \rightarrow N_2 \rightarrow N_4 \rightarrow N_5 \rightarrow N_6 \rightarrow N_7$ or $N_1 \rightarrow N_2 \rightarrow N_3 \rightarrow N_6 \rightarrow N_7$ or more specifically $T_1 \rightarrow T_2 \rightarrow T_4$ or $T_1 \rightarrow T_3 \rightarrow T_4$ depending on the nodes' execution times and the communication delay of the tasks created. The following equations show the calculation of the objective value for the execution time in a more formal way:

The execution time $ET(T_i)$ for task T_i is equal to the sum of the execution times $ETN(n, S_{n,k})$ of all child nodes n which are mapped to task T_i increased by a constant task creation overhead TCO like defined in Equation 6.1. The execution time $ETN(n, S_{n,k})$ of child node n is defined by the chosen hierarchical parallel solution $S_{n,k}$ (cf. Goal (II)) and was calculated by the high-level models proposed in this section at the time node n was processed.

$$ET(T_i) = TCO + \sum_{n \in Nodes(T_i)} ETN(n, S_{n,k}) \tag{6.1}$$

The path costs $PC(T_i)$ of task T_i are recursively defined in Equation 6.2 and can only be used in this form since it is ensured that the sub-graph is cycle-free. The path costs are based on predecessor relationships between the extracted tasks based on data dependencies of the mapped child nodes. If, e.g., node n is mapped to task T_i while node m is mapped to task T_j and a directed edge exists from node n to node m, then task T_i is a predecessor of task T_j such that T_j has to wait until T_i has completed the execution of its statements and communicated the required data to T_j. Therefore, the path costs $PC(T_i)$ of task T_i are equal to the sum of the execution times $ET(T_i)$ (cf. Equation 6.1) of task T_i itself plus the path costs $PC(t)$ of the most expensive predecessor task t including the communication costs $CC(t, T_i)$ from task t to T_i.

$$PC(T_i) = ET(T_i) + \max\{PC(t) + CC(t, T_i) | \forall t \in Pred(T_i)\} \tag{6.2}$$

Finally, the overall execution time is equal to the longest execution path of the node's sub-graph. Since the path costs $PC(T_i)$ contain the costs of T_i and all its predecessors, the longest path is equal to the maximum path costs like shown in Equation 6.3.

$$OverallET = \max\{PC(t) | \forall t \in Tasks\} \tag{6.3}$$

6.2.3.2 Objective 2: Energy Consumption

The objective value of the solution candidates' energy consumption contains energy costs which arise due to task spawning, statement execution and communication costs, like shown in the following equations.

The energy consumption $ICE(T_i)$ (Incoming Communication Energy) for receiving the necessary input data of a task T_i is determined first. $ICE(T_i)$ is calculated by summing up a static overhead for the incoming data $ICEO$ (for, e.g., setting up the communication channels etc.) and a factor ICM (Incoming Communication Multiplier) per communicated byte like shown in Equation 6.4.

$$ICE(T_i) = \sum_{d \in InData(T_i)} ICEO + \#Bytes(d) * ICM \tag{6.4}$$

The energy consumption $OCE(T_i)$ (Outgoing Communication Energy) for the outgoing communications is similar to $ICE(T_i)$ like defined in Equation 6.5.

$$OCE(T_i) = \sum_{d \in OutData(T_i)} OCEO + \#Bytes(d) * OCM \tag{6.5}$$

The total amount of energy $E(T_i)$ consumed by each task T_i is equal to the sum of a constant task creation overhead TCE and the energy $EEN(n, S_{n,k})$ which has to be spent to execute all direct child nodes n mapped to T_i. The energy consumption $EEN(n, S_{n,k})$ of child node n depends on the chosen hierarchical solution candidate $S_{n,k}$ and was estimated by the proposed high-level models of this section as soon as n was parallelized. Finally, $E(T_i)$ is increased by the energy consumption for incoming $(ICE(T_i))$ and outgoing $(OCE(T_i))$ communication like defined in Equation 6.6.

$$E(T_i) = TCE + \sum_{n \in Nodes(T_i)} EEN(n, S_{n,k}) + ICE(T_i) + OCE(T_i) \tag{6.6}$$

The estimated overall energy consumption for an individual's configuration is equal to the sum of the energy consumption of all tasks like shown in Equation 6.7.

$$OverallEnergy = \sum_{t \in Tasks} E(t) \tag{6.7}$$

6.2.3.3 Objective 3: Communication Overhead

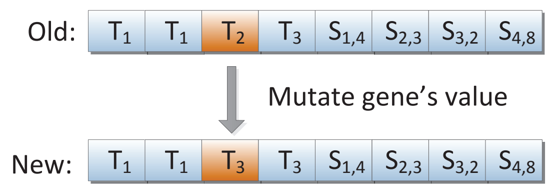
The evaluation of the communication overhead objective value is the simplest one and is equal to the sum of the communicated bytes $(\#Bytes(data))$ of all tasks multiplied by a specified communication delay $COSTS$ like shown in Equation 6.8.

$$CommOverhead = \sum_{data \in Comm} \#Bytes(data) * COSTS \tag{6.8}$$

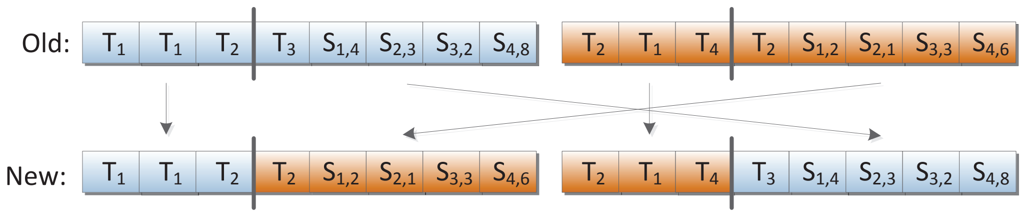
6.2.3.4 Portability of Models

To enable portability to multiple target platforms, the hierarchical task graph is augmented with additional cost information, like, e.g., execution time and energy consumption. Hence, as long as these values can be extracted, the presented approach and its high-level models are portable to multiple target platforms.

In addition, all constants of the evaluation functions can be configured in the framework of this thesis. The communication costs $CC(t, T_i)$ are determined by


Old:
T1 T1 T2 T3 S1,4 S2,3 S3,2 S4,8
Mutate gene's value
New:
T1 T1 T3 T3 S1,4 S2,3 S3,2 S4,8

(a) Mutation Function


Old:
T1 T1 T2 T3 S1,4 S2,3 S3,2 S4,8
T2 T1 T4 T2 S1,2 S2,1 S3,3 S4,6
New:
T1 T1 T2 T2 S1,2 S2,1 S3,3 S4,6
T2 T1 T4 T3 S1,4 S2,3 S3,2 S4,8

(b) Cross-Over Function

Figure 6.5: Mutation and Cross-Over (Recombination) Functions

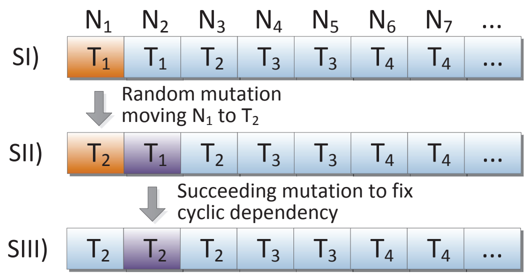
multiplying a configurable communication factor by the amount of communicated bytes. The task creation overheads for execution time TCO and energy consumption TCE can also be configured, to mention only some of the configurable constants. By combining these configurable parts with the extracted objective values, it should be easy to adapt the framework to multiple target architectures.

6.2.4 Mutation & Cross-Over

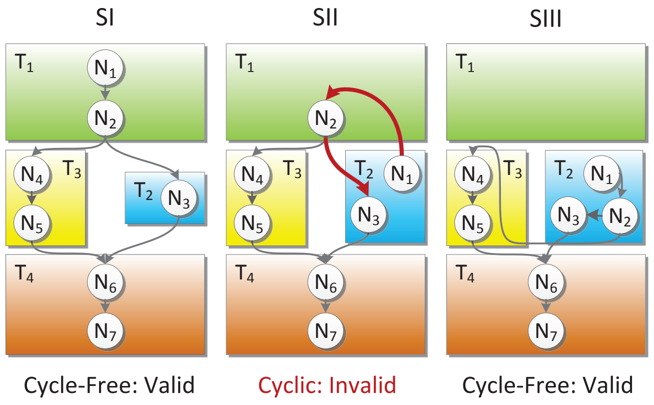
The last missing piece to complete the work-flow of a Genetic Algorithm is the provision of mutation and cross-over functions. These functions alter the most promising solution candidates to create new populations for the GA's next iterations (cf. step 5 in Figure 6.2).

In the current version of the presented parallelization approach, a 1-position mutation strategy is implemented, like visualized in Figure 6.5(a). One gene of the mutating individual's chromosome is randomly chosen and modified. Thus, one direct child node is moved from one task T_i to another one T_j or a different hierarchical parallel solution candidate $S_{n,k}$ is chosen for node n. The employed cross-over function (also called recombination) splits two individuals at a random position and joins the left-hand side of the first one with the right-hand side of the second one and vice versa (cf. Figure 6.5(b)). This can easily be achieved since all chromosomes of the same node to be parallelized have the same length and the split position is equal for both individuals. With the combination of both operations, the GA is able to create new solution candidates based on promising existing ones.

So far, both functions seem to be simple state-of-the-art implementations. However, experiments have shown that a large number of invalid solutions ($> 50\%$) is created by such simple mutation and cross-over functions which drastically reduces the solution quality of the employed Genetic Algorithm. Solutions are invalid if,


N1 N2 N3 N4 N5 N6 N7 ...
SI) T1 T1 T2 T3 T3 T4 T4 ...
Random mutation moving N1 to T2
SII) T2 T1 T2 T3 T3 T4 T4 ...
Succeeding mutation to fix cyclic dependency
SIII) T2 T2 T2 T3 T3 T4 T4 ...

(a) Chromosome



(b) Corresponding Task-Graph

Figure 6.6: Smart Mutation Function Fixing Invalid Solution Candidates

e.g., a cyclic dependency between the created tasks appears by mutation or recombination so that one task is waiting for data of another one and vice versa. This can occur if a child node is moved from one task to another one since dependencies between the tasks may change, as well. The presented GA-based approach also marks solutions with more concurrently executed tasks than available processing units to be invalid to reduce additional runtime overhead for scheduling. For such invalid solutions, the objective evaluation functions return the highest possible objective values to take care that those solutions are not part of the final Pareto front (a similar technique is, e.g., also employed in [TBH+07]). Nevertheless, they are still part of the solution space so that they can be selected for mutation or recombination with a low probability. The performed evaluations have shown that it is better to avoid the creation of too many invalid solution candidates.

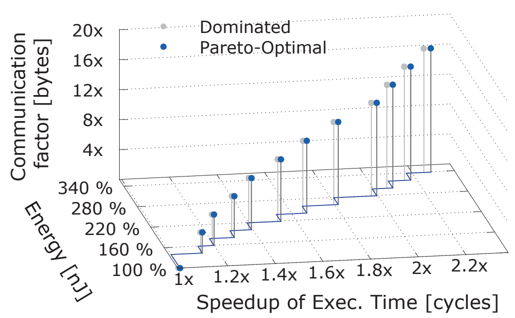
An example for a mutation step causing a cyclic dependency is given in Figure 6.6. Figure 6.6(a) shows the chromosome representation and the performed mutation steps while Figure 6.6(b) covers the corresponding changes in the extracted task graph. The task graph of situation *SI* is free of dependency-based cycles. If, for example, node N_1 is moved from task T_1 to T_2, like depicted in situation *SII*, a cyclic dependency between tasks T_1 and T_2 appears such that T_1 waits for data generated by T_2 and vice versa. Since in the employed model tasks can only start when the necessary input data is available, this situation would lead to a deadlock. Therefore, a smart mutation strategy was developed as part of this thesis which automatically tries to fix cyclic dependencies. Such a strategy is commonly used to optimize a GA-based approach (cf., e.g., [Mit05]). The strategy employed by the approach presented in this section determines the source of the dependency by considering the predecessors and successors of the mutated node and tries to solve this cycle by a succeeding mutation. If necessary, this step is continued until a cycle-free solution could be extracted or a maximum number of fixing steps is reached. This is also visualized in Figure 6.6 between situations *SII* and *SIII*. Here, the algorithm has detected that an edge from N_1 to N_2 exists which causes the cyclic dependency between T_1 and T_2 so that the gene's value of N_2 is modified by a directly following succeeding mutation step. As a result, *SIII* is cycle-free again so that a valid solution is returned to the GA-based approach. A similar correction method is also performed if a mutation step of a selected hierarchical solution candidate created too many extracted tasks. Here, a different solution is chosen for one of the other nodes to create a valid solution.

Invalid solutions generated by cross-over methods are fixed in a similar way. Here, the genes to the left and right of the cutting position are verified and used as a starting point to fix occurred cyclic dependencies or an invalid number of extracted tasks. By using these smart correction algorithms, the number of generated invalid solutions could be decreased from over 50% down to less than 4%. This supports the GA to find efficient solutions significantly faster since more valid solutions are analyzed in each population. The author of this thesis is aware of the fact that these modifications applied to the mutation and cross-over functions may influence the solutions generated by the GA-based approach. Therefore, the user can choose between the original and the smart mutation and cross-over functions in the framework. The evaluation chapter will later show that the smart mutation and cross-over functions lead to efficient results in a short period of time.

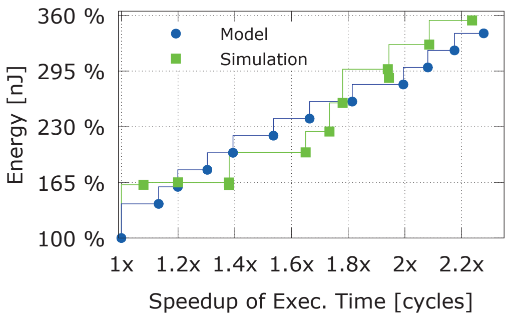
6.2.5 Experimental Results

To evaluate the applicability of the newly presented multi-objective aware task-level parallelization approach for homogeneous embedded MPSoCs the same set of benchmarks was chosen as before. The cycle-accurate MPARM simulator [BBB+05] providing up to four single-core ARM processors was selected as target platform. The simulator is equipped with a detailed energy model called MEMSIM [Kat08] which is based on [WM06]. The other target architectures, used for evaluation in the previous chapter, could not be used here since the Virtualizer tool suite does not provide energy models. The operating system, the employed middleware as well as the measurement structure are identical to the evaluations presented in the previous chapter.

Detailed results for three of the considered applications are presented in Figures 6.7 - 6.9. All figures show the Pareto-frontier of all considered objectives (based on the high-level models presented in Section 6.2.3) on the left-hand side



(a) Model-based Results

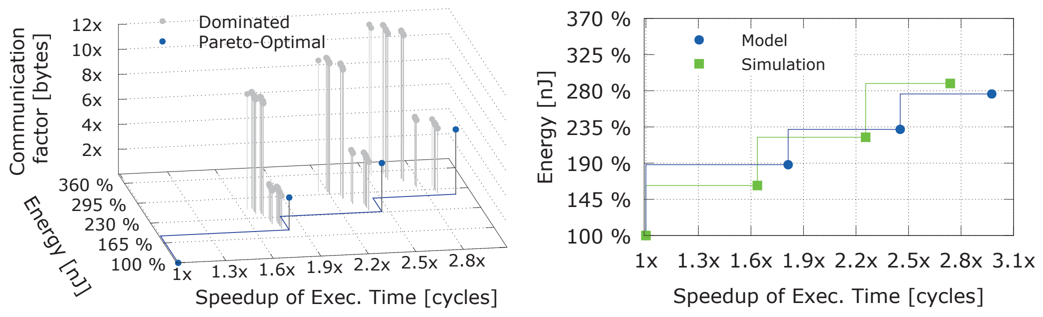


(b) Simulation-based Results

Figure 6.7: Final Parallel Solutions for the Edge Detect Benchmark.

(Figures 6.7(a) - 6.9(a)). The right-hand side of each figure (Figures 6.7(b) - 6.9(b)) depicts results obtained by simulation on the MPARM simulator for execution time and energy consumption to validate the accuracy of the employed models. The framework supports three objectives, namely speedup of the execution time, energy consumption and inserted communication overhead. These objectives are arranged on the x-, y-, and z-axes on each figure's left-hand side, accordingly. Other objectives, like, e.g., the reduction of thermal issues or the size of allocated memory, can easily be added by future research work. The sequential version of the application is located at the bottom left of each diagram. This solution is the slowest one with a speedup of 1× and consumes the lowest amount of energy executed on a single-core architecture configuration. This solution is used as a baseline for all considered objectives. For improved readability, vertical bars are added to the left-hand sides' diagrams to project the points into the x-y-plane. A solid blue line marks the front of Pareto-optimal solutions, also projected to the x-y-plane. Each diagram contains both, Pareto-optimal (blue) and Pareto-dominated solutions (gray). Of course, only the first ones are finally returned to the application designer as possible solution candidates. Such a front of Pareto-optimal solutions is generated for each node of the AHTG during parallelization until the root node of the graph is reached. The root node's Pareto front is finally returned to the application designer containing solution candidates with parallelism from different granularity levels. Each solution was evaluated on an architecture with an appropriate number of processing units. A solution running, for example, at most three tasks in parallel is simulated on a platform providing three cores.

The application designer is now able to choose one of the Pareto-optimal solutions which complies with the best trade-off for a considered application scenario. If, for example, a speedup of 1.4× is sufficient for the parallelized *edge detect* benchmark (cf. Figure 6.7(a)), the amount of consumed energy (compared to the sequential execution) increases to around 200%. If the solution with the highest speedup would be chosen – like done by most existing parallelization approaches, which are only optimizing for speedup – the energy consumption would increase to over 340%



(a) Model-based Results (b) Simulation-based Results

Figure 6.8: Final Parallel Solutions for the Mult Benchmark.

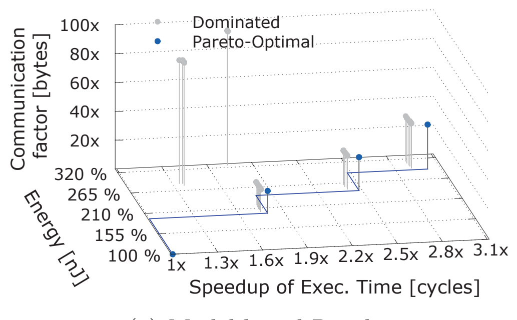
for a speedup of 2.3×[3]. 140% of energy could be saved here, by using the new multi-objective aware task-level parallelization approach. Overall, twelve different solutions are returned as the final solution candidates for the *edge detect* benchmark providing meaningful trade-offs between the considered objectives.

Similar trade-offs can also be observed for the benchmarks *mult* (cf. Figure 6.8(a)) and *boundary value* (cf. Figure 6.9(a)). Compared to the results of the *edge detect* application, it can be seen that a lower number of Pareto-optimal points were found since many solutions are Pareto-dominated. Nevertheless, speedups of up to 2.8× with approximately 280% energy consumption (*mult*) and 3.1× with 310% energy consumption (*boundary value*) span a large solution space for trade-offs generated by the presented multi-objective aware task-level parallelization approach.

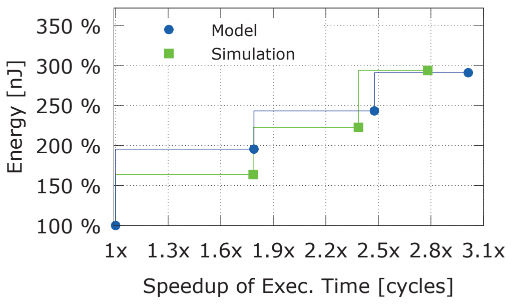
6.2.5.1 Evaluation of High-Level Objective Models

Since the applicability of the presented approach strongly depends on the soundness of the employed high-level models, Figures 6.7(b) - 6.9(b) compare the Pareto-optimal points of the model to the simulated results on the target platform. The third objective (amount of communication) is not shown here since it cannot be measured by the simulator. Nevertheless, the most important ones are speedup and energy consumption. The MPARM simulator was configured to have the same amount of cores as the number of concurrently executed tasks appearing in the parallelized application. Thus, a solution with two concurrently executed tasks is executed on a platform with two cores. Of course, there can be multiple parallel regions in the application. All cores that are not executing threads at a given timeframe are put into idle mode to save energy. As depicted in the figures, the trend of increasing energy consumption for more expressed parallelism like estimated by the employed high-level models was confirmed by the results returned by the simulator. Moreover, the figures show that the points based on the proposed high-

[3]The extracted speedups are lower than the speedups extracted for the previous approaches since the MEMSIM model does not provide caches. This has a negative impact on the parallelized performance since more communication takes place over the shared bus.



(a) Model-based Results



(b) Simulation-based Results

Figure 6.9: Final Parallel Solutions for the Boundary Value Benchmark.

level models are comparable to the simulated ones making the models accurate enough to be used to extract parallelism in a multi-objective aware manner. By using these models, it is possible to extract a large number of solution candidates since they can be evaluated very efficiently. This would not be possible if the objective values would be determined by, e.g., time consuming simulations for each extracted solution candidate. This is important since the solution quality of a GA-based approach strongly depends on the number of created solution candidates.

6.2.5.2 Additional Results & Statistics

Summarized results for all evaluated benchmarks can be found in Table 6.1. The columns contain information about the time in minutes which was necessary to parallelize the applications with the presented parallelization approach (*Time*), the number of processed nodes (#*N*), the number of generated populations (#*Pop*), the overall number of generated and evaluated individuals (#*Ind*), the number of mutated (#*Mut*) and recombined (#*Cross*) individuals and the number of offered Pareto-optimal solutions (#*Sol*) which are returned to the application designer. The number of individuals and populations used to parallelize a hierarchical node is determined dynamically, based on the number of child nodes and the number of hierarchical solutions found. Therefore, nodes offering less parallelized solutions are processed much faster. The shown numbers of populations, individuals, mutations, etc., are summed up over all parallelized hierarchical nodes.

As can be seen, the new multi-objective aware approach is able to create and evaluate individuals very fast due to the use of the presented high-level models. To parallelize, e.g., the *compress* benchmark, more than half a million individuals were generated and evaluated in about 9 minutes. The number of offered Pareto-optimal solutions varies between 4 and 42 (11 in the average case), depending on the available parallelism of the application. This shows that the presented approach is able to provide the application designer a large amount of freedom in finding good trade-offs in the parallelization process.

Compared to the ILP-based approaches, the GA-based one requires significantly more execution time to find reasonable solutions. One of the reasons is that GA-

Benchmark	Time[4]	#N	#Pop	#Ind	#Mut	#Cross	#Sol
adpcm enc.	00:49	27	1,376	146,003	27,164	91,616	5
bound. value	01:04	6	532	79,933	15,122	50,492	4
compress	09:07	131	6,552	592,124	110,426	371,048	5
edge detect	02:08	47	2,088	175,203	32,506	109,528	12
filterbank	01:36	4	212	22,198	4,223	14,012	6
fir 256 64	00:18	7	292	26,462	4,883	16,620	4
iir 4 64	00:51	7	564	82,224	15,373	51,996	6
jpeg2000	04:36	43	2,468	294,970	55,655	186,988	42
latnrm 32 64	00:13	11	460	37,122	6,884	23,172	4
mult 10 10	00:14	10	404	34,511	6,529	21,500	4
spectral	01:36	38	1,948	190,266	35,498	120,084	33
average	02:02	31	1,536	152,819	28,569	96,096	11

Table 6.1: Evaluation of GA-based Task-Level Parallelization Approach for Homogeneous MPSoCs

based approaches do not know whether they found an optimal solution. Instead, they have to iterate until a given stopping criterion (e.g., a maximum number of created populations) is met. The GA-based task-level parallelization approach took two minutes on average to extract the final solution, whereas the ILP-based approach could finish its work in only nine seconds for a four core architecture (cf. Section 5.2.5). However, the novel approach of this chapter is able to deliver trade-offs for multiple objectives, which was not possible for the ILP-based approaches.

To summarize, the following results were achieved:

1. A large optimization potential is exploited since multiple objectives are considered at the same time in the parallelization process especially for embedded devices.
2. The new multi-objective aware task-level parallelization approach produces solutions with good trade-offs between high speedups, low energy consumption, and low communication overhead.
3. The time to extract these solution candidates with the GA-based approach ranges between 13 seconds up to 9 minutes for the considered benchmarks and thus stays in an acceptable amount of time.

6.3 GA-based Pipeline Parallelization Approach

The evaluation of the multi-objective aware task-level parallelization approach in the previous section has shown that the presented GA-based parallelization technique is able to provide well-balanced solutions with trade-offs between the considered objectives. However, the ILP-based parallelization techniques presented in Chapter 5 revealed that pipeline parallelism is important for embedded systems since

[4]Time format MM:SS, measured on an AMD Opteron core running at 2.4 GHz.

many applications are written in a pipelined manner. Therefore, the multi-objective aware parallelization techniques presented in this chapter are extended to be able to extract well-balanced pipeline parallelism. The pipeline parallelization approach presented in this section extracts the same kind of parallelism as the ILP-based one (cf. Section 5.3) and is also integrated into the global parallelization approach described in Chapter 4. Also here, an augmented Program Dependence Graph (PDG) (cf. Section 5.3.2) is used as an intermediate representation to divide loop(-nests) horizontally and vertically into concurrently executed tasks (cf. Section 5.3.4). It is recommended to make sure to have read at least the referenced sections before continuing with the approach presented in this section.

The rest of this section describes the multi-objective aware pipeline parallelization approach in more detail and highlights its combination with the previously presented task-level parallelization approach. Therefore, Section 6.3.1 presents the integration of the pipeline parallelization technique into the global parallelization framework. Afterwards, the employed chromosome structure is explained in Section 6.3.2 before the evaluation functions are defined in Section 6.3.3. Finally, the developed mutation and cross-over functions are presented in Section 6.3.4 before the proposed approach is evaluated in combination with the multi-objective aware task-level parallelization approach in Section 6.3.5.

6.3.1 Integration into the Global Parallelization Approach

The integration of the multi-objective aware GA-based pipeline parallelization approach optimized for homogeneous MPSoCs into the global parallelization framework (cf. Section 4.2) is shown in Algorithm 8.

Algorithm 8 Pseudo Code of the GA-based Pipeline Parallelization Approach

```
1: // Called bottom-up hierarchically by Algorithm 4 in line 18 on page 56
2: function ExtractHomGAPipeline(Node $n$, Platform $pf$, int $maxTasks$)
3:     // This function is only applicable to loops.
4:     if not $n.getStmt().isLoopStmt()$ then
5:         return $\emptyset$
6:     end if
7:     // Create an augmented PDG for the loop(-nest).
8:     $loopPDG \leftarrow$ ConstructPDG($n.getStmt()$)
9:     // Extract pipeline parallelism from the loop's PDG.
10:    $iPopul \leftarrow$ CreateInitialPopulation($loopPDG, maxTasks$)
11:    $finalPopul \leftarrow$ HomGAPipelineParallelizer($loopPDG, iPopul, pf, maxTasks$)
12:    $front \leftarrow$ ExtractParetoFrontier($finalPopul$)
13:    return $front$
14: end function
```

The function ExtractHomGAPipeline is executed by the global parallelization algorithm (cf. Algorithm 4) as soon as all child nodes are processed. As arguments, the function expects the node n to be parallelized, platform specific information pf containing, e.g., the performance characteristics and the number of

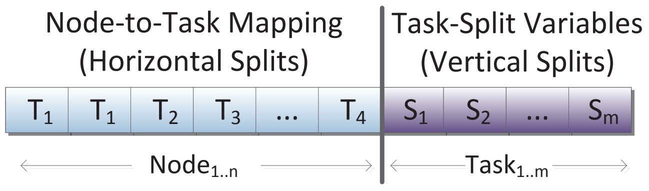

Node-to-Task Mapping (Horizontal Splits)
Task-Split Variables (Vertical Splits)
T1 T1 T2 T3 ... T4
S1 S2 ... Sm
Node1..n
Task1..m

Figure 6.10: Individual's Chromosome Structure

available processing units, and an upper bound of extractable tasks $maxTasks$. The variable $maxTasks$ is set to the number of available processing units of the targeted architecture by default and can be customized by the application designer.

The integration is similar to the integration of the multi-objective aware task-level parallelization approach presented in the previous section. Here, the approach is only applicable to flat or nested loops of the application to be parallelized. If the currently processed node of the AHTG does not represent a loop statement, the function ExtractHomGAPipeline returns an empty set of solutions in line 5 of Algorithm 8. Otherwise, an augmented PDG is created containing only those nodes which are part of the considered loop's body (cf. Section 5.3.2). Afterwards, an initial population is created in line 10 containing the sequential solution of the loop as well as randomly generated solution candidates splitting the loop horizontally and vertically into concurrently executed tasks. Based on this initial population, the loop's PDG, the platform information, and the upper task boundary, the GA-based pipeline parallelization approach starts to iterate over several populations to create efficient parallelized versions of the processed loop in line 11. Finally, the front of Pareto-optimal solution candidates is extracted and returned in lines 12-13 based on the resulting population of the GA-based approach.

The global parallelization algorithm (cf. Algorithm 4) has the possibility to execute other parallelization techniques as well (like the GA-based task-level parallelization approach) to combine the results of different parallelization techniques.

6.3.2 Chromosome Structure

In order to extract pipeline parallelism in the form described in Section 5.3.1, two goals have to be covered by the structure of the chromosomes:

I) Statements of the loop's body have to be mapped to disjunctive pipeline stages (horizontal splits) to profit from parallel execution.

II) Different iterations of each pipeline stage have to be allocatable to concurrently executed sub-tasks (vertical splits) to further increase parallel execution.

Both splits have to be part of each individual's chromosome to extract pipeline parallelism in a multi-objective aware manner[5]. The employed chromosome struc-

[5]The pipeline parallelization approach does not combine hierarchical solution candidates since it is hard to handle a different amount of inner tasks for different iterations of the loop. However, this limitation is based on a technical simplification and can be removed in future work.

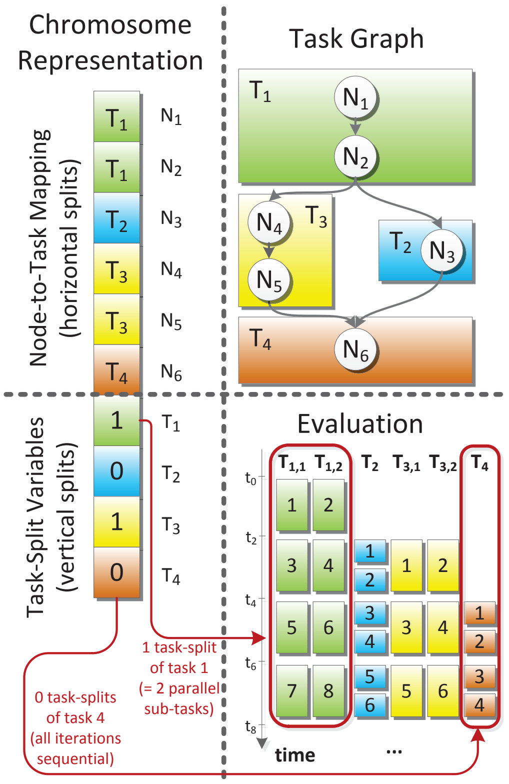

Chromosome Representation
Task Graph
Node-to-Task Mapping (horizontal splits)
T1 N1
T1 N2
T2 N3
T3 N4
T3 N5
T4 N6
Task-Split Variables (vertical splits)
1 T1
0 T2
1 T3
0 T4
Evaluation
T1,1 T1,2 T2 T3,1 T3,2 T4
t0 t2 t4 t6 t8
time
1 task-split of task 1 (= 2 parallel sub-tasks)
0 task-splits of task 4 (all iterations sequential)

Figure 6.11: Impact of Chromosome Configuration on the Parallelized Node

ture is shown in Figure 6.10. The structure of the chromosomes is divided into two parts. The mapping of PDG nodes, representing statements of the loop's body, to pipeline stages is shown on the left-hand side. Here, each node is mapped to exactly one pipeline stage created by horizontal splits. On the right-hand side, an integer variable declares how often each pipeline stage is split into sub-tasks which are executing different loop iterations of the stage in parallel. In the example shown in Figure 6.10, the statements represented by nodes N_1 and N_2 are mapped to pipeline stage T_1 while node N_3 is mapped to stage T_2. The first pipeline stage T_1 is split S_1 times, T_2 is split S_2 times, and so on. Each chromosome can be encoded by an array of integers. The size of this array is only as large as the number of statements contained in the loop's body plus the maximum number of pipeline stages to generate. Hence, each chromosome can be encoded efficiently enabling the generation of a large amount of individuals consuming only a small amount of memory.

The impact of a chromosome's configuration is visualized in more detail in Figure 6.11. In the top part of the figure, nodes N_1 and N_2 are mapped to the first pipeline stage (T_1), N_4 and N_5 are mapped to stage T_3 while N_3 and N_6 are mapped to stages T_2 and T_4, respectively. T_1 starts with the execution of the first iteration of nodes N_1 and N_2. Afterwards, the generated data is sent to pipeline stages T_2 and T_3 so that the next iteration of T_1 is executed concurrently to the first iteration of T_2 and T_3. The dependencies between the different extracted pipeline stages rely on the node to pipeline stage mapping. If one node is moved from one pipeline stage to another one, e.g., by mutation or recombination, the dependencies between the pipeline stages may change which also influences the execution order of the stages. If, e.g., node N_2 would be moved from pipeline stage T_1 to T_2, a new edge would arise between stages T_2 and T_3 which has to be taken into account while mutating and recombining individuals. As a consequence, even small changes in the mutation steps may have big influences on the evaluation of different objectives. Of course, not all tasks have to be used so that some of the processors can be put into idle mode to, e.g., save energy.

The genes representing vertical splits are shown in the lower left part of Figure 6.11. Here, pipeline stages T_1 and T_3 are split once, resulting in two sub-tasks for both pipeline stages. Stages T_2 and T_4 are not split in this example so that each iteration of these stages is executed sequentially. The timing belonging to the chromosome's configuration is visualized in the lower right corner of Figure 6.11. The first two iterations of stage T_1 are executed concurrently, due to the split of this stage. As soon as the results are available, data is communicated to T_2 and both instances of pipeline stage T_3. In the next time frame, iterations 3 and 4 of stage T_1 are executed in parallel to the first and second iterations of T_2 and both sub-tasks of T_3. After four time slots, all six sub-tasks are concurrently executing the statements assigned to their pipeline stage. Depending on task creation and communication costs, this configuration of the genes' values might represent a good solution candidate for the given example regarding executing time. Nevertheless, six tasks are executed in parallel so that many processing units have to execute their work concurrently which consumes significantly more energy if voltage scaling is not applied. Thus, other allocations of the chromosome's decision values might lead to solutions which require less limited resources, as well.

With the presented chromosome structure, horizontal and vertical splits can be encoded very efficiently so that the structure can be used to extract effective pipeline parallelism from embedded applications in a multi-objective aware manner.

6.3.3 Objective Evaluation

To render the calculation of fitness values possible for the Genetic Algorithm, objective evaluation functions have to be provided. Also here, high-level evaluation functions similar to the ones presented in Section 6.2.3 are used.

Different iterations of the same pipeline stage may or may not depend on each other according to the number and position of horizontal and vertical splits (cf. Section 5.3.4). In order to consider these dependencies in the high-level objective models, loop iterations are virtually unrolled for evaluation purposes. Therefore, the objectives are evaluated for all loop iterations. Constants like a task creation overhead or a communication multiplier can be adjusted by the framework to support different target architectures. In addition, values like execution times or the amount of communicated data are extracted automatically by the pre-processing steps presented in Section 3.2 and are annotated to the augmented Program Dependence

Graph of the loop to be parallelized.

6.3.3.1 Objective 1: Execution Time

To evaluate the objective value representing the execution time of a parallelized loop, a high-level model based on the one presented in Section 6.2.3 is used. The model described in the previous section was developed to evaluate task-level parallelism so that it has to be extended to handle different loop iterations and their dependencies for pipeline parallelism correctly. The returned value of the proposed model for pipeline parallelism is equal to the number of cycles of the longest (most critical) execution path of the loop to be parallelized. The following equations describe the evaluation in a formal way. The first component necessary to calculate the overall execution time of the parallelized loop is the execution time of pipeline stage T_i in iteration j. In the proposed model, the execution time of a pipeline stage T_i is uniformly distributed over all loop iterations of the stage. Hence, the execution time $ET(T_i^j)$ of pipeline stage T_i in iteration j is equal to the sum of the execution times $ETN(n)$ of all nodes n which are mapped to T_i, divided by the number of loop iterations LI like defined in Equation 6.9.

$$ET(T_i^j) = \sum_{n \in Nodes(T_i)} \frac{ETN(n)}{LI} \tag{6.9}$$

Based on the execution time of one iteration of a pipeline stage, path costs can be calculated which denote the maximum time until the jth iteration of a stage T_i is finished, including all iterations of its predecessors. The path costs $PC(T_i^j)$ of a pipeline stage T_i in its jth iteration are equal to the sum of T_i's execution costs $ET(T_i^j)$ in iteration j and the path costs of the most expensive predecessor stage T_ℓ^k including the communication costs $CC(T_\ell^k, T_i^j)$ between T_ℓ^k and T_i^j like defined in Equation 6.10. The communication costs can be adapted to different architectures using a platform-dependent communication overhead. The costs of all indirect predecessors are also included, due to the recursive structure of this formula.

$$PC(T_i^j) = ET(T_i^j) + \max\{PC(T_\ell^k) + CC(T_\ell^k, T_i^j) | T_\ell^k \in Pred(T_i^j)\} \tag{6.10}$$

The overall execution costs are composed of the platform-dependent configurable task creation overhead TCO, multiplied by the number of created tasks NT plus the most expensive path costs $PC(T_i^{LI-1})$ of all pipeline stages T_i in their last iterations $LI - 1$ like shown in Equation 6.11.

$$OverallET = TCO * NT + \max\{PC(T_i^{LI-1}) | \forall T_i \in Tasks\} \tag{6.11}$$

The changes made to the original model presented in Section 6.2.3 are mainly that the execution time of a pipeline stage T_i is now distributed over its loop iterations j and that each iteration is considered separately to make the models applicable to pipeline parallelism. The task creation overhead is now added once before the extracted tasks are executed.

6.3.3.2 Objective 2: Energy Consumption

The objective value describing the estimated energy consumption consists of energy costs induced by task creation, communication overhead, and the execution costs of statements which are mapped to the extracted pipeline stages. First, the incoming communication energy costs $ICE(T_i^j)$ for pipeline stage i in iteration j are defined in Equation 6.12. They are equal to the sum of a constant incoming communication energy overhead $ICEO$ plus the number of transferred bytes $\#Bytes(d)$ multiplied by a platform-dependent communication energy factor ICM:

$$ICE(T_i^j) = \sum_{d \in InData(T_i^j)} ICEO + \#Bytes(d) * ICM \tag{6.12}$$

The estimated energy consumption for outgoing communication $OCE(T_i^j)$ is calculated analogously to $ICE(T_i^j)$ like shown in Equation 6.13.

$$OCE(T_i^j) = \sum_{d \in OutData(T_i^j)} OCEO + \#Bytes(d) * OCM \tag{6.13}$$

The energy $E(T_i^j)$ which is necessary to execute iteration j of pipeline stage i contains both, the incoming $(ICE(T_i^j))$ and outgoing $(OCE(T_i^j))$ communication energy costs, increased by the energy $EN(n)$ which is necessary to execute the statements mapped to pipeline stage T_i. The energy consumption is also uniformly distributed over all loop iterations of the stage like defined in Equation 6.14.

$$E(T_i^j) = ICE(T_i^j) + OCE(T_i^j) + \sum_{n \in Nodes(T_i)} \frac{EN(n)}{LI} \tag{6.14}$$

Finally, the overall estimated energy consumption $OverallEnergy$ includes a constant energy overhead for task creation TCE multiplied by the number of created tasks NT, increased by the energy which is consumed by all pipeline stages $E(T_i^j)$ in each iteration like depicted in Equation 6.15.

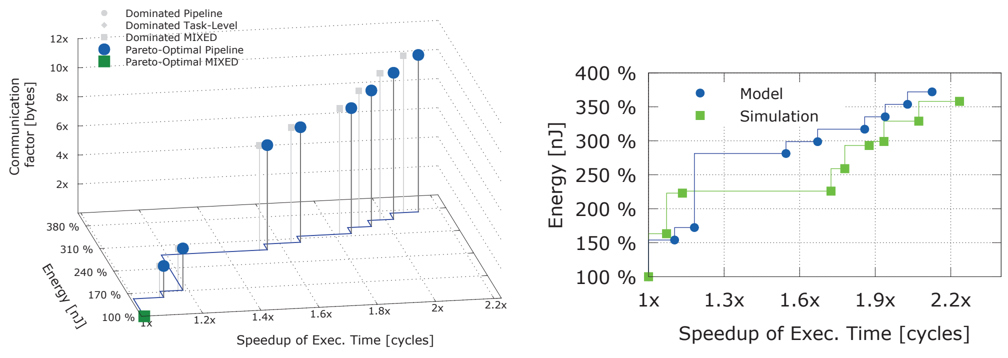
$$OverallEnergy = TCE * NT + \sum_{i \in Tasks} \sum_{j \in \{0..LI-1\}} E(T_i^j) \tag{6.15}$$

Here, the main difference to the original model presented in Section 6.2.3 is that the energy consumption is modeled for each loop iteration separately. The proposed models are later evaluated in Section 6.3.5.

6.3.3.3 Objective 3: Communication Overhead

The evaluation of the communication overhead is based on a simple model which is identical to the original model presented in Section 6.2.3. All communicated data is summed up and multiplied by a platform-dependent communication factor $COSTS$ like shown in Equation 6.16.

$$CommOverhead = \sum_{d \in Comm} \#Bytes(d) * COSTS \tag{6.16}$$



(a) Model-based Results (b) Simulation-based Results

Figure 6.12: Final Parallel Solutions for the Edge Detect Benchmark.

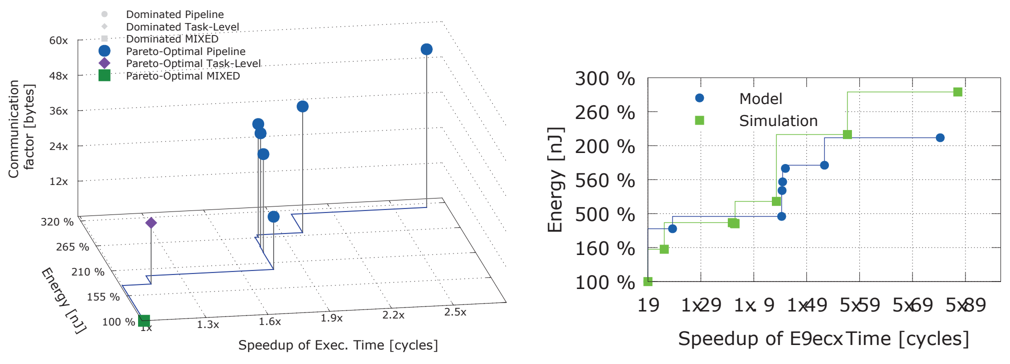
6.3.4 Mutation & Cross-Over

The employed mutation and recombination operations are described in detail in Section 6.2.4 for the GA-based task-level parallelization approach. The basic mutation and recombination operations can also be used here for the extraction of pipeline parallelism.

However, also for the approach presented in this section, the generation of a large number (more than 50%) of invalid solution candidates by state-of-the-art mutation and recombination operations was observed. Thus, smart mutation and recombination algorithms were also developed for the pipeline parallelization approach. With these algorithms, significantly more valid solution candidates are created in a shorter period of time, which increases the solution quality of the developed GA-based parallelization approach. The correction operations avoiding cyclic task dependencies (cf. Section 6.2.4) could, for example, be directly adapted to fix cyclic dependencies between the different pipeline stages. In addition, if too many tasks are extracted as a result of, e.g., too many pipeline stage splits, a succeeding mutation is performed which limits the number of generated sub-tasks for other pipeline stages. By using these smart mutation and cross-over operations, the generation of a large number of invalid solution candidates can be avoided which leads to efficient parallel solutions in a shorter period of time. On average, only 11% of all created solutions were invalid which shows that these correction methods work well. If necessary, the user can, of course, always fall back to the standard mutation and cross-over functions even if a restriction of the solution space excluding efficient solutions was not observed.

6.3.5 Experimental Results

Instead of just presenting results for the pipeline parallelization approach only, a combination with the previously presented multi-objective aware task-level parallelization approach of Section 6.2 is given, here. This highlights the easy incorporation of multiple parallelization approaches into the proposed framework and compares advantages and limitations of the different approaches. The evaluated



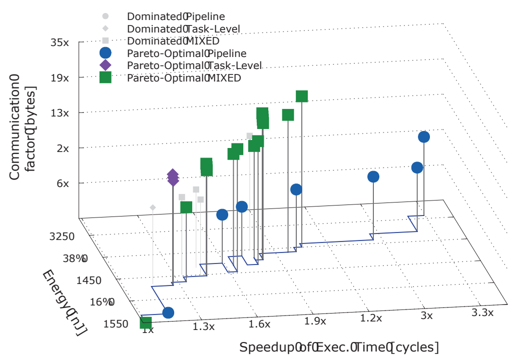
(a) Model-based Results (b) Simulation-based Results

Figure 6.13: Final Parallel Solutions for the Filterbank Benchmark.

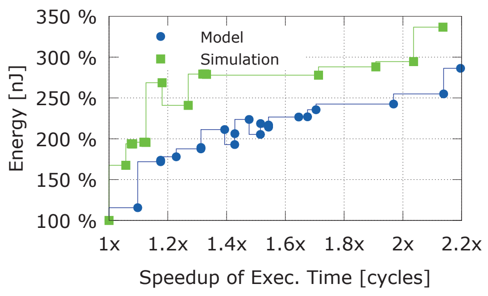
benchmarks are identical to the ones used to evaluate the other parallelization approaches and are part of the UTDSP benchmark suite [Lee13], extended by other meaningful embedded applications like a *jpeg2000* encoder. The target platform is also the same as the one used in Section 6.2.5 and is simulated by the cycle-accurate MPARM simulator [BBB+05] with the MEMSIM [WM06] energy model providing up to four single-core ARM processors. All other components like the operating system and the employed middleware are also the same to facilitate a comparison to the results presented in the previous section.

Figures 6.12(a) - 6.14(a) show detailed results for three of the considered benchmarks. As explained in Section 6.3.3, the current framework optimizes for the three objectives: Speedup (execution time), energy consumption, and the communication overhead introduced by extracted parallelism. These objectives are arranged on the x-, y- and z-axes in three dimensional diagrams, respectively. All axes are relative to the sequential solution which is located at the bottom-left point of the diagrams with a speedup of $1\times$, 100% energy consumption and zero communication overhead. The points of the 3D-diagrams are projected to the x-y-plane for enhanced readability. A solid line marks the front of Pareto-optimal solutions, also projected to the x-y-plane[6]. The communication overhead of different solutions can be compared by the height of the vertical bars. Each diagram contains both, Pareto-optimal and Pareto-dominated points of the final solutions generated by both parallelization approaches. Of course, only the Pareto-optimal ones are returned to the application designer. To be able to compare the efficiency of the new multi-objective aware pipeline parallelization approach presented in this section with the previously presented multi-objective aware task-level parallelization technique, different shapes are used for both types of parallelism. The diagrams also contain solution candidates employing some parallel sections produced by task-level as well as some sections produced by pipeline parallelism. Thus, these points are based on a combination of

[6]The projected Pareto-frontier is not always in a straight echelon form due to the third objective. Even if a solution is worse in execution time and energy consumption, it may be added to the front of Pareto-optimal solutions if it adds less communication overhead.



(a) Model-based Results



(b) Simulation-based Results

Figure 6.14: Final Parallel Solutions for the Spectral Benchmark.

both approaches and are labeled as a MIXED solution.

By analyzing the results for the three benchmarks presented in the figures, one can see that the number of Pareto-optimal solutions returned to the application designer ranges from 8 up to 25 solutions. This highlights the large optimization potential for the different objectives considered. The solution with the highest speedup for, e.g., the *filterbank* benchmark (cf. Figure 6.13(a)) reduces the execution time by a factor of nearly 2.7×. Even if this solution drastically reduces the execution time of the application, it requires the highest communication overhead. In addition, all cores of the platform are executing threads in parallel which increases the energy consumption of the system to around 320% compared to the sequential solution. If the application designer knows that, e.g., a speedup of 1.9× is sufficient to meet the imposed timing requirements a solution can be chosen exploiting less extracted parallelism. Some of the cores can then be switched into idle mode for some time or a platform with less processing units can be chosen. This reduces the energy consumption to less than 270% for this solution of the *filterbank* benchmark. Hence, it is much more efficient compared to 320% energy consumption for the solution with the highest speedup. The amount of inserted communication is also reduced. The solution with a speedup of 1.7× even reduces the energy consumption from around 320% (for 2.7× speedup) to 180%, which highlights the various trade-offs of the proposed multi-objective aware approaches. These observations can be made for the other evaluated benchmarks, as well.

As already observed for the ILP-based single-objective aware pipeline parallelization approach (cf. Section 5.3), pipeline parallelism is able to generate solutions with a high speedup for many embedded applications. When looking at the results of the multi-objective aware approaches presented here, this observation can also be confirmed. The solutions shown in Figures 6.12(a) - 6.14(a) providing the highest speedup for the considered applications are always generated by pure pipeline parallelism (blue circles). However, these solutions also consume the highest amount of energy. The available trade-offs between the different parallelization types can be seen best in Figure 6.14(a) which shows the extracted solution candidates for the *spectral* benchmark. Here, three of four solutions (purple diamonds) with a speedup of less than 1.2× are generated by the previously presented multi-objective aware task-level parallelization approach (cf. Section 6.2). Even though the speedup is not as high as the speedup of the other approaches, only a small increase in energy consumption was observed for these solutions. The combination of task-level and pipeline parallelism (green squares) extracted 14 solutions with speedups between 1.2× and 1.7×. Thus, the energy consumption is slightly increased but is at least lower than for solutions generated by pure pipeline parallelism. Finally, the solution candidates with speedups of more than 1.7× are exclusively extracted by the new pipeline parallelization approach. As can be seen in the figure, other Pareto-dominated solutions with task-level parallelism, pipeline parallelism and also a mixture of both techniques were generated but not returned to the application designer. Other benchmarks like, e.g., the *edge detect* benchmark shown in Figure 6.12(a) profit even more from the new pipeline parallelization approach even though the results are close to the ones presented in Section 6.2.5.

6.3.5.1 Evaluation of High-Level Objective Models

The solution candidates presented in the 3D-diagrams of Figures 6.12(a) - 6.14(a) are based on the high-level models presented in Section 6.3.3. Therefore, Figures 6.12(b) - 6.14(b) depict results obtained by simulation on the MPARM platform with the MEMSIM energy model. These 2D-diagrams compare speedup and energy consumption of the extracted solution candidates. The communication overhead cannot be measured by the simulator so that it is skipped, here. In all three diagrams, the trend of increasing execution time and energy consumption is close enough between model-based evaluation and simulation so that the models are considered sufficiently accurate to deliberate whether it may be beneficial to extract parallelism at a certain point of the application.

6.3.5.2 Additional Results & Statistics

Summarized results and additional statistics of the GA-based approaches are shown in Table 6.2. The table contains information about all benchmarks shown in the 3D-diagrams and also about further evaluated applications. The columns contain information about the time in minutes which was necessary to parallelize the applications with the combination of the task-level and pipeline parallelization approaches (*Time*), the number of processed nodes (*#N*), the number of generated populations (*#Pop*), the overall number of generated and evaluated individuals (*#Ind*), the number of mutated (*#Mut*) and recombined (*#Cross*) individuals and the number of offered Pareto-optimal solutions (S) returned to the application designer. The numbers in the last columns depict how many Pareto-optimal solutions were generated by the task-level (*TL*) and pipeline parallelization (*PL*) approaches as well as the number of solutions generated by a combination of both approaches (*MI*). The number of individuals and populations used to parallelize a node is determined dynamically, based on the number of child nodes. Therefore, nodes with

Benchmark	Time[7]	#N	#Pop	#Ind	#Mut	#Cross	S	TL	PL	MI
adpcm enc.	01:04	36	1,520	151,049	28,154	98,766	5	2	2	1
bound. value	01:11	12	644	83,331	15,804	54,032	4	0	3	1
compress	14:31	336	10,444	821,854	161,617	608,250	5	0	4	1
edge detect	04:48	105	2,872	196,720	38,125	137,788	9	0	8	1
filterbank	02:39	7	412	51,035	15,005	136,485	8	1	6	1
fir 256 64	00:39	13	388	29,607	5,863	22,317	4	0	3	1
iir 4 64	14:35	13	852	103,294	21,298	92,830	4	0	3	1
jpeg2000	04:49	62	2,868	313,047	62,630	231,390	45	0	27	18
latnrm 32 64	01:34	17	636	53,462	11,931	46,358	4	0	3	1
mult 10 10	02:45	36	1,060	70,442	14,984	60,399	4	0	3	1
spectral	03:04	51	2,260	211,023	41,667	160,477	25	3	7	15
average	04:41	63	2,178	189,533	37,916	149,917	11	1	6	4

Table 6.2: Evaluation of Combined GA-based Parallelization Approaches for Homogeneous MPSoCs

a smaller search space are processed much faster. The numbers of populations, individuals, mutations, etc., shown are summed up over all parallelized nodes and may marginally differ between different tool flow executions due to random decisions taken by Genetic Algorithms.

Most Pareto-optimal solutions are created by the new multi-objective aware pipeline parallelization approach presented in this section (cf. column PL). Nevertheless, some benchmarks also profit from the previously presented task-level parallelization approach and the combination of both approaches. For example, 25 Pareto-optimal solutions are returned to the application designer for the *spectral* benchmark, like also shown in Figure 6.14(a). Three of these 25 solutions were generated by the task-level approach while seven solutions are purely based on extracted pipeline parallelism. Nevertheless, 15 solutions were generated by the combined approach containing parallel sections with both, task-level and pipeline parallelism. This shows that both approaches and also their combination provide meaningful solutions optimizing at least one objective of the Pareto-space.

The table also presents some statistics of the employed Genetic Algorithm and the time which was necessary to parallelize the application with both approaches. For the *jpeg2000* encoder, for example, more than 300,000 individuals were created by mutation and recombination. The whole parallelization approach took less than five minutes, which correlates to less than a millisecond to mutate or recombine and also evaluate one of the individuals. Otherwise, it would not be possible to generate such a huge amount of solution candidates, which would drastically reduce the quality of the solutions generated by the Genetic Algorithm.

To summarize, the following results could be confirmed by the evaluation:

1. A large optimization potential is exploited since multiple objectives are considered at the same time in the parallelization process.

[7]Time format MM:SS, measured on an AMD Opteron core running at 2.4 GHz.

2. The novel multi-objective aware pipeline parallelization approach extracts, in general, the most efficient parallel solutions regarding speedup of execution time for the considered embedded applications.

3. Solutions generated by the task-level parallelization approach are less efficient regarding speedup for many embedded applications, but they use less energy.

4. The combination of task-level and pipeline parallelism produces highly beneficial solutions providing good trade-offs between high speedups of pipeline parallelism and less energy consumption of task-level parallelism.

The results have shown that the multi-objective aware extraction of pipeline parallelism improves the quality of the solutions returned by the existing multi-objective aware parallelization approach. Moreover, the combination with task-level parallelism also extends the space of Pareto-optimal solutions.

6.4 Summary

This chapter presented two multi-objective aware parallelization approaches which are well applicable to homogeneous embedded MPSoCs. While most state-of-the-art parallelization approaches consider the extraction of speedup as their only optimization objective, the approaches presented in this chapter are able to trade-off different objectives directly in the parallelization process. The performed evaluations have, for example, shown that it is possible to save a significant amount of energy if the extracted parallelism is reduced so that some of the cores can be put into idle mode or a platform providing fewer cores can be used.

It could further be shown that both, the newly presented task-level and pipeline parallelization approaches are able to extract efficient solution candidates leading to either high speedups, low energy consumption, low communication overhead, or solutions with good trade-offs between the different objectives. Furthermore, also a combination of the presented task-level and pipeline parallelization approaches contributed additional solutions to the space of Pareto-optimal points. These solution candidates represent trade-offs between the advantages and limitations of both presented parallelization approaches. It is easy to integrate additional objectives to the GA-based parallelization approach since only a high-level objective evaluation function has to be provided. As a consequence, other objectives like memory consumption could easily be integrated for future research work, as well.

To conclude, the following results could be achieved by the multi-objective aware parallelization approaches presented in this chapter:

1. Creation and integration of a multi-objective aware task-level parallelization approach.

2. Creation and integration of a multi-objective aware pipeline parallelization approach.

3. Combination of task-level, data-level[8], and pipeline parallelization approaches, which can be executed separately or in a combined fashion.

4. Usage of high-level evaluation functions which are accurate enough to estimate whether parallel execution may lead to increased performance for the considered objectives. Moreover, the model-based evaluation is fast enough to evaluate a large number of solution candidates.

5. The combination of the different multi-objective aware parallelization approaches produced highly beneficial solutions providing either high speedups, low energy consumption, low communication overhead, or also good trade-offs between the considered objectives.

[8]DoAll parallelism can be extracted as a special case from the pipeline parallelization approach.

Chapter 7

Single-Objective Parallelization for Heterogeneous MPSoCs

Contents

The previous two chapters presented efficient parallelization approaches integrating high-level cost models into Integer Linear Programming (ILP) and Genetic Algorithm (GA)-based parallelism extraction techniques. By combining task-level and pipeline parallelization approaches into single and also multi-objective aware contexts, solutions could be extracted which are well tailored towards specific requirements imposed by modern homogeneous embedded MPSoCs. However, the design of current state-of-the-art MPSoCs moves from traditional homogeneous multi-core architectures towards heterogeneity. In heterogeneous MPSoCs, different kinds of processors are combined on one die to tackle problems concerning, among others, processing speed, energy consumption, and heat dissipation arising if the same kind of processing unit is replicated multiple times by homogeneous MPSoCs. These heterogeneous systems often combine general purpose cores with processing units which are highly optimized for specific use cases, like network or digital signal processors. Parts of the application can be executed on these specialized cores which can significantly reduce, e.g, execution time and the system's energy consumption simultaneously.

Even though these fully heterogeneous architectures provide high performance at a significantly lower energy consumption compared to homogeneous MPSoCs, their

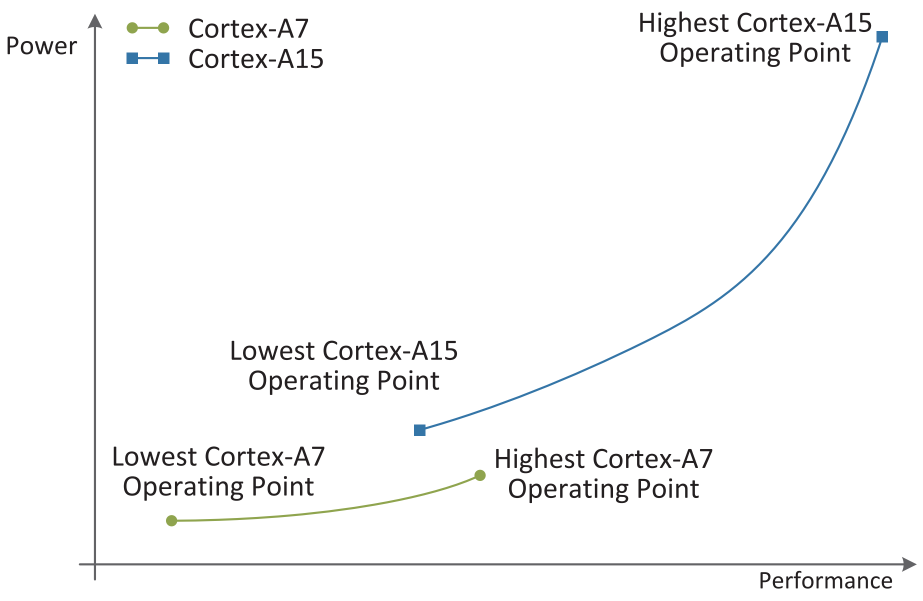


Figure 7.1: Performance vs. Energy of big.LITTLE Architecture [Pet13]

optimization potential is often not fully exploited. The main reason is certainly the high complexity of such systems. In most cases, the processing units employed behave completely differently when executing the same statements of the application. In addition, most of these cores are not binary compatible so that application code must be re-written or at least re-compiled to enable exploitation of the specialized processing units. This often leads to parallelized versions of the targeted application executing only pre-compiled parts of it (like libraries) on these specialized processing units. This wastes a lot of optimization potential since these cores may also be useful for multiple parts of the considered application.

Since these fully heterogeneous platforms are often too complex to be fully exploited, an interesting design pattern was presented by Kumar et al. several years ago in [KTR+04]. They proposed a heterogeneous same instruction set architecture (same-ISA) multicore platform which combines different processing units supporting binary compatibility. This has the advantage that the application has to be compiled only once, independent of the executing processing unit, which significantly eases the step of porting and mapping an application to such an MPSoC platform. This idea has recently caught on in the industry resulting in designs like ARM's big.LITTLE architecture [Pet13] combining multiple Cortex-A15 and Cortex-A7 cores on one die. The benefit of such a platform is depicted in Figure 7.1. While the (little) Cortex-A7 cores provide only low performance, they are efficient regarding energy consumption due to their simple processor pipeline. In contrast, the (big) Cortex-A15 cores provide a large amount of processing power at the cost of the system's energy consumption. By combining the benefits of both processor types, a binary compatible heterogeneous platform providing high-performance, low energy consumption and many trade-offs between these objectives can be exploited.

However, independently of whether the heterogeneous architecture is binary compatible or not, new problems arise if sequentially written applications are to be mapped onto the cores of these platforms. On homogeneous platforms, extracting parallelism has already shown to be time-consuming and error-prone, prompting the development of automatic parallelization approaches like the ones presented in Chapters 5 and 6. For heterogeneous systems, the task of parallelizing a given application becomes even more complex than in the homogeneous case, since the execution time required for a given section of code differs depending on the executing processing unit. Hence, manual parallelization of a sequential application and balancing of tasks becomes a nearly infeasible problem for developers. As a consequence, efficient automatic parallelization becomes indispensable when targeting heterogeneous platforms.

Since the ILP-based approaches presented in Chapter 5 have shown to be efficient for extracting and balancing tasks for embedded homogeneous architectures, they are used as a starting point for new extraction techniques tailored towards heterogeneous MPSoCs. The Augmented Hierarchical Task Graph (AHTG) with its divide-and-conquer-based approach (cf. Chapter 4) has already been useful to make the complexity of the parallelization problem manageable for homogeneous architectures and is now indispensable for heterogeneous ones. Execution times, as well as other objectives, may vary depending on the processing unit executing a given piece of code for heterogeneous MPSoCs. This significantly increases the solution space of the parallelization problem and makes load-balancing an even more challenging problem. But if such performance variances can be integrated into the parallelization extraction step, optimized tasks for specific processing units can be extracted creating well-balanced solutions even for heterogeneous architectures. While this chapter presents new ILP-based single-objective aware parallelization techniques for heterogeneous architectures, the succeeding Chapter 8 will further present approaches extracting parallelism for such platforms in a multi-objective aware manner.

The rest of this chapter is structured as follows: The first ILP-based parallelization approach for heterogeneous architectures extracting task-level parallelism is presented in Section 7.1. Afterwards, Section 7.2 presents the second ILP-based parallelization technique for heterogeneous architectures, extracting pipeline parallelism. Finally, Section 7.3 summarizes the approaches presented in this chapter and gives directions for future work.

7.1 ILP-based Task-Level Parallelization Approach

Task-level parallelism (cf. Section 5.2.1) could be used to efficiently parallelize sequentially written application for homogeneous MPSoCs so that it should be considered for extraction techniques focusing heterogeneous MPSoCs as well. The approach presented in this section uses the ILP-based task-level parallelization technique of Section 5.2 as a starting point to extract the same kind of parallelism with its employed fork-join model (cf. Section 5.2.3). It also operates on the Augmented Hierarchical Task Graph with its global divide-and-conquer-based parallelization approach to extract and combine parallelism with different granularities (cf. Chapter 4). Hence, small groups of statements, different loop(-nest)s or also function calls may be executed in parallel. The new main challenge that has to be coped with for heterogeneous MPSoCs is the balancing of the extracted tasks for processing units with varying performance characteristics. Since the heterogeneous approach presented here is based on the homogeneous one described in Section 5.2, it is recommended to read this section first.

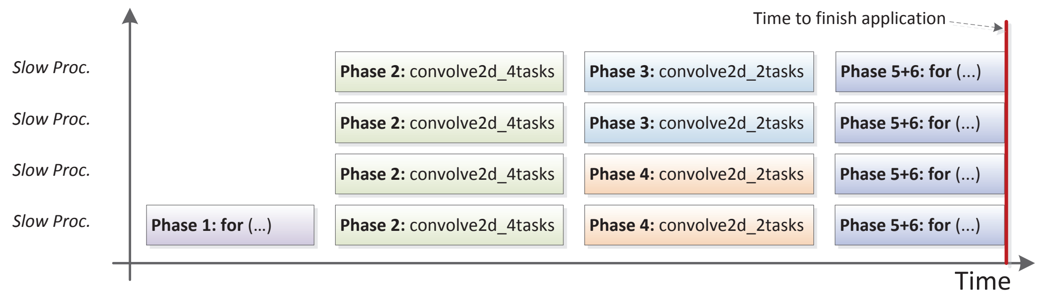
The rest of this section is structured as follows: First, Section 7.1.1 demonstrates the impact of the balancing problem on heterogeneous architectures by a small example before the integration of the presented parallelization technique into the global parallelization approach is described in Section 7.1.2. Afterwards, the employed ILP-based parallelization approach is described in Section 7.1.3 extended by a simple loop-parallelization approach in Section 7.1.4. Finally, the newly presented heterogeneous approach is evaluated and compared against the homogeneous task-level parallelization approach in Section 7.1.5.

7.1.1 Motivating Example

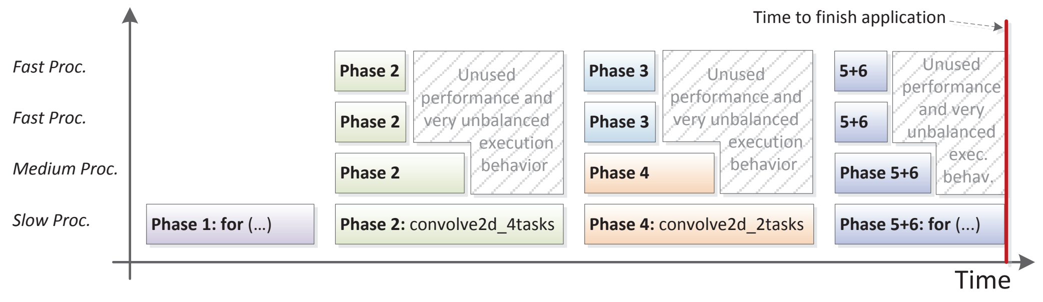
An approach which is able to extract task-level parallelism for homogeneous embedded MPSoCs was already presented in Section 5.2. As shown in the motivating example of that section, the extracted tasks could be balanced so that all tasks belonging to the same parallel section finish nearly at the same time. The resulting partitioning of this example is once again shown in Figure 7.2(a). There, six phases are grouped into four parallel sections (i.e., four blocks of concurrently executed tasks) and all tasks finish nearly at the same time.

The new challenge arising in the case of heterogeneous MPSoCs is that the tasks' execution times may differ depending on the executing processing unit. Figure 7.2(b) shows what may happen if the same solution would be mapped to a heterogeneous architecture with two fast, one slow processor, and one processor in between (medium). In this scenario, three of the four processing units have to wait for a long time since the tasks of the different parallel sections are not well-balanced. Thus, a large amount of processing power is wasted. Even though three faster processing units were added to the target platform as accelerators, the overall execution time could not be reduced since all faster processing units have to wait for the slowest one in each parallel section. Most existing parallelization approaches do not consider these performance variances of the available processing units which may result in solutions like the one shown in Figure 7.2(b).

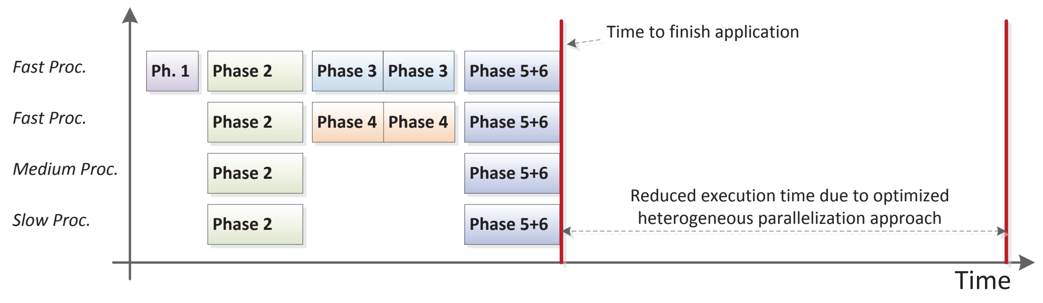
To find a better solution for this problem, the new task-level parallelization approach presented in this section considers these performance differences and combines task extraction with a pre-mapping of tasks to processor classes. These processor classes represent identical processing units of the heterogeneous architecture. In this way, tasks can be optimized for specific processing units directly in the


Time to finish application
Slow Proc.
Slow Proc.
Slow Proc.
Slow Proc.
Phase 1: for (...)
Phase 2: convolve2d_4tasks
Phase 3: convolve2d_2tasks
Phase 4: convolve2d_2tasks
Phase 5+6: for (...)
Time

(a) Execution on a Homogeneous MPSoC


Time to finish application
Fast Proc.
Fast Proc.
Medium Proc.
Slow Proc.
Phase 1: for (...)
Phase 2
Phase 2: convolve2d_4tasks
Unused performance and very unbalanced execution behavior
Phase 3
Phase 4
Phase 4: convolve2d_2tasks
5+6
Phase 5+6
Phase 5+6: for (...)
Unused performance and very unbalanced exec. behav.
Time

(b) Execution of the Same Solution on a Heterogeneous MPSoC


Fast Proc.
Fast Proc.
Medium Proc.
Slow Proc.
Ph. 1
Phase 2
Phase 3
Phase 4
Phase 5+6
Time to finish application
Reduced execution time due to optimized heterogeneous parallelization approach
Time

(c) Execution of an Optimized Solution for a Heterogeneous MPSoC

Figure 7.2: Possible Parallel Solutions of the Edge Detect Benchmark (cf. Section 5.2.1)

parallelization process which enables well-balanced solutions even for heterogeneous architectures. An exemplary solution depicting the possibilities of the new approach is shown in Figure 7.2(c). There, phase 1 of the application is first moved to one of the fastest processing units so that the first parallel section is finished earlier. The second and the final parallel sections, executing phase 2 and phases 5 and 6, are executing different iterations of the sections' loops in parallel. Here, a balancing of iterations with respect to the provided performance is employed. Heavier workloads, i.e. tasks with more iterations, are mapped to the two fastest processing units while the slower ones are allocated with lighter workloads, respectively. In the third parallel section of the example shown in Figure 7.2(c), the algorithm may decide to put the slow and the medium processing units into idle mode since their allocation would decrease the extracted speedup. Instead, both parts of phase 3 and phase 4 are allocated to one of the fastest processing units each. This solution is more than twice as fast as the solution shown in Figure 7.2(b) which significantly increases the application's performance.

This example has shown that it is indispensable to take performance variances of the available processing units into account if applications should be efficiently mapped onto heterogeneous MPSoCs. To achieve this, different estimated execution times are annotated to the AHTG for each processor class. In addition, a pre-mapping of tasks to processing units, grouped to processor classes, is performed to ensure that the extracted tasks are mapped to the processing units for which they are optimized. These steps are integrated into the ILP-based task-level parallelization approach in the remainder of this section.

7.1.2 Integration into the Global Parallelization Approach

The integration of the new ILP-based task-level parallelization approach tailored towards heterogeneous MPSoCs into the global parallelization framework (cf. Section 4.2) is shown in Algorithm 9.

Algorithm 9 Integration of Het. ILP-based Task-Level Parallelization Approach

```
 1: // Called bottom-up hierarchically by Algorithm 4 in line 18 on page 56
 2: function ExtractHetTLP(Node $n$, Platform $pf$, int $maxTasks$)
 3:     // This function is only applicable to hierarchical nodes.
 4:     if IsNotHierarchicalNode($n$) then
 5:         return $\emptyset$
 6:     end if
 7:     // If n represents an independent loop, use the simple loop parallelizer.
 8:     if $n.getStmt().isLoopStmt() \wedge$ IsLoopIndependent($n$) then
 9:         $solutions \leftarrow$ SimpleParallelizer($n, pf, maxTasks$)
10:         return $solutions$
11:     end if
12:     // Otherwise, apply the ILP-based approach.
13:     for all $seqPC \in pf.getProcClasses()$ do
14:         $i \leftarrow maxTasks$
15:         while $i >= 2$ do
16:             $result \leftarrow$ HetILPTaskLevelParallelizer($n, seqPC, pf, i$)
17:             $solutions \leftarrow solutions \cup \{result\}$
18:             $i \leftarrow$ NumberOfTasks($result$) $- 1$
19:         end while
20:     end for
21:     return $solutions$
22: end function
```

The function ExtractHetTLP is executed by the global parallelization algorithm (cf. Algorithm 4 on page 56) as soon as all child nodes deeper in the hierarchy are processed. The first lines are identical to the integration of the homogeneous task-level parallelization approach depicted in Algorithm 5. The algorithm first determines in lines 4-6 whether the currently processed node n is not a hierarchical node since other node types (e.g. simple nodes) are not processed by this approach.

Since the complexity of the heterogeneous ILP-based task-level parallelization approach is high, the algorithm first determines whether the currently processed node n represents a loop statement. If this is true and the iterations are free of loop-carried dependencies, a less complex approach described in Section 7.1.4 can be applied in lines 8-11 to extract efficient solutions for these special cases. The solutions extracted by a call to SimpleParallelizer are finally returned as solution candidates of this node.

If node n represents a non-loop statement or the loop contains loop-carried dependencies, the sophisticated ILP-based task-level parallelization approach presented in Section 7.1.3 is repeatedly called in line 16 for different processor classes and changing upper task boundaries. In this way, several solutions with different processing unit allocations are extracted to provide the parallelization process upwards in the hierarchy with flexibility for extracting and combining parallelism with different granularities. All extracted results are collected and finally returned as solution candidates for node n in line 21.

7.1.3 ILP-based Parallelization Approach

The main goals that have to be covered by the new ILP-based task-level parallelization approach for heterogeneous embedded MPSoCs are summarized in the following:

I) Map statements of direct child nodes into newly extracted, disjunctive tasks to reduce the overall execution time by parallel execution.

II) Combine newly extracted tasks with tasks which were extracted deeper in the hierarchy, if such a solution increases the overall performance (Parallel Set Mapping).

III) Keep track of dependencies which may change if child nodes representing statements are moved from one task to another one.

IV) Minimize the overall execution time by taking task creation and communication overhead as well as task execution costs depending on the mapped processor class into account.

V) Create a mapping of tasks to processor classes of heterogeneous MPSoCs to take care that solutions are well balanced, even for architectures containing processing units with differing performance characteristics.

Goals (I)-(IV) are already covered by the homogeneous ILP-based task-level parallelization approach presented in Section 5.2.4. The referenced section defines and describes several decision variables and constraints enabling the extraction of predecessor and successor relationships within an ILP system. It further determines the different execution paths of the node to be parallelized in Equations 5.3 - 5.17. By using these equations, it is possible to reduce the execution time of the currently processed hierarchical node by moving its direct child nodes to concurrently executed, well-balanced tasks. The ILP solver also has the option to combine newly extracted tasks with parallelism which was found deeper in the hierarchy. Most of these equations can be re-used to build the new heterogeneous task-level parallelization approach for heterogeneous architectures on top of the existing one. Only the Parallel Solution Candidate Constraint (cf. Equations 5.5 - 5.6 in Section 5.2.4.2) and the Task Execution Costs Constraint (cf. Equation 5.9 in Section 5.2.4.4) have to be adapted to distinguish between varying execution times depending on the mapped processing unit for heterogeneous MPSoCs. In addition, the constraints limiting the number of extractable tasks defined in Equations 5.12 - 5.15 are useless in this form for heterogeneous architectures. All other equations which map nodes to tasks and describe execution paths as well as the objective function can be re-used without being changed. To summarize, the eight Equations 5.3 - 5.4, 5.7 - 5.8, 5.10 - 5.11, and 5.16 - 5.17 are re-used without modifications, extended by ten new equations, defined in the remainder of this section.

In the following, decision variables are written in lower case letters, sets start with a capital letter, and constants contain exclusively capital letters. Indices n and o are used for child nodes of the node to be parallelized, t and u represent indices for tasks while c represents a processor class. The most important variables presented in Section 5.2.4 are summarized here once again:

- $\boldsymbol{x_n^t = 1}$ if node n is mapped to task t (Section 5.2.4.1)
- $\boldsymbol{p_{n,s} = 1}$ if parallel solution s of child node n is chosen (re-defined in Section 7.1.3.1)
- $\boldsymbol{pred^{t,u} = 1}$ if task t is a predecessor of task u (Section 5.2.4.3)
- $\boldsymbol{cost^t}$ execution costs of task t (re-defined in Section 7.1.3.2)
- $\boldsymbol{accumcost^t}$ path execution costs of t including all its predecessors (Section 5.2.4.5)

A graphical representation of all equations is also given in Figures 7.3 - 7.8 visualizing the decision variables and constraints used. The sub-figures have the same titles as the corresponding subsections. To avoid repetition, the re-used and unchanged parts of the homogeneous ILP are not repeated here. The complete ILP formulation containing all decision variables and constraints used can also be found in the corresponding publication of this approach [CEN+13c].

7.1.3.1 Parallel Solution Candidate Constraint

The largest impact on the tasks' execution times is caused by the execution times of the child nodes mapped to the corresponding tasks. Profitable parallel solution candidates were created and collected in a so-called parallel set for each child node by the bottom-up parallelization approach (cf. Section 4.2.2). This set can contain both

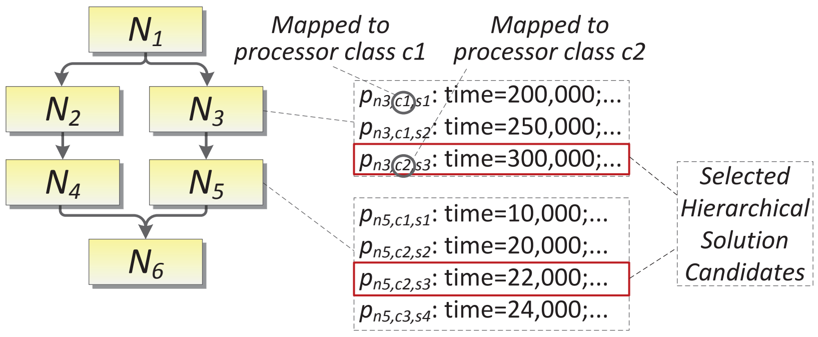

N1
N2
N3
N4
N5
N6
Mapped to processor class c1
Mapped to processor class c2
p_{n3,c1,s1}: time=200,000;...
p_{n3,c1,s2}: time=250,000;...
p_{n3,c2,s3}: time=300,000;...
p_{n5,c1,s1}: time=10,000;...
p_{n5,c2,s2}: time=20,000;...
p_{n5,c2,s3}: time=22,000;...
p_{n5,c3,s4}: time=24,000;...
Selected Hierarchical Solution Candidates

Figure 7.3: Parallel Solution Candidate Constraint

sequential as well as parallelized solutions containing parallelism which was found deeper in the nodes' hierarchy. The algorithm has to choose one of these solutions for each child node (Goal (II)) to combine new tasks with previously extracted ones.

In the homogeneous case, the execution times are independent of the executing processing unit since all cores behave identically. As already motivated, this assumption would lead to poor results for heterogeneous architectures since the extracted tasks would operate in a highly unbalanced execution behavior due to the different performance characteristics of the employed processing units. Therefore, each solution is now labeled with a specific processor class which has to be used to execute the selected parallel solution candidate of node n to reach the estimated performance characteristics. In addition, an internal task-to-processor class mapping is also newly integrated for solution candidates running tasks in parallel. In the example shown in Figure 7.3, three solution candidates exist for child node n_3. The first two ones assume an execution of this node on processor class $c1$ while the last one assumes an execution of this node on processor class c_2. If the last solution would be selected for node n_3, the execution would take 300,000 time units on processor class c_2. The different execution times were estimated by the high-level models integrated into the ILP-based parallelization approach in the parallelism extraction step for the corresponding child node.

Equation 7.1 defines variable $p_{n,c,s}$ which was extended to incorporate the new processor class dimension. The variable evaluates to 1 if parallel solution s of node n executed on processor class c is chosen. This new variable replaces $p_{n,s}$ defined by the homogeneous ILP-based approach.

$$p_{n,c,s} = \begin{cases} 1, & \text{if parallel solution } s \text{ of child node } n \\ & \text{executed on processor class } c \text{ is chosen} \\ 0, & \text{otherwise} \end{cases} \tag{7.1}$$

Equation 7.2 takes care that exactly one hierarchical parallel solution candidate

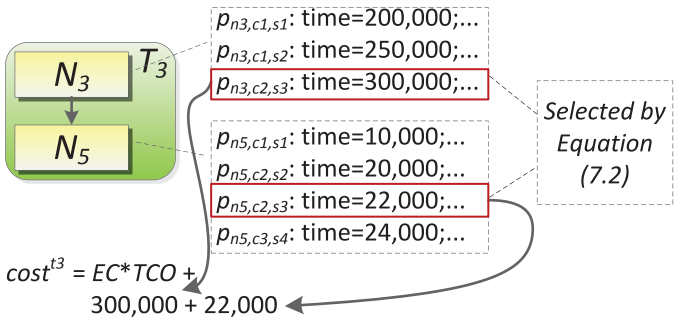

N3
N5
T3
pn3,c1,s1: time=200,000;...
pn3,c1,s2: time=250,000;...
pn3,c2,s3: time=300,000;...
pn5,c1,s1: time=10,000;...
pn5,c2,s2: time=20,000;...
pn5,c2,s3: time=22,000;...
pn5,c3,s4: time=24,000;...
Selected by Equation (7.2)
cost^t3 = EC*TCO +
300,000 + 22,000

Figure 7.4: Task Execution Costs Constraint

is chosen for each child node as enforced by the global parallelization algorithm[1].

$$\forall n \in Nodes : \sum_{c \in ProcClasses} \sum_{s \in Solutions_{n,c}} p_{n,c,s} = 1 \tag{7.2}$$

7.1.3.2 Task Execution Costs Constraint

The execution costs $cost^t$ of task t depend on the chosen parallel solution candidates of its mapped child nodes. Since the definition of $p_{n,c,s}$ was changed in the previous two equations, the calculation of the task costs must also be updated to be aware of heterogeneous architectures with varying performance characteristics of the available processing units. Therefore, the costs for each task are now calculated like defined in Equation 7.3 and shown in Figure 7.4.

$$\forall t \in Tasks : cost^t = EC * TCO + \sum_{n \in Nodes} \sum_{c \in ProcClasses} \sum_{s \in Solutions_{n,c}} (x_n^t \wedge p_{n,c,s}) * COSTS_{n,c,s} \tag{7.3}$$

Equation 7.3 includes decision variable $p_{n,c,s}$ which selects one solution candidate for each child node involving different execution costs $COSTS_{n,c,s}$[2] depending on the mapped processor class c and the number of extracted tasks deeper in the hierarchy. The execution costs $cost^t$ of task t consist of a configurable task creation overhead TCO multiplied by the execution count EC. This overhead is increased by the execution costs $COSTS_{n,c,s}$ of all nodes n which are executed on processor class c and mapped to task t depending on the chosen parallel solution candidate $p_{n,c,s}$. Thus, $cost^t$ contains all execution costs of task t by considering the mapped processor class.

The variable $cost^t$ is further embedded in the calculation of the path costs so that the different execution times depending on the selected processor class are

[1]Such a solution can be guaranteed since at least one sequential solution candidate exists for each processor class denoting the sequential execution.

[2]Variables $COSTS_{n,c,s}$ were calculated deeper in the hierarchy and are thus constants in the parallel configurations.

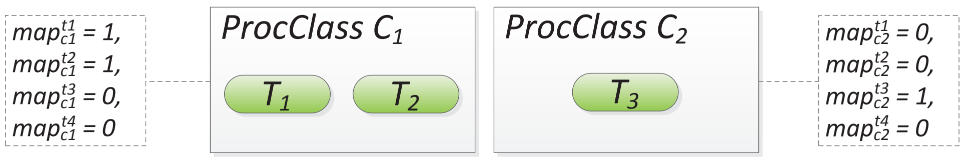


Figure 7.5: Mapping of Tasks to Processor Classes Constraint

automatically integrated into the path calculation and therefore the minimization of the overall execution time. Hence, the consideration of differing performance characteristics of the available processing units could be achieved with only two small adjustments of the previously presented homogeneous ILP systems. However, to be able to extract highly optimized tasks for heterogeneous MPSoCs, a pre-mapping of tasks to processor classes has to be integrated into the new ILP formulation which was not necessary in the homogeneous case. Therefore, the rest of this section presents the necessary decision variables and constraints used to achieve this.

7.1.3.3 Mapping of Tasks to Processor Classes Constraint

Up to now, newly extracted tasks are not mapped to any processor class. This has not been necessary for homogeneous architectures but has a huge impact on the execution time for heterogeneous ones. Therefore, the presented approach of this section combines the extraction of parallelism with a mapping of tasks to processor classes representing identical processing units of the targeted heterogeneous architecture. This enables the extraction of well-balanced tasks, which are optimized for a given processor class like demanded by Goal (V). A new decision variable map_c^t is introduced which evaluates to 1 if task t is mapped to processor class c like shown in Figure 7.5 and defined in Equation 7.4.

$$map_c^t = \begin{cases} 1, & \text{if task } t \text{ is mapped to processor class } c \\ 0, & \text{otherwise} \end{cases} \tag{7.4}$$

Each task t has to be mapped to exactly one processor class c, which is ensured by Equation 7.5.

$$\forall t \in Tasks : \sum_{c \in ProcClasses} map_c^t = 1 \tag{7.5}$$

7.1.3.4 Available Processing Units per Processor Class Constraint

By taking advantage of platform information in the task extraction step, it is possible to avoid additional scheduling overhead at runtime. Therefore, each processing unit should either be used by newly extracted tasks or tasks which were extracted deeper in the hierarchy. The number of already allocated processing units of a processor class c, used by the selected hierarchical solution candidates of the child nodes, has to be determined for each task t. The constant $USEDPROCS_{s,c}$ represents the number of already allocated processing units of class c for hierarchical

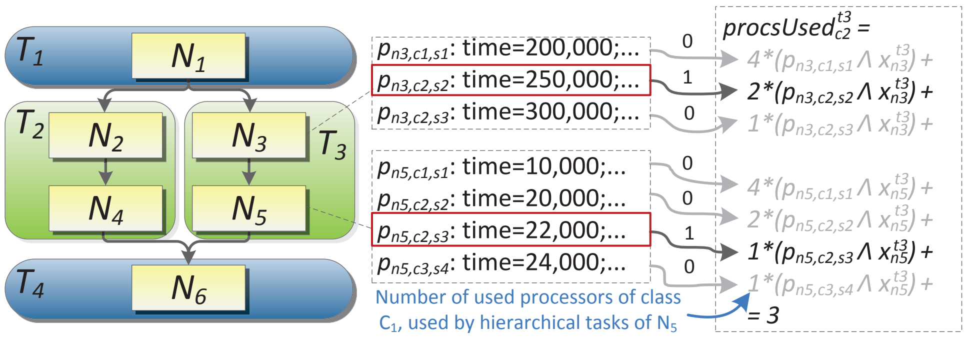

T1
N1
T2
N2
N3
T3
N4
N5
T4
N6
pn3,c1,s1: time=200,000;...
pn3,c2,s2: time=250,000;...
pn3,c2,s3: time=300,000;...
pn5,c1,s1: time=10,000;...
pn5,c2,s2: time=20,000;...
pn5,c2,s3: time=22,000;...
pn5,c3,s4: time=24,000;...
0
1
0
0
0
1
0
procsUsed t3 c2 =
4*(pn3,c1,s1 ∧ x t3 n3) +
2*(pn3,c2,s2 ∧ x t3 n3) +
1*(pn3,c2,s3 ∧ x t3 n3) +
4*(pn5,c1,s1 ∧ x t3 n5) +
2*(pn5,c2,s2 ∧ x t3 n5) +
1*(pn5,c2,s3 ∧ x t3 n5) +
1*(pn5,c3,s4 ∧ x t3 n5) +
= 3
Number of used processors of class C1, used by hierarchical tasks of N5

Figure 7.6: Available Processing Units per Processor Class Constraint

parallel solution candidate s and is a constant for each parallel solution candidate. Equation 7.6 defines variable $procsused_c^t$ which stores the amount of already allocated processing units on the basis of $USEDPROCS_{s,c}$ for all processor classes c, used by the child nodes mapped to task t.

$$\forall c \in ProcClasses : \forall t \in Tasks : \forall n \in Nodes : \forall s \in Solutions_{n,c} :$$
$$procsused_c^t \geq USEDPROCS_{s,c} * (p_{n,c,s} \wedge x_n^t) \tag{7.6}$$

The constant $USEDPROCS_{s,c}$ is only added to the number of used processing units $procsused_c^t$ if the hierarchical solution candidate s was selected for node n ($p_{n,c,s}$) and n is mapped to task t (x_n^t). An example is shown in Figure 7.6. Nodes n_3 and n_5 are mapped to task t_3 so that only their hierarchical solutions are considered. The second hierarchical solution is selected for node n_3, containing two hierarchical tasks allocated to processor class $c2$. For node n_5, the solution with one hierarchical task mapped to processor class $c2$ was selected, so that the number of hierarchical tasks contained in t_3 mapped to processor class $c2$ is three.

With $procsused_c^t$, it is now possible to calculate the amount of processing units which are still available for allocation of newly extracted tasks, like shown in Equation 7.7.

$$\forall c \in ProcClasses : numPC_c = NUMPROCS_c - (\sum_{t \in Tasks} procsused_c^t) \tag{7.7}$$

The constant number of available processing units $NUMPROCS_c$ per processor class c is derived from the provided platform information pf (cf. Algorithm 9).

7.1.3.5 Limit Allocated Processing Units Constraint

So far, all newly extracted tasks can be mapped to the fastest processor class even if not enough processing units of this class for parallel execution are available. Therefore, the number of tasks mapped to this processor class should be limited to be less or equal to the number of available processing units of this class (cf. Figure 7.7). Equation 7.8 ensures that the number of newly extracted tasks t, mapped to processor class c, does not exceed the number of still available processors $numPC_c$ for

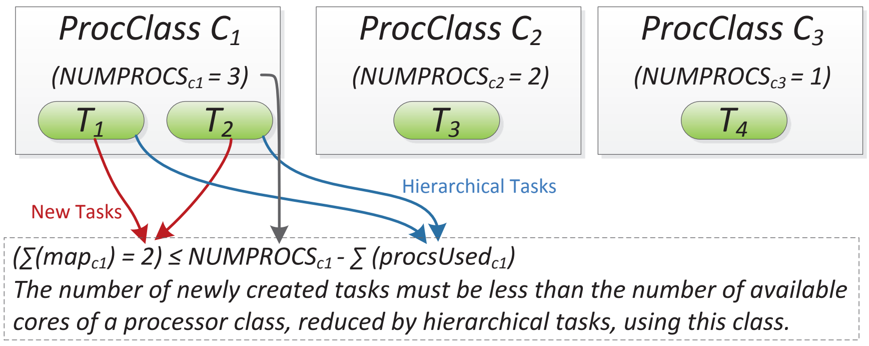

ProcClass C_1
(NUMPROCS$_{c1}$ = 3)
T_1
T_2
ProcClass C_2
(NUMPROCS$_{c2}$ = 2)
T_3
ProcClass C_3
(NUMPROCS$_{c3}$ = 1)
T_4
New Tasks
Hierarchical Tasks
(∑(map$_{c1}$) = 2) ≤ NUMPROCS$_{c1}$ - ∑ (procsUsed$_{c1}$)
The number of newly created tasks must be less than the number of available cores of a processor class, reduced by hierarchical tasks, using this class.

Figure 7.7: Limit Allocated Processing Units Constraint

each processor class c.

$$\forall c \in ProcClasses : \sum_{t \in Tasks} map_c^t \leq numPC_c \tag{7.8}$$

With this constraint, it is finally ensured that the number of newly extracted and combined hierarchical tasks does not exceed the number of available processing units of the different processor classes.

7.1.3.6 Restrict Solution Candidates Constraint

So far, new tasks can be extracted, balanced, and combined with parallelism which was extracted deeper in the hierarchy to reduce the overall execution time of node n. However, one last aspect has to be taken into account to ensure that the solutions extracted by the ILP system are valid. As shown in Figure 7.3 of the Parallel Solution Candidate Constraint, all solution candidates $p_{n,c,s}$ of child nodes n are tagged with a specific processor class c. Thus, the execution time of a solution candidate $p_{n,c,s}$ determined earlier is only valid if node n is mapped to processor class c. Otherwise, the execution time would alter due to the varying performance characteristics of the available processing units. Node n is mapped to processor class c if the task t executing node n is also mapped to processor class c. Therefore, the ILP system must be restricted to choose only one of those solution candidates using the same processor class as the task to which node n is mapped. Equation 7.9 is responsible for this, defining decision variable $nodeOnProcClass_{n,c}$ which evaluates to 1 if node n is executed on processor class c with respect to the node-to-task mapping x_n^t and task-to-processor class mapping map_c^t, defined earlier.

$$\begin{gathered}\forall n \in Nodes : \forall c \in ProcClasses : \\ nodeOnProcClass_{n,c} = \sum_{t \in Tasks} x_n^t \wedge map_c^t\end{gathered} \tag{7.9}$$

Finally, Equation 7.10 takes care that only those hierarchical parallel solution candidates can be chosen which are valid with respect to the task-to-processor class

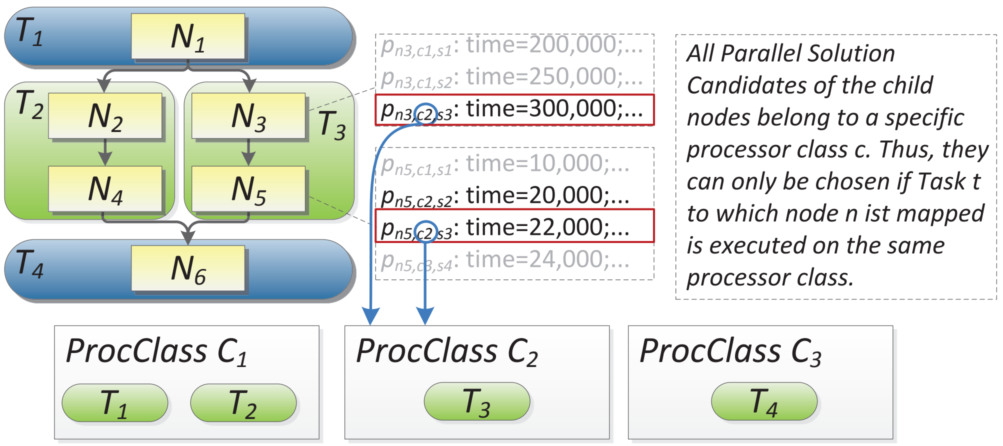

T_1
N_1
T_2
N_2
N_3
T_3
N_4
N_5
T_4
N_6
$p_{n3,c1,s1}$: time=200,000;...
$p_{n3,c1,s2}$: time=250,000;...
$p_{n3,c2,s3}$: time=300,000;...
$p_{n5,c1,s1}$: time=10,000;...
$p_{n5,c2,s2}$: time=20,000;...
$p_{n5,c2,s3}$: time=22,000;...
$p_{n5,c3,s4}$: time=24,000;...
All Parallel Solution Candidates of the child nodes belong to a specific processor class c. Thus, they can only be chosen if Task t to which node n ist mapped is executed on the same processor class.
ProcClass C_1
T_1
T_2
ProcClass C_2
T_3
ProcClass C_3
T_4

Figure 7.8: Restrict Solution Candidates Constraint

mapping. This constraint is also visualized in Figure 7.8.

$$\forall n \in Nodes : \forall c \in ProcClasses :$$
$$\sum_{s \in Solutions_{n,c}} p_{n,c,s} = nodeOnProcClass_{n,c} \qquad (7.10)$$

If the task executing node n is not mapped to processor class c, the sum of all hierarchical solution candidates' decision variables must be equal to zero, which avoids the selection of those solution candidates. Vice versa, if the task of node n is mapped to processor class c, the sum of all hierarchical solution candidates' decision variables must be equal to one so that one of those candidates must be chosen. Note that the parallel solution set of child node n contains at least one solution candidate for each processor class which represents the sequential execution on this processor class. This guarantees that the ILP finds a solution for this mapping.

7.1.3.7 Summary

The newly presented ILP-based parallelization approach for heterogeneous embedded MPSoCs combines parallelism extraction with a mapping of tasks to processor classes in a clear mathematical model. By employing this model, the approach is able to determine a good balancing of extracted tasks for heterogeneous processors with different performance characteristics automatically and to combine this with a mapping of tasks to processor classes.

7.1.4 Simple Loop Parallelization Approach

The ILP-based task-level parallelization approach described in this section automatically balances extracted tasks of embedded applications. In addition, it directly maps the extracted tasks to processor classes of an embedded heterogeneous MPSoC. A fast but simple loop-level parallelization approach is combined with the presented task-level one (cf. Algorithm 9), due to the complexity of the problem's

Processor	Execution Time	Factor	Percentage	Iterations
CPU 1	1,153,280	5.13	38.61	30 (+1)
CPU 2	1,281,280	4.62	34.76	27 (+1)
CPU 3	2,332,160	2.54	19.11	15
CPU 4	5,920,320	1.00	7.52	6
Sum	(irrelevant) -	13.29	100.00	78 (80)

Table 7.1: Simple Loop Parallelization Approach Example

solution space and the property that it is difficult to balance loop iterations to concurrently executed tasks by task-level parallelism. The loop-level parallelization approach presented here just divides the different iterations of a loop into concurrently executed tasks. Therefore, a simple approach can be used to extract this kind of parallelism. But due to its simplicity, it is only able to parallelize loops without loop-carried dependencies. A more sophisticated pipeline parallelization approach that overcomes this limitation is later presented in Section 7.2.

An example for a loop with 80 iterations, parallelized for a platform with four different processing units is given in Table 7.1. In a first step, the approach calculates how long each processing unit takes to execute one iteration of the loop (cf. column *Execution Time*). This time includes the time to execute one loop iteration on the specific processor as well as task creation and communication costs for the task created. Just as the ILP-based parallelization approaches, the loop-level one gets this cost information from the AHTG. Based on the determined execution times, a factor is calculated (cf. column *Factor*) which denotes the number of iterations executed on processing unit c while one iteration is executed on the slowest processing unit as defined by Equation 7.11.

$$Factor_c = \max_{p \in Processors} \{ExecTime_p\}/ExecTime_c \tag{7.11}$$

In a third step, the percentage of executed loop iterations is calculated by dividing each factor by the sum of all factors (cf. column *Percentage*) like shown in Equation 7.12.

$$Percentage_c = (Factor_c / \sum_{p \in Processors} Factor_p) * 100 \tag{7.12}$$

Finally, the iterations of the loop are distributed to the different CPUs, depending on the calculated percentages (cf. column *Iterations*). If the sum of assigned iterations is less than the number of loop iterations, the remaining iterations are assigned to the fastest processing units (numbers in brackets). As a result, the number of loop iterations assigned to a processing unit is automatically balanced according to the processing unit's performance characteristics.

By combining this simple but fast loop parallelization approach with the rich but complex ILP-based task-level parallelization approach, efficient parallelism for embedded heterogeneous architectures can be extracted as shown in the following evaluation section.

7.1.5 Experimental Results

To evaluate the efficiency of the new approach, results are obtained from the same benchmarks used to evaluate the previously presented parallelization approaches. One exception is the *jpeg2000* encoder for which the new approach was not able to extract speedups since other parallelization types are required. However, to emphasize the quality of the new approach, it is compared with results generated by the homogeneous task-level parallelization approach presented in Section 5.2. This comparison against the approach which was used as a starting point also highlights the influence of the newly added and adapted constraints.

The ARM11QuadProc multi-processor architecture (cf. Section 3.3.2) equipped with four ARM1176 single-core processors [ARM13a], simulated by the cycle-accurate Vast MPSoC simulator [Syn13b], was used as evaluation platform. Unfortunately, no heterogeneous embedded target platform with different processor types was available for evaluation purposes. Instead, the processors of the ARM11-QuadProc were clocked with varying frequencies to simulate a platform which is comparable to a same-ISA multicore platform, like, e.g., ARM's big.LITTLE platform [Pet13]. The MPARM (cf. Section 3.3.1) as well as the Arm11MPCore platform (cf. Section 3.3.3) could not be adapted since the cores' frequencies could not be configured to be different. However, since the presented approach considers varying execution costs for the application's statements depending on the executing processing unit, the parallelization approach should also perform well for architectures providing cores with different instruction sets.

To emphasize the adaptability of the presented approach to multiple architectures, results are presented for two different platform configurations. Platform configuration (A) configures the four available processors of the ARM11QuadProc platform to work at 100 MHz (1×), 250 MHz (1×) and 500 MHz (2×). This configuration shows that the approach works well for architectures with large performance variances. Platform configuration (B) configures the cores to work at 200 MHz (2×) and 500 MHz (2×) to simulate a performance discrepancy of approximately 2.5×. This is also the average performance difference of ARM's big.LITTLE platform [Pet13] with two Cortex-A7 and two Cortex-A15 cores.

7.1.5.1 Evaluation of Speedup

Both platform configurations were evaluated for two different application scenarios:

I) The main processor of the platform is the slowest one (100 MHz) and the additional cores are added as accelerators.

II) The main processor of the platform is the fastest one (500 MHz) and the other (slower) processors are added to the platform due to, e.g., power or thermal issues.

The measurement baseline in both scenarios is the sequential execution on the main processor. Figure 7.9 depicts results for evaluation scenario (I) with platform

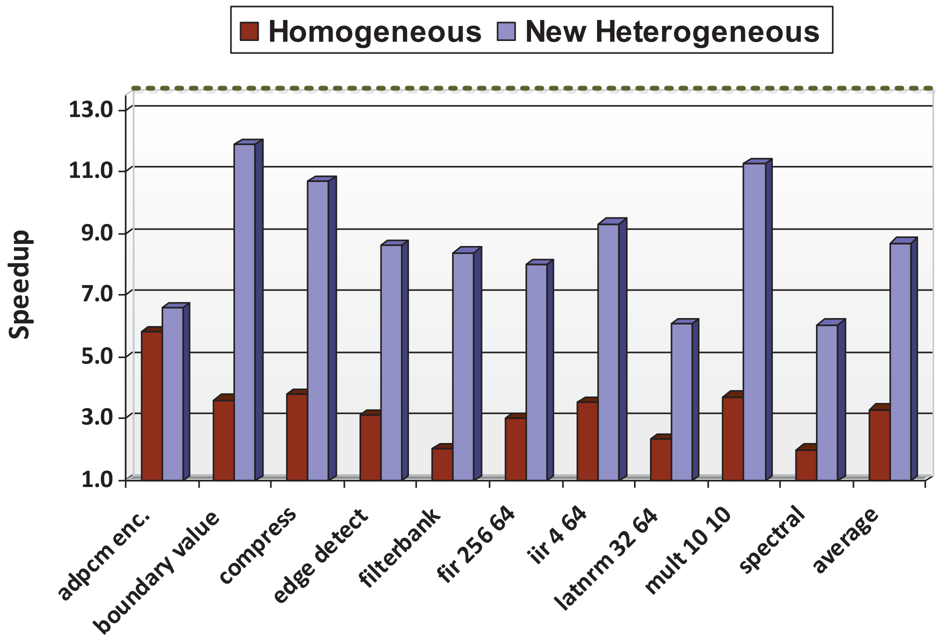


Figure 7.9: Results for Platform Configuration(A): 100/250/500/500MHz in Scenario(I)

configuration (A) and compares the new heterogeneous parallelization approach to the homogeneous one presented in Section 5.2. The dashed line shows the theoretical maximum speedup limit for the evaluated platform configuration which can of course never be fully reached due to, e.g., inserted communication and task creation overhead.

Like shown in Figure 7.9, both approaches improve the performance of all evaluated applications. Since the homogeneous approach is not aware of different processor types, it tries to balance the workload for all available processors uniformly. Speedups between 3× up to 4× were achieved for most applications, which are good performance increases for homogeneous architectures equipped with four processing units. However, the results do not exploit the full potential of the targeted heterogeneous platform. In contrast, results generated by the newly presented heterogeneous approach of this section are much more impressive. It automatically balances the extracted tasks by respecting different performance characteristics of the available processing units. Thus, the two processors with 500 MHz are automatically assigned with heavier workloads than the slower ones. This results in performance increases of up to 10-12× for some of the considered benchmarks (e.g., *boundary value*, *compress* and *mult*) which significantly outperforms the speedup of the homogeneous parallelization approach and is close to the theoretical maximum speedup of 13.5×[3]. On average, the homogeneous parallelization tool increased the applications' performance by 3.3×. In contrast, the new heterogeneous one reached an average speedup of 8.7×.

[3]Theoretical speedup limit: $(1 * 100 + 1 * 250 + 2 * 500\text{MHz})/100\text{MHz} = 13.5\times$

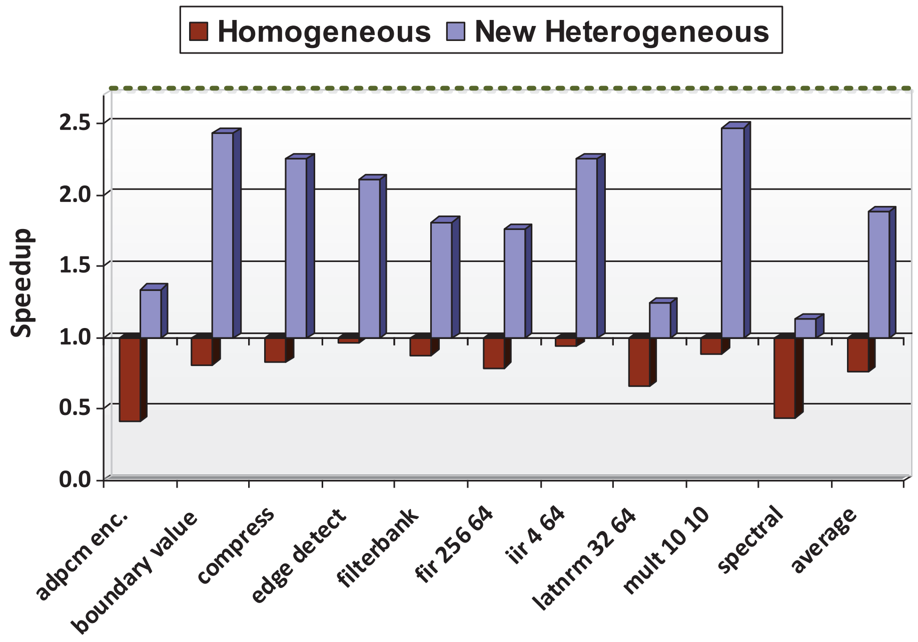


Figure 7.10: Results for Platform Configuration(A): 500/500/250/100MHz in Scenario(II)

Figure 7.10 shows results for evaluation scenario (II) with a fast main processor (500 MHz) and slower additional cores. Here, the speedup produced by the homogeneous approach is less than 1.0, meaning that the parallelized application performs slower than its sequential version. The reason is that the homogeneous approach considers all cores to be identical and uniformly distributes the work to the available processing units. As a consequence, the faster processors have to wait until the slower cores have finished their tasks. This behavior was already discussed in the motivating example at the beginning of this chapter and is now confirmed by these measurements. It also shows that it is even more challenging to extract beneficial parallelism for an architecture with slower additional cores. However, in contrast to the homogeneous approach, the new heterogeneous one was able to speed up the application by generating tasks that perfectly utilize the slower processing units so that all cores finish nearly at the same time. The speedup ranges between 1.2× and nearly 2.5× showing that the approach did not only allocate tasks to the 500 MHz cores and is also close to the theoretical speedup limit of 2.7×[4]. The average speedup for this scenario is 0.8× for the homogeneous and 1.9× for the novel heterogeneous approach, respectively.

To highlight the adaptability of the new heterogeneous parallelization approach, Figures 7.11 and 7.12 present additional results for platform configuration (B) where two ARM cores are configured to run at 200 MHz and the two other cores run at 500 MHz. Both evaluation scenarios with a slow (I) and a fast (II) main processor are visualized, respectively. As can be seen in Figure 7.11, both approaches perform

[4]Theoretical speedup limit: $(1 * 100 + 1 * 250 + 2 * 500\text{MHz})/500\text{MHz} = 2.7\times$

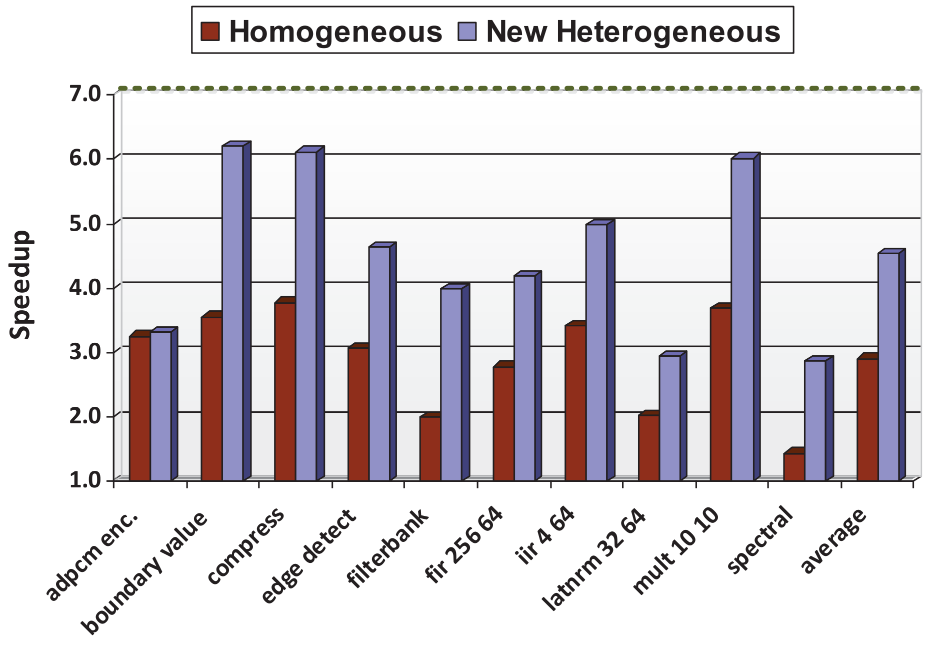


Figure 7.11: Results for Platform Configuration(B): 200/200/500/500MHz in Scenario(I)

well for evaluation scenario (I). The homogeneous parallelization approach reached speedups of around 3× for most evaluated benchmarks. In contrast, the new heterogeneous approach presented in this section reached speedups of more than 6× for the benchmarks *boundary value*, *compress* and *mult*. The achieved speedups are not as high as the ones extracted for platform configuration (A) since the performance difference between the platform's processing units is not as large, here. The theoretical speedup limit of this platform is 7×[5], the one for platform configuration (A) was 13.5×. Hence, the quality of the results is similar for both evaluated platforms even if the achieved speedups are different. The homogeneous approach reached an average speedup of 2.9× for this platform while the heterogeneous one was able to increase the applications' performance by 4.5× on average.

The same observations were made for evaluation scenario (II). The results for this scenario are presented in Figure 7.12. The homogeneous parallelization approach reached speedups of up to 1.7× while the heterogeneous one increased the applications' performance up to 2.6×. Also here, the results are close to the platform's theoretical speedup limit of 2.8×[6] which emphasizes that the new approach balances the workload well between all available processing units by respecting different performance characteristics. In contrast to platform configuration (I), the homogeneous parallelization approach reached speedups larger than 1.0 since the performance discrepancy between the available cores is not as large as for the first platform configuration. However, the new heterogeneous approach reached higher

[5]Theoretical speedup limit: $(2 * 200 + 2 * 500\text{MHz})/200\text{MHz} = 7\times$

[6]Theoretical speedup limit: $(2 * 200 + 2 * 500\text{MHz})/500\text{MHz} = 2.8\times$

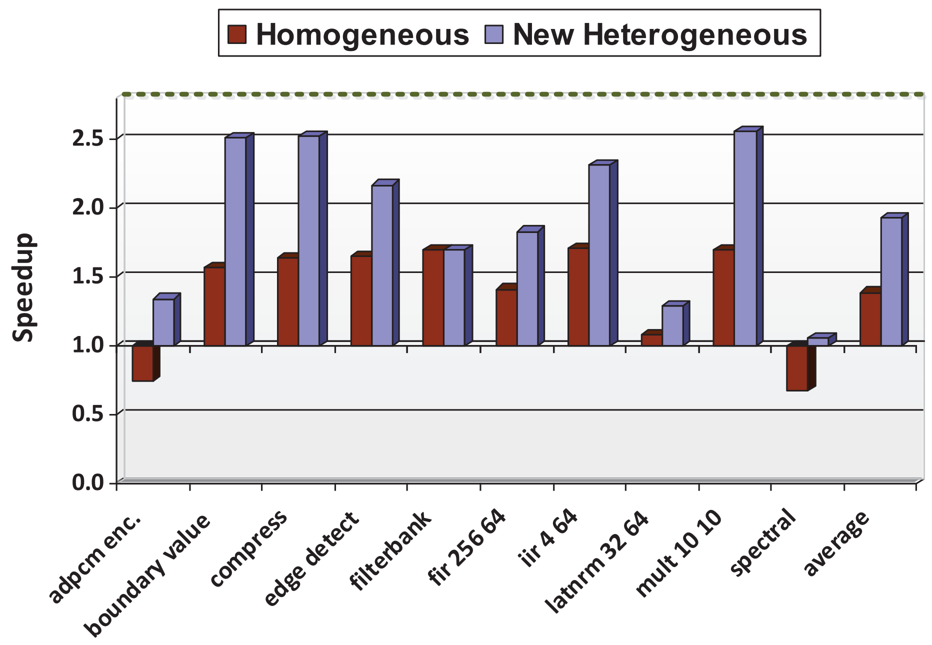


Figure 7.12: Results for Platform Configuration(B): 500/500/200/200MHz in Scenario(II)

speedups for all evaluated benchmarks. The performance of some benchmarks (e.g., *latnrm* or *spectral*) can still be improved. Those benchmarks have higher communication loads, and the current approach extracts task-level parallelism only but the applications profit more from other parallelism types, like, e.g., pipeline parallelism. Therefore, extraction techniques for this kind of parallelism for heterogeneous architectures will later be presented in Section 7.2.

7.1.5.2 Optimization Time & ILP Statistics

Table 7.2 summarizes collected data for all evaluated benchmarks for platform configuration (A) in scenario (I) and compares the newly presented heterogeneous parallelization approach of this section to the homogeneous one, presented in Section 5.2. The table contains information about the time in minutes which was necessary to parallelize the applications with both approaches (*Time*), the number of generated ILPs (#ILPs), the number of created variables for all generated ILPs (#*Var*), and the overall number of created constraints (#*Const*). For the new heterogeneous approach, absolute numbers are given on the left-hand side of Table 7.2 while the right-hand side depicts factors describing the increased complexity of the heterogeneous approach compared to the homogeneous one.

The ILP formulations are more complex in the heterogeneous case since a new dimension was added describing the task-to-processor type mapping. The new heterogeneous parallelization approach also created and solved more ILPs than the homogeneous one. This is necessary since parallel solution candidates for different processor classes have to be extracted on each hierarchical level. Otherwise,

	New Heterogeneous Approach				Difference to Homogeneous Appr.			
Benchmark	Time[7]	#ILPs	#Var	#Const	Time[7]	#ILPs	#Var	#Const
adpcm enc.	00:15	78	50,631	70,488	5.0×	3.4×	7.3×	5.6×
bound. value	00:08	21	18,303	26,832	1.6×	3.0×	5.4×	4.4×
compress	12:12	438	242,382	347,448	34.9×	7.4×	14.8×	11.2×
edge detect	00:42	141	73,647	108,594	5.3×	2.9×	6.0×	4.7×
filterbank	06:27	20	27,918	38,962	55.3×	3.3×	7.5×	5.8×
fir 256 64	00:02	24	7,152	9,192	2.0×	2.4×	5.7×	4.8×
iir 4 64	00:08	24	35,817	52,950	4.0×	2.4×	4.9×	4.1×
latnrm 32 64	00:04	36	13,200	17,568	2.0×	2.6×	5.6×	4.6×
mult 10 10	00:06	45	16,005	23,157	3.0×	4.1×	7.0×	5.4×
spectral	11:40	102	74,595	111,212	29.2×	3.1×	5.6×	4.3×
average	03:10	93	55,965	80,640	14.2×	3.5×	7.0×	5.5×

Table 7.2: Comparison of Homogeneous vs. Heterogeneous ILP-based Task-Level Parallelization Algorithms based on Platform Configuration (A) in Scenario (II)

the parallelization process on the parent hierarchical level would be limited which would drastically reduce the solution quality. The number of generated ILPs increases by factors between 2.4× and 7.4× while moving from the homogeneous to the heterogeneous case. The average increase of generated ILPs over all evaluated benchmarks is 3.5×. The increase of newly created variables in the heterogeneous case ranges from 4.9× up to 14.8× (7.0× on average) while the number of created constraints are increased by 4.1× up to 11.2× (5.5× on average). Many new variables and constraints had to be added to parallelize applications for heterogeneous architectures. However, if the number of constraints is increased to 5.5× (average case) while the number of generated ILPs is increased to 3.5× (average case), the average increase of constraints per ILP is manageably low (<1.5×). This has also an impact on the time the parallelization approach needs to parallelize an application. The original homogeneous approach parallelized an application for four cores in 9 seconds on average (cf. Table 5.1), while the heterogeneous one needs 3:10 minutes. Note that, due to the hierarchical approach, runtimes are still remaining in an acceptable state. Nevertheless, the speedups outweigh the higher execution times at compile time since the approach has to be executed only once in the compilation process. The new heterogeneous approach can of course also be applied to extract parallelism for homogeneous architectures and is therefore "downwards compatible". But due to higher execution times it makes more sense to use the approach which is optimized for homogeneous architectures.

To summarize, the following results were achieved:

1. The presented heterogeneous parallelization approach utilizes heterogeneous platforms in an excellent way. Speedups of up to 10–12× could be achieved for evaluation platform (A) in scenario (I) with a theoretical speedup limit of 13.5×.

[7]Time format MM:SS, measured on an AMD Opteron core running at 2.4 GHz

2. The combination of mapping decisions with knowledge of heterogeneous performance characteristics in the parallelization approach is highly beneficial since tasks can be directly optimized for specific processing units.

3. The new heterogeneous approach is able to increase the applications' performance even for platforms with cores which are much slower than the main processor.

4. In contrast to the homogeneous approach, the new heterogeneous one never generated speedups less than 1.0 and significantly outperformed the homogeneous approach on heterogeneous architectures for all benchmarks.

5. Even though ILP is NP-complete in the general case, it could be shown that, due to the hierarchical approach, execution times still remain acceptable.

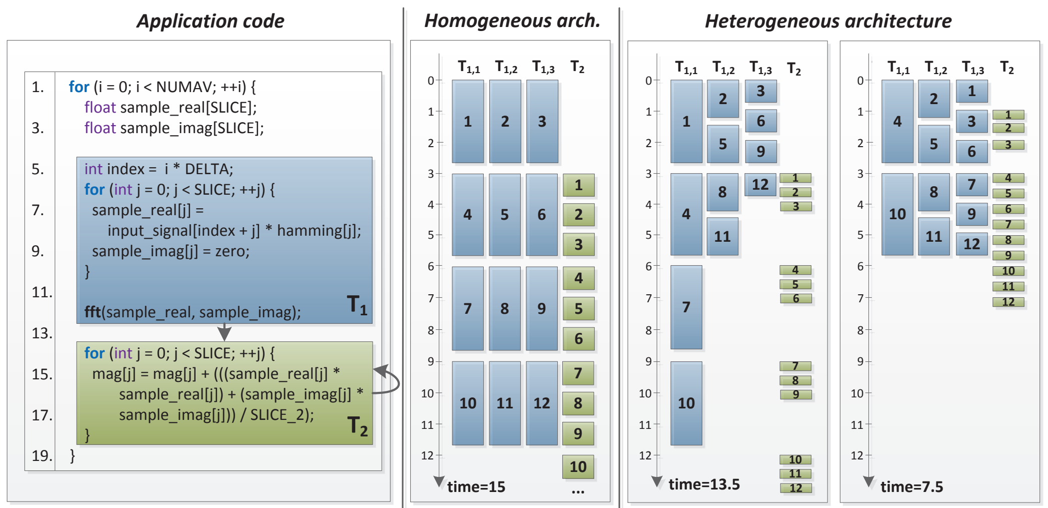
7.2 ILP-based Pipeline Parallelization Approach

The task-level parallelization approach presented in the previous section has revealed large performance increases exploitable by parallelization approaches which are aware of heterogeneity in embedded MPSoCs. However, the presented results have also shown that some of the benchmarks (e.g., *spectral*, cf. Figure 7.12) require additional parallelization techniques to exploit their parallel potential entirely. Therefore, this section presents a new pipeline parallelization approach optimized for heterogeneous architectures, based on the homogeneous one presented in Section 5.3. The previously presented ILP system for homogeneous MPSoCs and the kind of extracted parallelism are used as a basis even though a lot of changes had to be made. The approach presented in this Section is also integrated into the global parallelization approach employing the Augmented Hierarchical Task Graph (cf. Section 4.1) with its divide-and-conquer-based parallelization technique (cf. Section 4.2). Thus, the pipeline parallelization approach presented in this section can easily be executed separately or in a combined fashion with other parallelization techniques. It is recommended to read Section 5.3 first, since the approach presented here is based on the homogeneous one.

The rest of this section is structured as follows: First, Section 7.2.1 demonstrates the impact of the task balancing problem arising on heterogeneous architectures for pipeline parallelism. The section also proposes possibilities to solve this problem by discussing a concise example. Afterwards, the integration of the presented parallelization model into the global parallelization approach is described in Section 7.2.2. The employed ILP-based parallelization technique is described in Section 7.2.3 before the new approach is evaluated in Section 7.2.4.

7.2.1 Motivating Example

The *spectral* benchmark from the UTDSP benchmark suite [Lee13] has already been used as a motivating example for the homogeneous ILP-based pipeline parallelization



```
for (i = 0; i < NUMAV; ++i) {
   float sample_real[SLICE];
   float sample_imag[SLICE];

   int index =  i * DELTA;
   for (int j = 0; j < SLICE; ++j) {
     sample_real[j] =
         input_signal[index + j] * hamming[j];
     sample_imag[j] = zero;
   }

   fft(sample_real, sample_imag);

   for (int j = 0; j < SLICE; ++j) {
     mag[j] = mag[j] + (((sample_real[j] *
          sample_real[j]) + (sample_imag[j] *
          sample_imag[j])) / SLICE_2);
   }
}
```

(a) application code (b) 2 stages, 3 sub-tasks for T1 (c) solution (b), executed on heterogeneous system (d) same solution, optimized for heterogeneous system

Figure 7.13: Motivating Example for Heterogeneous Pipeline Parallelization

technique in Section 5.3.1. It is used here, once again, to describe the demands imposed by heterogeneous MPSoCs. The application code is shown in Figure 7.13 (a) divided into two pipeline stages T_1 and T_2 by horizontal splits (cf. Section 5.3.1). Task T_1 is vertically split twice into the three sub-tasks $T_{1,1}$, $T_{1,2}$, and $T_{1,3}$ executing the different iterations of the extracted pipeline stages in parallel. The timing of this solution mapped to a homogeneous architecture is depicted in Figure 7.13 (b). This result was already obtained by the homogeneous pipeline parallelization approach as the final solution for the *spectral* benchmark (cf. Section 5.3.1). In this example, four tasks are performing their work in a well-balanced execution behavior.

However, Figure 7.13 (c) shows what might happen if solution (b) would be mapped to a heterogeneous architecture containing processors with different performance characteristics. This example assumes that the processing unit executing task $T_{1,1}$ has the same performance as the processing units of the homogeneous architecture assumed in Figure 7.13 (b). The other three processing units are added as accelerators. Thus, task $T_{1,2}$ is executed twice as fast as $T_{1,1}$ while the processing unit used for $T_{1,3}$ executes the iterations three times faster than the processing unit for $T_{1,1}$. Moreover, T_2 is executed twice as fast as $T_{1,1}$.

Most existing parallelization tools do not possess any information about the targeted architecture and are not aware of those performance variances. Rather, they would have to assume a homogeneous architecture. This leads to a highly unbalanced timing behavior like the one shown in Figure 7.13 (c). Task $T_{1,3}$ has executed all its iterations in 4 time units while $T_{1,1}$ needs 12 time units since it is mapped onto a slower processing unit. Hence, the processing units executing $T_{1,2}$ and $T_{1,3}$ are idle for a long time which reduces the performance of the parallelized solution. Task T_2 also often has to wait for data generated by the three sub-tasks of T_1. Solution (b), executed on the homogeneous architecture, needs 15 time units while the execution of solution (c) took 13.5 time units. This shows that the potential of the accelerated cores can only be exploited sub-optimally by parallelization approaches that are not optimized for heterogeneous architectures.

To circumvent this problem, the approach presented in this section handles different execution times for all statements of the application (and therefore also for created tasks) depending on the mapped processing unit while extracting pipeline parallelism. In addition, the approach performs a pre-mapping of tasks to processor classes, representing identical processing types of the heterogeneous target architecture as done by the heterogeneous task-level parallelization approach before. Another difference to the homogeneous pipeline parallelization approach presented in Section 5.3 is the way how the approach maps iterations to sub-tasks. The previously presented approach maps all iterations with a different offset to the same sub-task by vertical splits. In contrast, the approach presented in this section freely maps iterations to sub-tasks. In the example of Figure 7.13 (d), $T_{1,3}$ executes iterations $\{1, 3, 6, 7, 9, 12\}$ while task $T_{1,2}$ executes $\{2, 5, 8, 11\}$. This freedom of decision increases the complexity of the solution space but enables the extraction of well-balanced tasks for heterogeneous embedded MPSoCs. All sub-tasks of T_1 finish at the same time and provide task T_2 with input data in an optimized way reducing the execution time from 13.5 to 7.5 time units. Compared to the previously presented homogeneous pipeline parallelization approach, the new one presented in this section is able to exploit advantages of heterogeneous architectures for the extraction of pipeline parallelism.

7.2.2 Integration into the Global Parallelization Approach

The integration of the new ILP-based pipeline parallelization approach optimized for heterogeneous MPSoCs into the global parallelization framework (cf. Section 4.2) is shown in Algorithm 10. The function EXTRACTHETPIPELINE is executed by the global parallelization algorithm (cf. Algorithm 4 on page 56) as soon as all child nodes of the AHTG deeper in the hierarchy are processed. The first lines are equal to the integration of the homogeneous pipeline parallelization approach depicted in Algorithm 6. The algorithm first determines in lines 4-6 whether the currently processed node n does not represent a loop statement since only those statements are processed by this approach.

Since the complexity of the heterogeneous ILP-based pipeline parallelization approach is very high, the algorithm first determines if the iterations of the loop to be parallelized are free of loop-carried dependencies. If this is true, the less complex loop parallelization approach described in Section 7.1.4 can also be applied here in lines 8-11. The solutions extracted by the simple loop parallelization approach are a subset of solutions extracted by the pipeline parallelization approach presented in this section. Such a solution contains only one pipeline stage divided by vertical splits into several concurrently executed tasks. As a consequence, the solution of the simple loop parallelization methodology called in lines 8-11 could also be extracted by the ILP-based approach but would result in a significantly longer execution time.

Algorithm 10 Integration of Het. ILP-based Pipeline Parallelization Approach

```
1: // Called bottom-up hierarchically by Algorithm 4 in line 18 on page 56
2: function ExtractHetPipeline(Node $n$, Platform $pf$, int $maxTasks$)
3:     // This function is only applicable to loops.
4:     if not $n.getStmt().isLoopStmt()$ then
5:         return $\emptyset$
6:     end if
7:     // If n represents an independent loop, use the simple loop parallelizer.
8:     if IsLoopIndependent($n$) then
9:         $solutions \leftarrow$ SimpleParallelizer($n, pf, maxTasks$)
10:        return $solutions$
11:    end if
12:    // Otherwise, create an augmented PDG for the loop(-nest).
13:    $loopPDG \leftarrow$ ConstructPDG($n.getStmt()$)
14:    // Apply the ILP-based approach.
15:    for all $seqPC \in pf.getProcClasses()$ do
16:        $i \leftarrow maxTasks$
17:        while $i >= 2$ do
18:            $result \leftarrow$ HetILPPipelineParallelizer($loopPDG, seqPC, pf, i$)
19:            $solutions \leftarrow solutions \cup \{result\}$
20:            $i \leftarrow$ NumberOfTasks($result$) $- 1$
21:        end while
22:    end for
23:    return $solutions$
24: end function
```

In general, this strategy does not harm the solution quality as long as the number of available processing units is not too high so that an insufficient number of tasks may be extracted by vertical splits.

If node n represents a loop statement containing loop-carried dependencies, the sophisticated ILP-based pipeline parallelization approach presented in Section 7.2.3 is repeatedly called in line 18 for different processor classes and varying upper task boundaries. Also here, an augmented Program Dependence Graph (extracted in line 13) is used to extract pipeline parallelism. Several solutions with different processing unit allocations are extracted in lines 15-22 to provide flexibility to the parallelization process upwards in the hierarchy. In this way, new and previously extracted parallelism with different granularities can later be combined. All extracted results are collected and finally returned as solution candidates for node n in line 23.

7.2.3 ILP-based Parallelization Approach

The pipeline parallelization approach presented in this section uses the homogeneous one presented earlier in Section 5.3 as a starting point. However, as shown in the motivating example (cf. Section 7.2.1), it is indispensable to distinguish between different execution times of the same code blocks depending on the executing processing unit for heterogeneous architectures. Therefore, many parts of homogeneous ILP have to be changed and new decision variables and constraints have to be added. In the following, these changes will be highlighted to ease a comparison between both approaches. A complete presentation of all ILP formulations is available in the corresponding publication of this approach in [CEN+13b].

The new heterogeneous approach covers five main goals which are translated into constraints and decision variables in the remainder of this section:

I) Extraction of different pipeline stages by mapping statements of the loop's body into disjunctive pipeline stages (cf. horizontal splits in Figure 7.13 (a)).

II) Division of pipeline stages into sub-tasks which execute different iterations of the stages in parallel (cf. vertical splits in Figure 7.13 (b)).

III) Handling of dependencies which may change if statements are moved from one pipeline stage to another one or if iterations of pipeline stages are mapped to different sub-tasks.

IV) Extraction of a mapping of tasks to processor classes of the targeted embedded heterogeneous architecture (cf. Figure 7.13 (d)) to balance the execution load.

V) Minimization of execution costs by considering task creation, communication, and execution costs depending on the processor class a task is mapped to.

In the following paragraphs, decision variables are written in lower case letters, sets start with a capital letter, and constants contain exclusively capital letters. Indices n and o are used for nodes of the augmented PDG, i and j are used for iterations of the loop to be parallelized, t and u represent indices for pipeline stages, while c represents a processor class and s is used for concurrently executed sub-tasks of a pipeline stage. A graphical representation of most equations is also given in Figures 7.14 - 7.21. The sub-figures have the same name as the corresponding subsections which describe the equations.

7.2.3.1 Pipeline Stage Mapping

Goal (I) of the heterogeneous ILP-based pipeline parallelization approach is a mapping of PDG nodes to pipeline stages. To perform this, decision variable x_n^t is defined in Equation 7.13. Since this constraint is important but unchanged compared to the homogeneous approach, it is just briefly summarized here.

$$x_n^t = \begin{cases} 1, & \text{if node } n \text{ is mapped to pipeline stage } t \\ 0, & \text{otherwise} \end{cases} \tag{7.13}$$

The constraint defined in Equation 7.14 ensures that every child node (representing statements of the loop to be parallelized) is mapped to exactly one pipeline stage.

$$\forall n \in Nodes : \sum_{t \in Stages} x_n^t = 1 \tag{7.14}$$

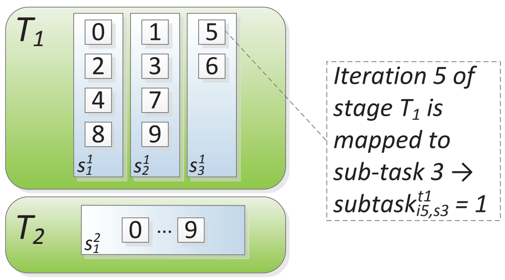

T_1
0 2 4 8 s_1^1
1 3 7 9 s_2^1
5 6 s_3^1
Iteration 5 of stage T_1 is mapped to sub-task 3 → $subtask^{t1}_{i5,s3} = 1$
T_2
s_1^2 0 ··· 9

Figure 7.14: Iterations of Pipeline Stages to Sub-Task Mapping

7.2.3.2 Iterations of Pipeline Stages to Sub-Task Mapping

Goal (II) of the ILP-based pipeline parallelization approach is a mapping of loop iterations of the created pipeline stages to concurrently executed sub-tasks. This constraint replaces the Vertical Split Constraint presented in Section 5.3.5.2. In the homogeneous case, it was sufficient to divide the iteration space of a pipeline stage into chunks of equal size to balance the extracted tasks. This was achieved by vertical splits denoted by decision variable $split^t_s$. As shown in the motivating example of this section (cf. Section 7.2.1), an approach optimized for heterogeneous architectures should be able to freely map iterations of the pipeline stages to additional sub-tasks. Thereby, heavier workloads can be mapped to faster processing units while tasks with fewer iterations can be mapped to slower ones. This is accomplished by decision variable $subtask^t_{i,s}$ which is defined in Equation 7.15. It evaluates to one if iteration i of pipeline stage t is mapped to sub-task s like also visualized in Figure 7.14.

$$subtask^t_{i,s} = \begin{cases} 1, & \text{if iteration } i \text{ of pipeline stage } t \text{ is mapped to sub-task } s \\ 0, & \text{otherwise} \end{cases} \tag{7.15}$$

To be compliant with the original program semantics, the ILP has to take care that each loop iteration (NI = number of loop iterations) is executed exactly once for each pipeline stage, which is ensured by Equation 7.16.

$$\forall t \in Stages : \forall i \in \{0, .., NI{-}1\} : \sum_{s \in SubTasks^t} subtask^t_{i,s} = 1 \tag{7.16}$$

7.2.3.3 Definition of Predecessor Relationships

The main objective of the new heterogeneous pipeline parallelization approach is a reduction of the loop's execution time by moving statements of the loop's body into disjunctive pipeline stages like performed by decision variable x^t_n. The loop iterations of each stage can also be executed concurrently in different sub-tasks like defined by decision variable $subtask^t_{i,s}$. In order to minimize the execution time, the critical (most expensive) path from the *entry* to the *exit* node of the loop's PDG has to be extracted. Therefore, decision variable $pred^{t,u}_{i,j}$ was defined by the homogeneous pipeline parallelization approach in Equation 5.22 which evaluates to one if pipeline stage t in iteration i is a predecessor of pipeline stage u in iteration

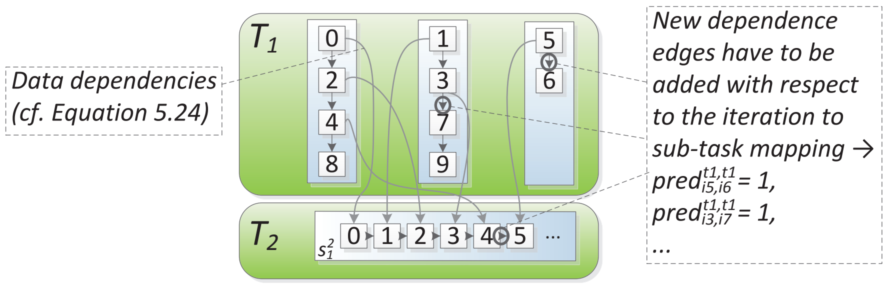

T_1
T_2
Data dependencies (cf. Equation 5.24)
New dependence edges have to be added with respect to the iteration to sub-task mapping → $pred^{t1,t1}_{i5,i6}$ = 1, $pred^{t1,t1}_{i3,i7}$ = 1, ...
s^2_1

Figure 7.15: Predecessor Relationships (Sub-Task Dependencies)

j (cf. Goal (III)). Two situations can lead to such a predecessor relationship. The first one relies on data and control flow dependencies of the child nodes mapped to the different pipeline stages while the second one relies on the mapping of loop iterations to sub-tasks. Data and control flow dependencies were covered by the constraint defined in Equation 5.24 and can be re-used without modifications for the heterogeneous ILP, as well. However, the Stage Split Dependencies defined in Equation 5.23 cannot be integrated into the heterogeneous ILP system since they rely on the deprecated $split^t_s$ variable which is not used in the heterogeneous system. Therefore, new sub-task dependencies are defined in Equation 7.17 and visualized in Figure 7.15, reflecting dependencies of the new approach which is able to freely map iterations to sub-tasks. Thus, if iteration i and iteration j (with $i < j$) of pipeline stage t are both mapped to sub-task s, iteration j depends on iteration i since both iterations are executed sequentially by the same sub-task.

$$\forall t \in Stages : \forall i \in \{0, .., NI{-}1\} : \forall j \in \{i+1, .., NI{-}1\} :$$
$$pred^{t,t}_{i,j} \geq subtask^t_{i,s} \wedge subtask^t_{j,s} \tag{7.17}$$

If, e.g., iterations 1 and 3 of pipeline stage t are mapped to the same sub-task, both iterations have to be executed sequentially. As a consequence, iteration 1 of pipeline stage t is a predecessor of iteration 3 of pipeline stage t. Those dependencies are created for all iteration combinations of the different pipeline stages

7.2.3.4 Extraction of Execution Costs of Pipeline Stages

To be able to reduce the overall execution time, path costs have to be determined. Therefore, execution costs for one iteration of the different pipeline stages have to be estimated first. In the homogeneous case, it was sufficient to sum up the execution costs of the nodes which are added to the considered pipeline stage. Since it is essential to consider different performance characteristics of the different processing units if an application should be parallelized for heterogeneous architectures, execution costs depend on the processor class executing the given pipeline stage. This is done in Equation 7.18 which creates one cost variable $cost^t_c$ for all pipeline stages t executed on processor class c. In the homogeneous case, the variable $cost_t$ was

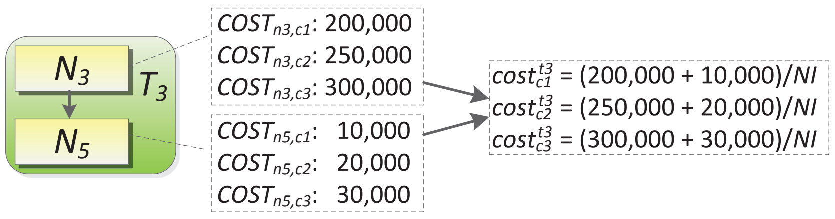


Figure 7.16: Extraction of Execution Costs of Pipeline Stages

independent of the executing processor class so that it had to be adjusted, here.

$$\forall c \in ProcClasses : \forall t \in Stages : cost^t_c \geq \sum_{n \in Nodes} x^t_n * (COST_{n,c}/NI) \quad (7.18)$$

The variable $cost^t_c$ contains the execution costs $COST_{n,c}$ of all child nodes n mapped to the corresponding pipeline stage t ($x^t_n = 1$) for the execution of one iteration on processor class c (cf. Figure 7.16). The overall execution costs of each child node are distributed in equal parts over the iterations NI of the loop. NI can be determined by static loop analyzers, like, e.g., [Cor08], [LCF+09], or the analyzer integrated in the employed ICD-C high-level IR [Inf13]. This saves several decision variables since the ILP does not have to distinguish between different execution costs of pipeline stages in different iterations. The execution costs of child node n executed on processor class c are annotated to the nodes of the PDG and are automatically extracted by the framework presented in this thesis.

7.2.3.5 Mapping of Sub-Tasks to Processor Classes

Variable $cost^t_c$ contains the execution costs of one iteration of pipeline stage t if it is executed on processor class c. But, up to now, pipeline stages and their sub-tasks are not mapped to any processor classes. This is not necessary for homogeneous architectures, but has a large impact on the execution time for heterogeneous ones. Therefore, the approach presented in this section combines the extraction of parallelism with a mapping of sub-tasks to processor classes to be able to create well-balanced solutions like demanded by Goal (IV). This mapping is expressed by decision variable $map^t_{s,c}$ defined in Equation 7.19 which evaluates to 1 if sub-task s of pipeline stage t is mapped to processor class c like visualized in Figure 7.17.

$$map^t_{s,c} = \begin{cases} 1, & \text{if sub-task } s \text{ of pipeline stage } t \\ & \text{is mapped to processor class } c \\ 0, & \text{otherwise} \end{cases} \quad (7.19)$$

Each sub-task s of pipeline stage t has to be mapped to exactly one processor class c so that it is executed exactly once which is ensured by Equation 7.20.

$$\forall t \in Stages : \forall s \in SubTasks^t : \sum_{c \in ProcClasses} map^t_{s,c} = 1 \quad (7.20)$$

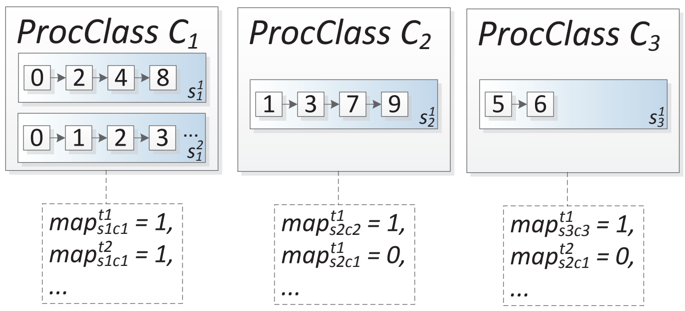

ProcClass C1
0 2 4 8
s11
0 1 2 3 ...
s12
ProcClass C2
1 3 7 9
s21
ProcClass C3
5 6
s31
map^t1_s1c1 = 1, map^t2_s1c1 = 1, ...
map^t1_s2c2 = 1, map^t1_s2c1 = 0, ...
map^t1_s3c3 = 1, map^t2_s2c1 = 0, ...

Figure 7.17: Mapping of Sub-Tasks to Processor Classes

7.2.3.6 Used Pipeline Stages & Used Sub-Tasks

The presented approach has the capability to extract as many pipeline stages as processing units are available. Nevertheless, task creation and communication costs as well as different performance characteristics of the available processing units may result in the ILP extracting fewer tasks if such a solution leads to a higher reduction of the overall execution time. Hence, some of the pipeline stages may not be used. This can be evaluated by decision variable $stageused^t$ which was defined by the homogeneous ILP in Equation 5.27 with its corresponding constraint in Equation 5.28. Both equations can be re-used without changes for the new heterogeneous ILP-based approach.

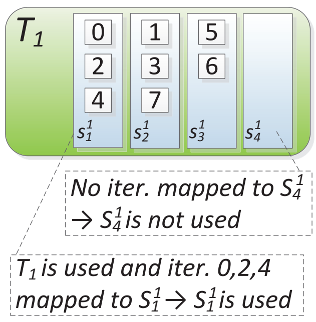
In addition, the approach enables the extraction of as many sub-tasks s for a pipeline stage t as processing units are available. The extraction of sub-tasks directly influences the overall execution costs since task creation costs are added for each created sub-task. To determine the amount of created sub-tasks, a decision variable $subtaskused_s^t$ is created for each sub-task s of pipeline stage t like shown in Equation 7.21.

$$subtaskused_s^t = \begin{cases} 1, & \text{if sub-task } s \text{ of pipeline stage } t \text{ is used} \\ 0, & \text{otherwise} \end{cases} \tag{7.21}$$

Sub-task s of pipeline stage t is used if at least one iteration i is mapped to it and pipeline stage t itself is used like ensured by Equation 7.22 and visualized in Figure 7.18.

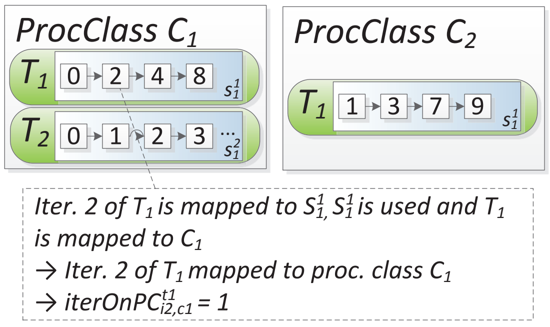
$$\begin{gathered} \forall t \in Stages : \forall s \in SubTasks^t : \forall i \in \{0, .., NI{-}1\} : \\ subtaskused_s^t \geq subtask_{i,s}^t + stageused^t - 1 \end{gathered} \tag{7.22}$$

The ILP formulation can further be optimized here by excluding solutions with the same objective value, which only differ in the allocated sub-tasks (cf. [LMM+97] and Section 9.2). This could be achieved by adding constraints like $subtaskused_s^t \geq subtaskused_s^{t+1}$ and may be integrated into the system in the future.



T_1
0
2
4
1
3
7
5
6
S_1^1
S_2^1
S_3^1
S_4^1
No iter. mapped to S_4^1 → S_4^1 is not used
T_1 is used and iter. 0,2,4 mapped to S_1^1 → S_1^1 is used

Figure 7.18: Used Sub-Tasks Constraint



ProcClass C_1
T_1 0 2 4 8 S_1^1
T_2 0 1 2 3 ... S_1^2
ProcClass C_2
T_1 1 3 7 9 S_1^1
Iter. 2 of T_1 is mapped to S_1^1, S_1^1 is used and T_1 is mapped to C_1
→ Iter. 2 of T_1 mapped to proc. class C_1
→ $iterOnPC_{i2,c1}^{t1} = 1$

Figure 7.19: Mapping of Sub-Task Iterations to Processor Classes

7.2.3.7 Mapping of Sub-Task Iterations to Processor Classes

Up to now, only a relation between sub-tasks and their mapped processor classes exists, like defined in Equation 7.19. For the calculation of the overall execution costs, the relation between iteration i of pipeline stage t and its mapped processor class is also necessary. Therefore, Equation 7.23 defines decision variable $iterOnPC_{i,c}^{t}$.

$$iterOnPC_{i,c}^{t} = \begin{cases} 1, & \text{if iteration } i \text{ of pipeline stage } t \\ & \text{is mapped to processor class } c \\ 0, & \text{otherwise} \end{cases} \tag{7.23}$$

Iteration i of pipeline stage t is mapped to processor class c if it is part of sub-task s and s is mapped to processor class c if sub-task s is really used. Equation 7.24 evaluates to one if this is true like visualized in Figure 7.19.

$$\begin{gathered} \forall t \in Stages : \forall c \in ProcClasses : \forall s \in SubTasks^{t} : \forall i \in \{0,..,NI{-}1\} : \\ iterOnPC_{i,c}^{t} \geq subtask_{i,s}^{t} + map_{s,c}^{t} + subtaskused_{s}^{t} - 2 \end{gathered} \tag{7.24}$$

7.2.3.8 Path Cost Constraint

Based on the knowledge of the execution costs of each pipeline stage and the mapping of loop iterations to processor classes, it is now possible to describe the accumulated path costs (cf. Figure 7.20). Equation 7.25 defines $accumcost_{j}^{t}$ and ensures that it contains the execution costs of all executed predecessors as well as the execution costs of the pipeline stage's iteration itself.

$$\begin{gathered} \forall t, u \in Stages : \forall c \in ProcClasses : \forall i \in \{0,..,NI{-}1\} : \\ \forall j \in \{i+1,..,NI{-}1\} : pred_{i,j}^{u,t} = 1 \wedge iterOnPC_{j,c}^{t} = 1 : \\ accumcost_{j}^{t} \geq cost_{c}^{t} + accumcost_{i}^{u} + commcost_{u} \end{gathered} \tag{7.25}$$

Equation 7.25 ensures that the path costs $accumcost_{j}^{t}$ for pipeline stage t in iteration j are at least as large as the costs $cost_{c}^{t}$ for the execution of one iteration of

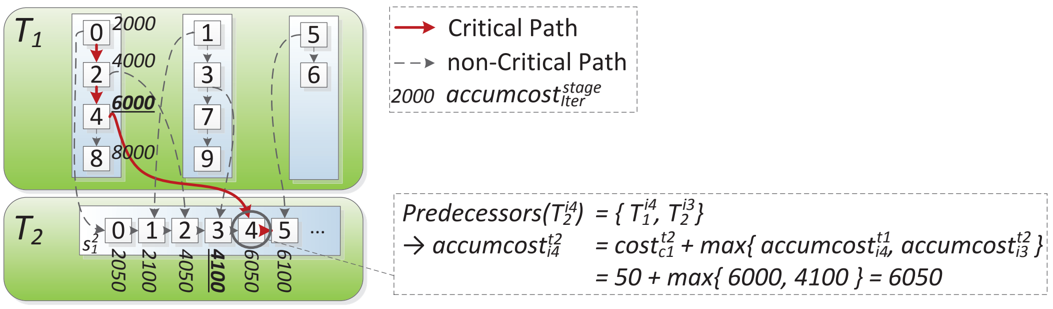

T1
0
2
4
8
2000
4000
6000
8000
1
3
7
9
5
6
Critical Path
non-Critical Path
2000 accumcost stage Iter
T2
s 2 1
0 1 2 3 4 5 ...
2050
2100
4050
4100
6050
6100
Predecessors(T i4 2) = { T i4 1, T i3 2 }
→ accumcost t2 i4 = cost t2 c1 + max{ accumcost t1 i4, accumcost t2 i3 }
= 50 + max{ 6000, 4100 } = 6050

Figure 7.20: Path Cost Constraint

pipeline stage t itself executed on processor class c. It is increased by the path costs of its most expensive predecessor $accumcost_i^u$, including all communication costs $commcost_u$ of pipeline stage u. The precondition $iterOnPC_{j,c}^t = 1$ takes care that the costs of the correct processing class c of task t is used. The accumulated costs are included in the objective function so that they are automatically minimized by the ILP solver.

7.2.3.9 Limit the Number of Allocated Tasks per Processor Class

A platform is equipped with a limited number of processing units. By taking advantage of platform information in the parallelization step, it is possible to avoid additional scheduling overhead at runtime. Therefore, each processing unit should execute only one sub-task of a pipeline stage in the proposed model at a time. Thus, the constant number of available processing units $NUMPROCS_c$ of a processor class c must be at least as large as the number of mapped sub-tasks $map_{s,c}^t$ if they are used ($subtaskused_s^t$). This is ensured by Equation 7.26 and visualized in Figure 7.21.

$$\forall c \in ProcClasses :$$
$$\sum_{t \in Stages} \sum_{s \in SubTasks^t} map_{s,c}^t \wedge subtaskused_s^t \leq NUMPROCS_c \tag{7.26}$$

7.2.3.10 Objective Function

With all decision variables and constraints defined, it is now possible to describe the objective function. As mentioned before, the most expensive execution path from the *entry* to the *exit* node of the loop's PDG should be minimized like defined by Goal (V). Therefore, additional constraints are added which statically set the *entry* node to be a predecessor of all pipeline stages. The *exit* node will be a successor of all pipeline stages, respectively. With the help of all defined constraints, it is easy to create the objective function, like shown in Equation 7.27. The objective function is equal to the one of the homogeneous parallelization approach but considers the new version of the *accumcost* variables.

$$exectime = numtasks * TASKOVERHEAD + accumcost_{exit} \rightarrow \min \tag{7.27}$$

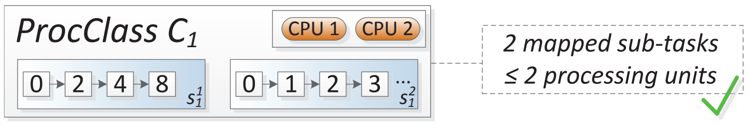


Figure 7.21: Limit the Number of Allocated Tasks per Processor Class

The variable *numtasks* contains the number of extracted sub-tasks used. Since the creation of such tasks increases the execution time, a constant task creation overhead, multiplied by the number of created sub-tasks, is added to the objective value as defined in Equation 7.28.

$$numtasks = \sum_{t \in Stages} \sum_{s \in SubTasks^t} subtaskused_s^t \tag{7.28}$$

The task creation overhead can be defined in the platform description together with a communication cost factor. By defining these platform-dependent parameters, it is easy to adapt the cost model of the ILP to different architectures. The value of the objective function is equivalent to the execution time of the parallelized loop on the targeted heterogeneous architecture. It is hence returned together with the node-to-pipeline-stage mapping, the mapping of the stages' loop iterations to sub-tasks and the mapping of sub-tasks to the processor classes of the targeted heterogeneous platform as result of the parallelization step.

7.2.4 Experimental Results

To highlight the efficiency of the new approach, results are obtained for the same real-world embedded applications that were used for evaluation purposes in the previous chapters. To emphasize the quality of the results exploited by the new approach, a comparison with the homogeneous pipeline parallelization approach presented in Section 5.3 and the heterogeneous task-level parallelization approach presented in Section 7.1 is demonstrated here.

The results were obtained for the same target architecture (ARM11QuadProc architecture (cf. Section 3.3.2)) simulated by the cycle-accurate Vast MPSoC simulator [Syn13b] that was also used to evaluate the heterogeneous task-level parallelization approach in the previous section. All approaches are (re-)evaluated for the new target platform configuration. To emphasize the adaptability of the approach to multiple architectures, results are presented for the same two configurations as used in Section 7.1.5. Platform configuration (A) contains four ARM cores running at 100 MHz (1×), 250 MHz (1×), and 500 MHz (2×) while platform configuration (B) is configured to provide two 200 MHz and two 500 MHz cores.

7.2.4.1 Evaluation of Speedup

The presented platforms were also evaluated for the two application scenarios:

I) The main processor of the platform is the slowest one (100 MHz) and the additional cores are added as accelerators.

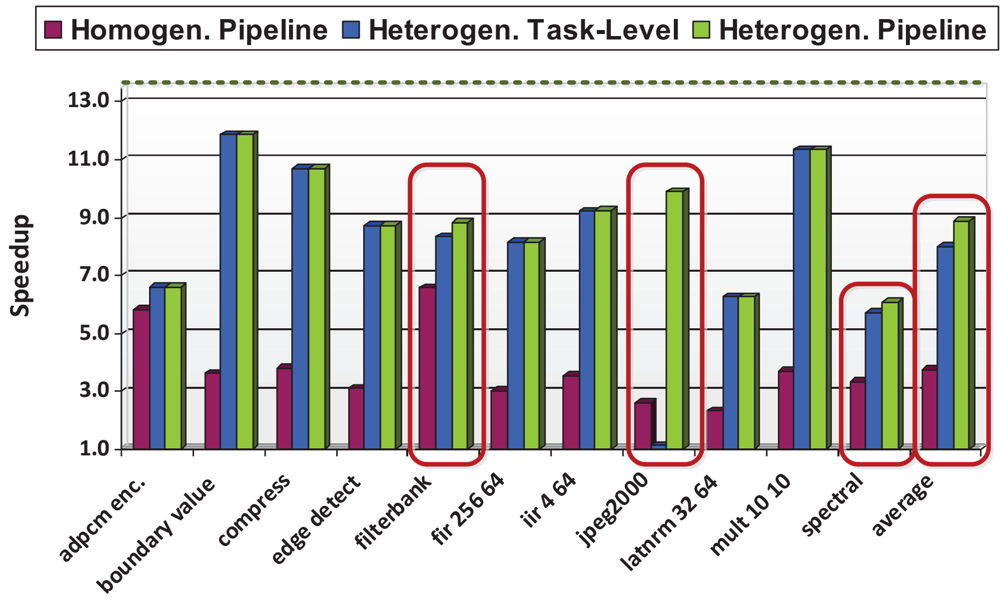


Figure 7.22: Results for Platform Configuration(A): 100/250/500/500MHz in Scenario(I)

II) The main processor of the platform is the fastest one (500 MHz) and the other (slower) processors are added to the platform due to, e.g., power or thermal issues.

These scenarios are the corner cases for the considered target architectures and were also used in Section 7.1.5 to ease comparability. The measurement baseline in both scenarios is the sequential execution on the main processor for all evaluated approaches. Figures 7.22 and 7.23 depict results for both evaluation scenarios for platform configuration (A) (100/250/500/500 MHz) and compare the new heterogeneous pipeline parallelization approach to the ones presented in Sections 5.3 and 7.1. The dashed line represents the theoretical maximum speedup of the considered target platforms in all figures.

Results for platform configuration (A) and the accelerator scenario (I) are shown in Figure 7.22. As can be seen, all three approaches increase the performance of all evaluated applications well. The homogeneous pipeline parallelization approach presented in Section 5.3 tries to balance the workload for all available processors uniformly. Speedups between 3× up to 4× are achieved for most applications, which are good results for a homogeneous architecture providing four processing units. However, the considered target architecture is heterogeneous and only a small amount of the available performance could be exploited. In contrast, results generated by the heterogeneous task-level parallelization approach presented in Section 7.1 and the new heterogeneous pipeline parallelization approach of this section are much more impressive. Both of them automatically balance the extracted tasks by respecting different performance characteristics of the available processing units. As a result, the two processors running at 500 MHz are automatically allocated with

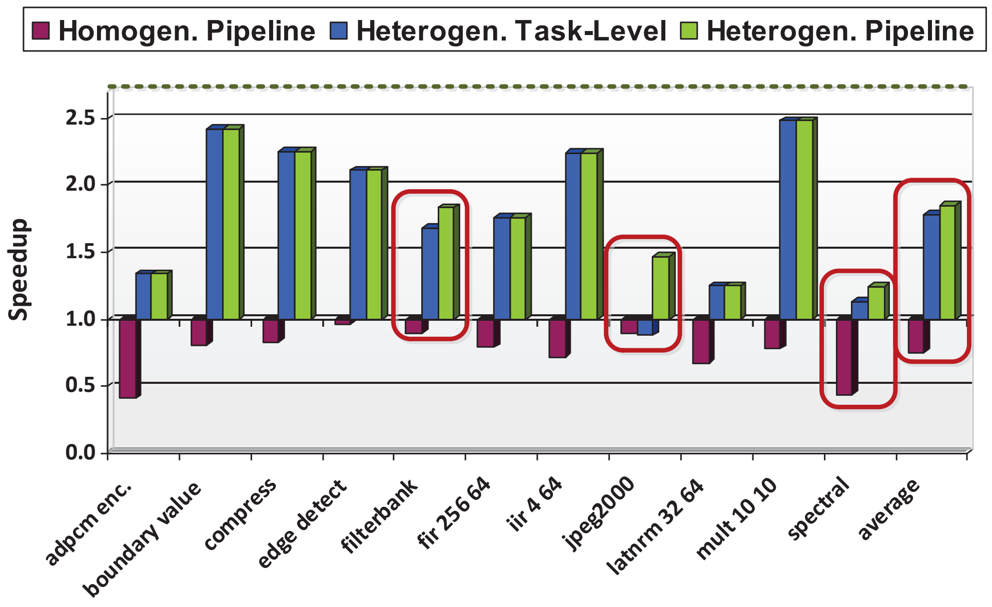


Figure 7.23: Results for Platform Configuration(A): 500/500/250/100MHz in Scenario(II)

heavier workloads than the slower ones. This results in performance increases of up to 10-12× for some of the considered benchmarks (e.g., *boundary value*, *compress* and *mult*) which significantly outperforms the speedup of the homogeneous pipeline parallelization approach and is close to the theoretical maximum speedup of 13.5×[8]. However, even if the heterogeneous task-level parallelization approach extracts comparable speedups to the newly presented pipeline parallelization approach of this section for many applications, it is outperformed for three of the considered benchmarks (*filterbank*, *jpeg2000*, *spectral*). The highest difference was observed for the *jpeg2000* encoder. Here, even the homogeneous pipeline parallelization approach extracted a more efficient parallel solution (2.6×) than the heterogeneous task-level approach (only 1.1×). This shows that for some embedded applications pipeline parallelism is most efficient. In contrast to both other approaches, the technique presented in this section is able to extract a speedup of nearly 10× which significantly outperforms the two earlier presented ones. On average, the homogeneous pipeline parallelization approach increased the applications' performance by 3.8× while the heterogeneous task-level parallelization approach reached speedups of on average 8×. In contrast, the new heterogeneous pipeline approach presented in this section reached an average speedup of nearly 9×.

Figure 7.23 depicts results for platform configuration (A) with application scenario (II) with a fast main processor (500MHz) and slower additional cores. Here, the speedup produced by the homogeneous pipeline parallelization approach is less than 1.0 which means that the parallelized application performs slower than its sequential version. The reason for this is that the homogeneous approach uni-

[8]Theoretical speedup limit: $(1 * 100 + 1 * 250 + 2 * 500\text{MHz})/100\text{MHz} = 13.5\times$

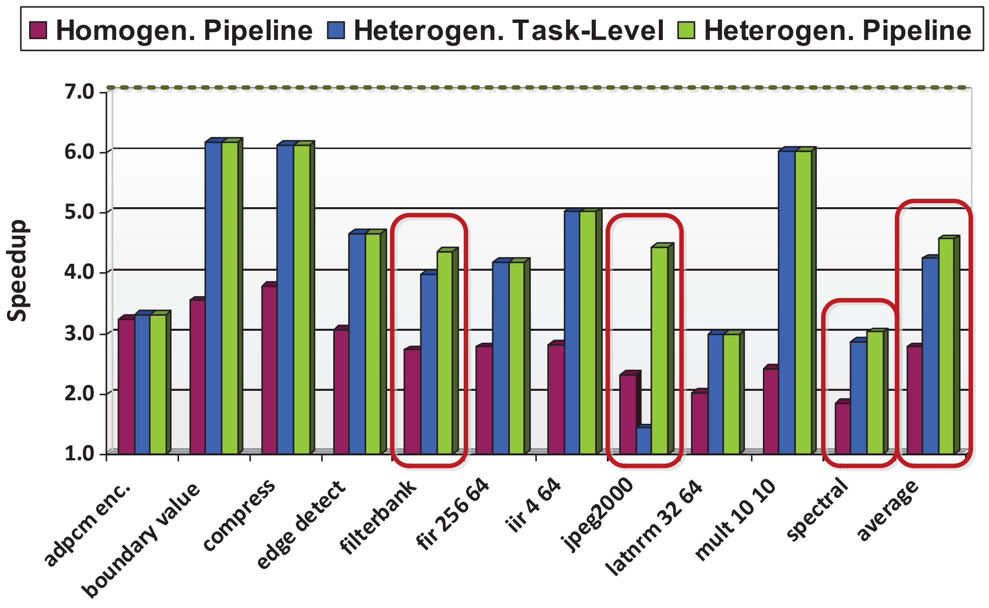


Figure 7.24: Results for Platform Configuration(B): 200/200/500/500MHz in Scenario(I)

formly distributes the work to the available processing units without considering information about varying performance characteristics of the available processing units. Thus, the fast main processor has to wait until the slower processing units have finished their tasks. In contrast, both heterogeneous parallelization approaches were able to speed up the applications by generating tasks that perfectly utilize the slower processing units so that all cores finish nearly at the same time. Again, the speedup of the applications *filterbank*, *jpeg2000*, and *spectral* could be increased by the new heterogeneous pipeline parallelization approach which outperforms both earlier presented approaches. It should also be mentioned that the new heterogeneous pipeline parallelization approach has extracted a higher speedup for all evaluated benchmarks in both application scenarios compared to the existing homogeneous pipeline parallelization approach. In addition, it performed better for more than one quarter of all evaluated benchmarks compared to the efficient heterogeneous task-level parallelization approach and otherwise never generated slower solutions. On average, the homogeneous pipeline parallelization approach decreased the applications' performance to 0.8× for this application scenario while the heterogeneous task-level parallelization approach was able to increase the applications' performance by approximately 1.8×. In contrast, the speedup of the new heterogeneous pipeline parallelization approach nearly reached 1.9×[9] for this platform configuration.

Similar results could also be observed for platform configuration (B) with two 200 MHz and two 500 MHz cores (cf. Figures 7.24 and 7.25). The performance difference between all three approaches is less than the performance difference for platform configuration (A) since the theoretical speedup limit is lower for platform

[9]Theoretical speedup limit: $(1 * 100 + 1 * 250 + 2 * 500\text{MHz})/500\text{MHz} = 2.7\times$

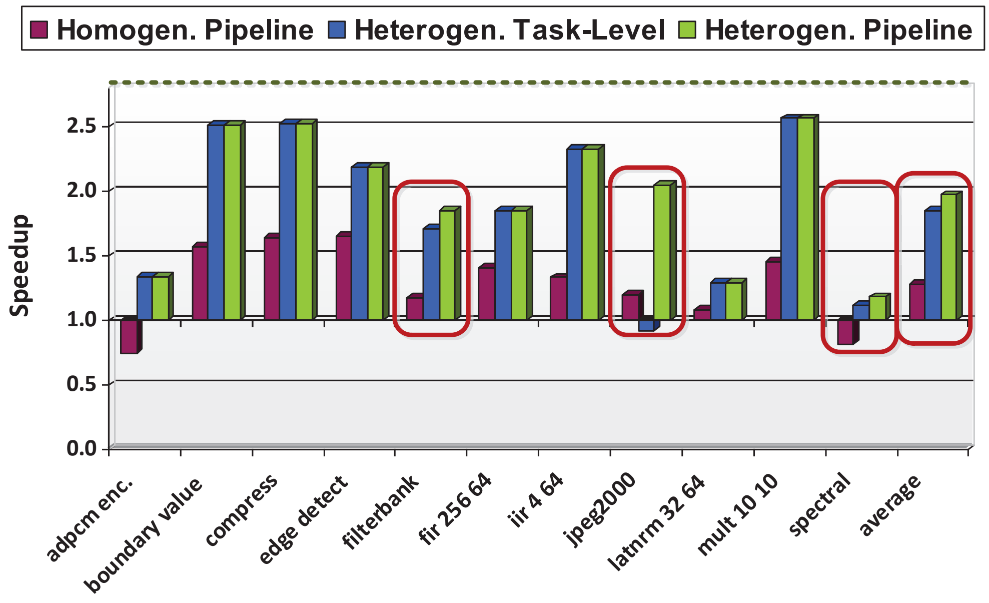


Figure 7.25: Results for Platform Configuration(B): 500/500/200/200MHz in Scenario(II)

configuration (B). Nevertheless, the relation between increased speedups and the theoretical speedup limit is similar to both platform configurations. Here, the three applications already mentioned, *filterbank*, *jpeg2000*, and *spectral*, profit most from the new heterogeneous pipeline parallelization approach. The average speedups for scenario (I) are 2.8×, 4.2× and 4.6× for the evaluated approaches with a theoretical speedup limit of 7×[10] for this platform configuration. The speedups of application scenario (II) with the slower additional cores are 1.3×, 1.8× and 2× with a theoretical speedup limit of 2.8×[11].

7.2.4.2 Optimization Time & ILP Statistics

The newly presented ILP-based pipeline parallelization approach for heterogeneous architectures has the longest execution time compared to the other two approaches. The time to parallelize the evaluated benchmarks with the three approaches is summarized on the left-hand side of Table 7.3. The timings shown are those which were necessary to extract the parallel solutions for platform configuration (A) with evaluation scenario (I).

The homogeneous pipeline parallelization approach (Hom Pip) performs faster than both other approaches while the heterogeneous task-level parallelization approach (Het TL) performs faster than the newly presented heterogeneous pipeline parallelization approach (Het Pip). The reason for this is that the complexity of the solution space increases between all three approaches from left to right. Nevertheless, the quality of the extracted solutions also increases with the complexity of

[10]Theoretical speedup limit: $(2 * 200 + 2 * 500\text{MHz})/200\text{MHz} = 7\times$

[11]Theoretical speedup limit: $(2 * 200 + 2 * 500\text{MHz})/500\text{MHz} = 2.8\times$

	Execution Times[12]			ILP Statistics New Approach			
Benchmark	Hom Pip	Het TL	Het Pip	#ILPs	#Var	#Const	%Opt.
adpcm enc.	00:00:02	00:00:18	00:12:52	4	3,235	16,502	75%
bound. value	00:00:45	00:00:15	00:20:22	2	19,538	157,421	0%
compress	00:06:40	00:10:41	00:52:30	9	8,387	42,239	44%
edge detect	00:06:50	00:03:15	00:00:04	6	1,746	3,592	100%
filterbank	00:00:40	00:15:34	00:04:32	2	1,445	5,065	100%
fir 256 64	00:00:07	00:00:02	00:35:04	3	313,039	2,767,507	0%
iir 4 64	00:14:33	00:00:07	00:10:03	1	3,415	24,061	0%
jpeg2000	00:02:03	01:09:56	01:23:09	11	36,995	274,928	27%
latnrm 32 64	00:01:19	00:00:04	00:10:03	1	3,296	23,577	0%
mult 10 10	00:04:17	00:00:06	00:51:06	5	66,814	555,567	0%
spectral	00:00:17	00:17:00	00:30:32	4	13,766	103,071	25%
average	00:03:25	00:10:40	00:28:12	5	42,880	361,230	34%

Table 7.3: Comparison of Homogeneous and Heterogeneous Parallelization Algorithms & Statistics of New Heterogeneous Pipeline Parallelization Approach

the approaches as shown earlier by the extracted speedups. On average, the homogeneous pipeline parallelization approach took around 3.5 minutes to parallelize the considered applications. But many of them could be processed in less than a minute. In contrast, the heterogeneous task-level parallelization approach took 10 minutes on average to parallelize the applications. The newly presented heterogeneous pipeline parallelization approach processed the considered applications in 28 minutes on average while many applications were parallelized in around 10.5 minutes. However, the execution time of the approach can significantly be reduced by parallelizing the parallelization approach itself due to the hierarchical divide-and-conquer-based approach of the global parallelization algorithm. Nevertheless, the high speedups outweigh the higher execution times of the new approach in most cases and are acceptable since parallelization has to be done only once in the compilation process.

The right-hand side of Table 7.3 summarizes ILP Statistics for the newly presented heterogeneous pipeline parallelization approach. 5 ILP systems had to be solved in the average case containing approximately 43k variables and 362k constraints summed up over all ILP systems. But the most interesting statistic is shown in the last column of Table 7.3. Here, the relative amount of optimally solved ILPs is shown. Only one third of all ILPs could be solved optimally in the configured amount of time (five minutes) on average[13]. In the other cases, the ILP returned the best solution found so far. However, this solution was already close or equal to the optimal solution (but the solver could not prove that the solution is optimal) so that the reported high speedups could be extracted. Nevertheless, this also shows that the complexity of the presented parallelization approach reached a dimension which is borderline for ILPs.

To summarize, the following results were achieved:

[12]Time format HH:MM:SS, measured on an AMD Opteron core running at 2.4 GHz

[13]The ILPs are solved multiple times for different input parameters with 5 minutes, each.

1. The newly presented heterogeneous pipeline parallelization approach is able to utilize heterogeneous platforms in an excellent way. Speedups of up to 10-12× were measured for evaluation platform (A) with a theoretical speedup limit of 13.5×.

2. The integration of mapping decisions and platform information in a heterogeneous parallelization approach is highly beneficial.

3. The new parallelization approach outperformed all previously presented ones. For the *jpeg2000* encoder, a speedup of nearly 10× could be measured compared to 2.6× and 1.1× for the homogeneous pipeline and heterogeneous task-level parallelization approaches for platform configuration (A) in scenario (I).

7.3 Summary

This chapter presented two parallelization approaches optimized for special requirements of heterogeneous embedded MPSoCs. Both, the proposed task-level and the proposed pipeline parallelization approaches, can be executed independently or in a combined fashion to extract well-balanced solutions even for target architectures providing processing units with different performance characteristics. For the new heterogeneous ILP-based parallelization approaches, the homogeneous ones presented in Chapter 5 were used as a starting point. The evaluation of both approaches has shown that the new heterogeneous ones are able to create well-balanced solutions which significantly outperform the homogeneous approaches on heterogeneous target platforms. To summarize, the following results could be achieved in this chapter:

1. Creation and integration of a sophisticated task-level parallelization approach optimized for heterogeneous embedded MPSoCs.

2. Creation and integration of a complex pipeline parallelization approach optimized for heterogeneous embedded MPSoCs.

3. Combination of task-level, data-level[14], and pipeline parallelization approaches which can be executed separately or in a combined fashion.

4. Creation of parallelization techniques combining task creation, communication and execution costs in high-level models to balance the extracted tasks even for processing units with varying performance characteristics automatically.

5. Highly efficient solutions could be extracted with speedups of up to 10-12× for an architectures with a theoretical speedup limit of 13.5×.

[14]DoAll parallelism is extracted by the simplistic loop and the pipeline parallelization approaches.

Chapter 8

Multi-Objective aware Parallelization for Heterogeneous MPSoCs

Contents

Parallelization extraction techniques optimized for heterogeneous embedded MPSoCs have already been presented in the previous chapter. The evaluation has shown that the ILP-based approaches are able to balance solutions well even for architectures providing processing units with different performance characteristics. As a result, solutions that are close to the theoretical speedup limit could be extracted. These solutions exploit the performance of heterogeneous systems in an optimized way.

However, the main purpose of heterogeneous embedded MPSoCs is not the extraction of as much speedup as exploitable. Often, the specialized cores available in those systems can be used to find excellent trade-offs between additionally considered optimization objectives, like, e.g., energy consumption. Figure 7.1 on page 138, for example, depicts the exploitable trade-offs between power consumption and performance of the Cortex-A7 and Cortex-A15 cores of ARM's big.LITTLE architecture [Pet13]. While the (little) Cortex-A7 cores provide only low performance, they consume significantly less energy compared to the (big) Cortex-A15 cores. One of the reasons is the simple processor pipeline employed in the Cortex-A7 cores. In contrast, the Cortex-A15 cores provide significantly more performance at the expense of higher energy consumption. The combination of both processors provides a large optimization space for multi-objective aware parallelization approaches. Besides solutions which execute the whole application either on Cortex-A7 or Cortex-A15 cores, various solution candidates lie in between these corner cases executing more or less parts of the application on the slower or faster processing units in parallel. This spans a large solution space containing solutions providing high performance, low energy consumption, or also useful trade-offs between multiple objectives. Other heterogeneous embedded target platforms (cf. Chapter 7) provide such trade-offs as well.

The Genetic Algorithm (GA)-based parallelization approaches presented in Chapter 6 were able to extract efficient trade-offs between multiple objectives directly in the parallelization process and were optimized for homogeneous embedded MPSoCs. The new perspective provided by heterogeneity of the available processing units renders a large optimization potential going far beyond the possibilities provided by homogeneous architectures. However, the approaches presented in Chapter 6 are not aware of varying performance characteristics of the provided processing units of a heterogeneous MPSoC. This leads to poor results for such architectures as already shown for the homogeneous ILP-based approaches applied to heterogeneous architectures (cf. Section 7.2.4). Therefore, this chapter presents new approaches optimized for heterogeneous target architectures extracting parallelism in a multi-objective aware manner. Since the efficiency of the GA-based approaches presented in Chapter 6 could be demonstrated on several real-world embedded applications, they are used as a starting point for the new heterogeneous parallelization approaches presented in this chapter.

Also here, a combination of task-level and pipeline parallelism is also employed to be able to extract parallelism from embedded applications covering a large number of domains. Both approaches can be executed separately or in a combined fashion like employed by the parallelization approaches presented in the previous chapters. Since many basic concepts were entirely discussed before, this chapter presents both approaches in a shorter way by combining most sections of both approaches. Accordingly, the rest of this chapter is structured as follows: The integration of the two multi-objective aware parallelization approaches for heterogeneous embedded MPSoCs into the global parallelization technique is described first in Section 8.1. Afterwards, both, the task-level as well as the pipeline parallelization approaches are presented in Section 8.2 describing all parts which have to be provided for a GA-based optimization technique. Finally, Section 8.3 presents results for a combined execution of both approaches before Section 8.4 summarizes the approaches presented in this chapter.

8.1 Integration into the Global Parallelization Approach

The integration of the multi-objective aware GA-based task-level parallelization approach optimized for heterogeneous embedded MPSoCs into the global parallelization framework (cf. Section 4.2) is shown in Algorithm 11. The function ExtractHetGATLP is executed by the global parallelization algorithm (cf. Al-

Algorithm 11 Pseudo Code of the GA-based Task-Level Parallelization Approach

```
1: // Called bottom-up hierarchically by Algorithm 4 in line 18 on page 56
2: function ExtractHetGATLP(Node $n$, Platform $pf$, int $maxTasks$)
3:     // This function is only applicable to hierarchical nodes.
4:     if IsNotHierarchicalNode($n$) then
5:         return $\emptyset$
6:     end if
7:     // Extract parallelism for hierarchical node n.
8:     // All nodes deeper in the hierarchy are already processed.
9:     $initPopul \leftarrow$ CreateInitialPopulation($n, maxTasks$)
10:    $finalPopul \leftarrow$ HetGATaskLevelParallelizer($n, initPopul, pf, maxTasks$)
11:    $front \leftarrow$ ExtractParetoFrontier($finalPopul$)
12:    return $front$
13: end function
```

gorithm 4) as soon as all child nodes deeper in the hierarchy are processed. As arguments, the function expects the node n to be parallelized, platform specific information pf containing, e.g., the performance characteristics and the number of available processing units, and an upper bound of extractable tasks $maxTasks$. The structure of this algorithm is comparable to the one presented for the homogeneous multi-objective aware task-level parallelization approach (cf. Algorithm 7) since the changes made are hidden in the *HetGATaskLevelParallelizer* method which is called in line 10 to start the GA-based approach. To summarize, the algorithm first determines in lines 4-6 whether the currently processed node is not a hierarchical one since only those nodes are processed by the presented approach. Afterwards, an initial population is created in line 9 containing randomly generated solution candidates and the sequential solution executed on all processor classes. This initial population is then used in line 10 to extract the final population with the help of the Genetic Algorithm-based parallelization approach presented in this section. Finally, the front of Pareto optimal solutions is determined in line 11 and returned as the result of the presented approach in line 12.

The integration of the multi-objective aware pipeline parallelization approach is structured in a similar way to Algorithm 11 and only differs in two points. First, lines 4-6 determine whether the currently processed node represents a loop statement since only those statements are parallelized by this parallelization method. Second, the pipeline parallelization approach operates on an augmented PDG (cf. Section 5.3.2) so that it is extracted in line 9 first, before the initial population is created and the GA-based approach is executed.

The outer parallelization algorithm (cf. Algorithm 4) has the possibility to further combine the results of these approaches with additional ones extracted by other parallelization approaches, due to the divide-and-conquer based parallelization method. Thus, each processed application can benefit from multiple parallelization methods.

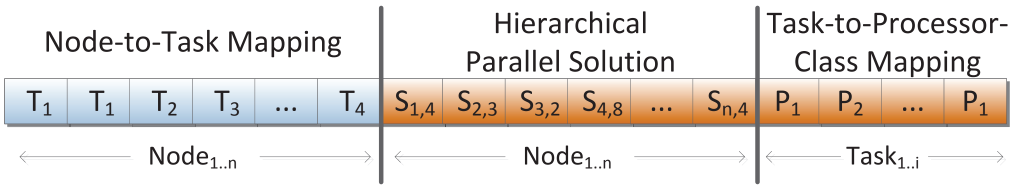

Node-to-Task Mapping
Hierarchical Parallel Solution
Task-to-Processor-Class Mapping
T_1 T_1 T_2 T_3 ... T_4
$S_{1,4}$ $S_{2,3}$ $S_{3,2}$ $S_{4,8}$ ... $S_{n,4}$
P_1 P_2 ... P_1
$Node_{1..n}$
$Node_{1..n}$
$Task_{1..i}$

Figure 8.1: Individual's Chromosome Structure for Heterogeneous Task-Level Parallelism

8.2 GA-based Approaches for Heterogeneous Architectures

According to Section 6.1, an efficient chromosome representation, objective evaluation functions as well as mutation and cross-over functions have to be provided for a GA-based optimization process. Again, the PISA framework [BLT+03] with SPEA2 [ZLT01] is used for selection and variation purposes. Therefore, the aforementioned parts of the employed GA-based approach are discussed in the following. While the chromosome representations for task-level (cf. Section 8.2.1) and pipeline parallelism (cf. Section 8.2.2) are described separately, the objective functions (cf. Section 8.2.3) as well as the mutation and cross-over techniques (cf. Section 8.2.4) employed are summarized in separate sections. The following section highlights the performed changes compared to the multi-objective aware parallelization approaches for homogeneous MPSoCs.

8.2.1 Chromosome Structure for Task-Level Parallelism

The chromosome structure presented in this section is able to extract task-level parallelism according to the fork-join model explained in Section 5.2.3. The approach presented here also operates on the Augmented Hierarchical Task Graph (AHTG) with its global divide-and-conquer-based parallelization approach to extract and combine parallelism with different granularities (cf. Chapter 4). Hence, small groups of statements, different loop(-nest)s or also function calls may be executed in parallel. Figure 8.1 depicts the developed chromosome structure used by the novel multi-objective aware task-level parallelization approach for heterogeneous embedded MPSoCs. Each hierarchical node of the AHTG is processed in isolation to extract parallelism from sequentially written applications. Since the new approach has to cope with heterogeneous MPSoCs, the extracted tasks are also directly mapped to processor classes of a heterogeneous MPSoC. All nodes deeper in the hierarchy are already processed, due to the bottom-up approach of the global parallelization technique. As a result, a front of Pareto-optimal solutions $S_{i,j}$ exists for each child node N_i containing solution candidates that might implement parallelism deeper in the hierarchy.

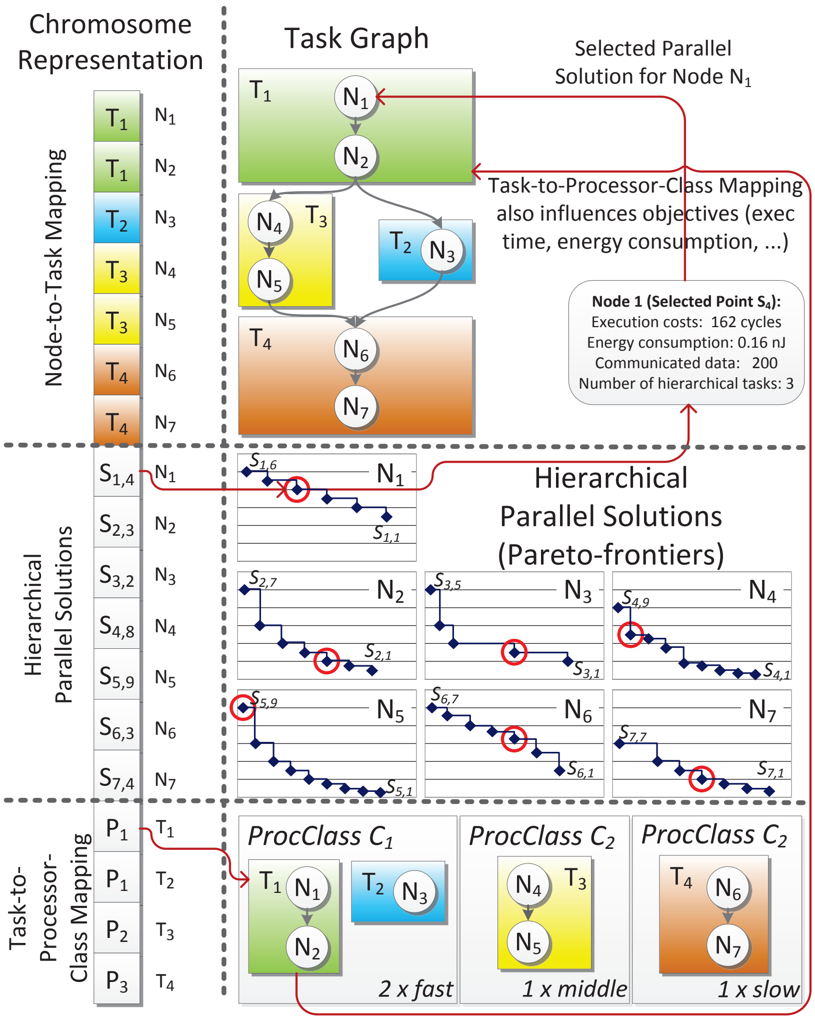
The first part of the employed chromosome structure maps each child node of the node to be parallelized to newly extracted tasks (cf. Figure 8.1). The second one chooses one hierarchical solution for each child node. These parts of the chromosome structure were also used in the homogeneous representation described in Section 6.2. However, to be able to extract efficient parallelism for heterogeneous multi-core architectures, a third part was added mapping newly extracted tasks to processor classes of the targeted MPSoC. This enables the possibility of optimizing extracted tasks for specific processing units while balancing the overall execution behavior. In the example shown in Figure 8.1, nodes N_1 and N_2 are mapped to task T_1 while node N_3 is mapped to task T_2 and so on. In addition, solution $S_{1,4}$ of child node N_1's Pareto front is chosen as hierarchical solution while $S_{2,3}$ is chosen for child node N_2. Task T_1 is mapped to processor class P_1 while task T_2 is mapped to processor class P_2.

Each chromosome is encoded as an array of integers. The size of each chromosome is twice as large as the number of direct child nodes n contained in the hierarchical node to be parallelized ($1\times$ node to task mapping $+$ $1\times$ hierarchical solution) plus the maximum number of extractable tasks[1] i for the task-to-processor class mapping. Each chromosome can be encoded efficiently by an array of $2 \times n + i$ integers enabling the creation of a large amount of individuals that require only a low amount of memory.

The impact on the evaluation of the individuals' objectives is depicted in Figure 8.2. The example is based on the one used to describe the impact of the homogeneous chromosome structure, extended by the new task-to-processor class mapping. As shown, a hierarchical node with seven child nodes which can be mapped to four newly created tasks on a platform providing four processing units grouped into three processor classes is processed here. The figure shows the genes' values on the left-hand side and their impact on the evaluation on the right-hand side. The upper part of the figure shows the task graph representation of the node to be parallelized according to the node-to-task mapping defined on the left-hand side. As can be seen, nodes N_1 and N_2 belong to task T_1 while node N_3 belongs to task T_2. Edges between the created tasks depend on the node-to-task mapping. Here, a dependence edge between node N_2 and N_3 exists which implicitly adds a dependence relation between tasks T_1 and T_2. Thus, the execution of task T_2 has to wait for completion of T_1 since data has to be communicated between both tasks before T_2 can start with its execution.

The second part of the chromosome representation contains the selection of hierarchical parallel solution candidates for all child nodes. All child nodes deeper in the hierarchy are already processed by the GA-based parallelization technique, due to the bottom-up approach. Therefore, a front of Pareto-optimal solutions exists for each child node evaluated by the high-level functions presented in Section 8.2.3. The frontiers contain solution candidates with parallelism which was found deeper in the hierarchy. The approach has to choose one solution candidate from each child node's Pareto-frontier providing different objective values for the corresponding node. A solution with more extracted parallelism may, for example, reduce the

[1]The maximum number of extractable tasks is set to the number of available processing units by default but can be changed by the user.



Chromosome Representation
Task Graph
Selected Parallel Solution for Node N1
Node-to-Task Mapping
T1 N1
T1 N2
T2 N3
T3 N4
T3 N5
T4 N6
T4 N7
Task-to-Processor-Class Mapping also influences objectives (exec time, energy consumption, ...)
Node 1 (Selected Point S4):
Execution costs: 162 cycles
Energy consumption: 0.16 nJ
Communicated data: 200
Number of hierarchical tasks: 3
Hierarchical Parallel Solutions
S1,4 N1
S2,3 N2
S3,2 N3
S4,8 N4
S5,9 N5
S6,3 N6
S7,4 N7
Hierarchical Parallel Solutions (Pareto-frontiers)
Task-to-Processor-Class Mapping
P1 T1
P1 T2
P2 T3
P3 T4
ProcClass C1
ProcClass C2
ProcClass C2
2 x fast
1 x middle
1 x slow

Figure 8.2: Impact of Chromosome Configuration on the Parallelized Hierarchical Node

overall execution time at the cost of the system's energy consumption. Hence, this part of the chromosome's structure also influences the objectives' evaluation.

The last part of the chromosome structure defines the task-to-processor class mapping which is crucial if applications should be parallelized for heterogeneous MPSoCs. Execution time, energy consumption, and other objectives depend on the processing unit used for a task. Therefore, these gene values map the tasks created in the first part of the chromosome structure to processor classes representing identical processing units. In the example shown, task T_1 which contains nodes N_1 and N_2, is mapped to processor class C_1 while task T_3, which executes the statements of nodes N_4 and N_5, is mapped to processor class C_2. Also here, the genes' values directly influence the evaluation of all objective values.

To summarize the chromosome representation, new tasks can be extracted, mapped to processor classes, and can also be combined with tasks which were found deeper in the hierarchy. Thereby, all necessary parts are covered to extract ef-

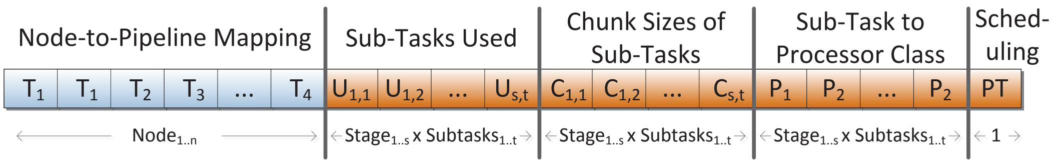


Figure 8.3: Individual's Chromosome Structure for Heterogeneous Pipeline Parallelism

ficient task-level parallelism from sequentially written applications optimized for heterogeneous embedded MPSoCs. More details on the evaluation of the considered objectives are presented in Section 8.2.3 just after the following Section presenting the developed chromosome structure for pipeline parallelism.

8.2.2 Chromosome Structure for Pipeline Parallelism

This section adapts the existing multi-objective aware pipeline parallelization approach for homogeneous MPSoCs presented in Section 6.3. The new approach is able to extract well-balanced parallelism for heterogeneous architectures. The adaptation of the pipeline parallelization approach required more changes while moving from the homogeneous to the heterogeneous case compared to the changes necessary to port the task-level parallelization approach. The main goal of pipeline parallelism is the extraction of concurrently executed pipeline stages (horizontal splits). The iterations of these stages can further be distributed into sub-tasks (vertical splits). These goals, as well as iteration balancing and processor class mapping, have to be extracted by the new GA-based pipeline parallelization approach targeting heterogeneous MPSoCs.

The developed chromosome structure is depicted in Figure 8.3. The first part of the chromosome maps child nodes of the loop to be parallelized to disjunct pipeline stages. This part is the only one that could be re-used from the homogeneous approach presented in Section 6.3 without changes. The two succeeding parts of the chromosome structure (namely *Sub-Tasks Used* and *Chunk Sizes of Sub-Tasks*) divide the different iterations of pipeline stages to concurrently executed sub-tasks. Finally, the *Sub-Task to Processor Class* and *Scheduling* genes map the extracted tasks to processing units of the target architecture and define the scheduling policy of loop iterations. An example using the new chromosome representation is shown in Figure 8.4. There, nodes N_1 and N_2 are mapped to pipeline stage T_1 while node N_3 is mapped to pipeline stage T_2. As a result, two pipeline stages are extracted which can be executed in a pipelined manner. A dependence edge exists between both pipeline stages which limits the achievable speedup of this solution since each iteration of pipeline stage T_2 waits for data generated by T_1. Therefore, the second part of the chromosome structure (called Sub-Tasks Used) defines whether sub-tasks are created for a given pipeline stage[2]. In the given example, three sub-tasks are used for T_1 like defined by the boolean Sub-Tasks Used genes. Thus, this pipeline

[2]The maximum number of generated sub-tasks can also be defined by the user and is set to the number of available processing units by default.

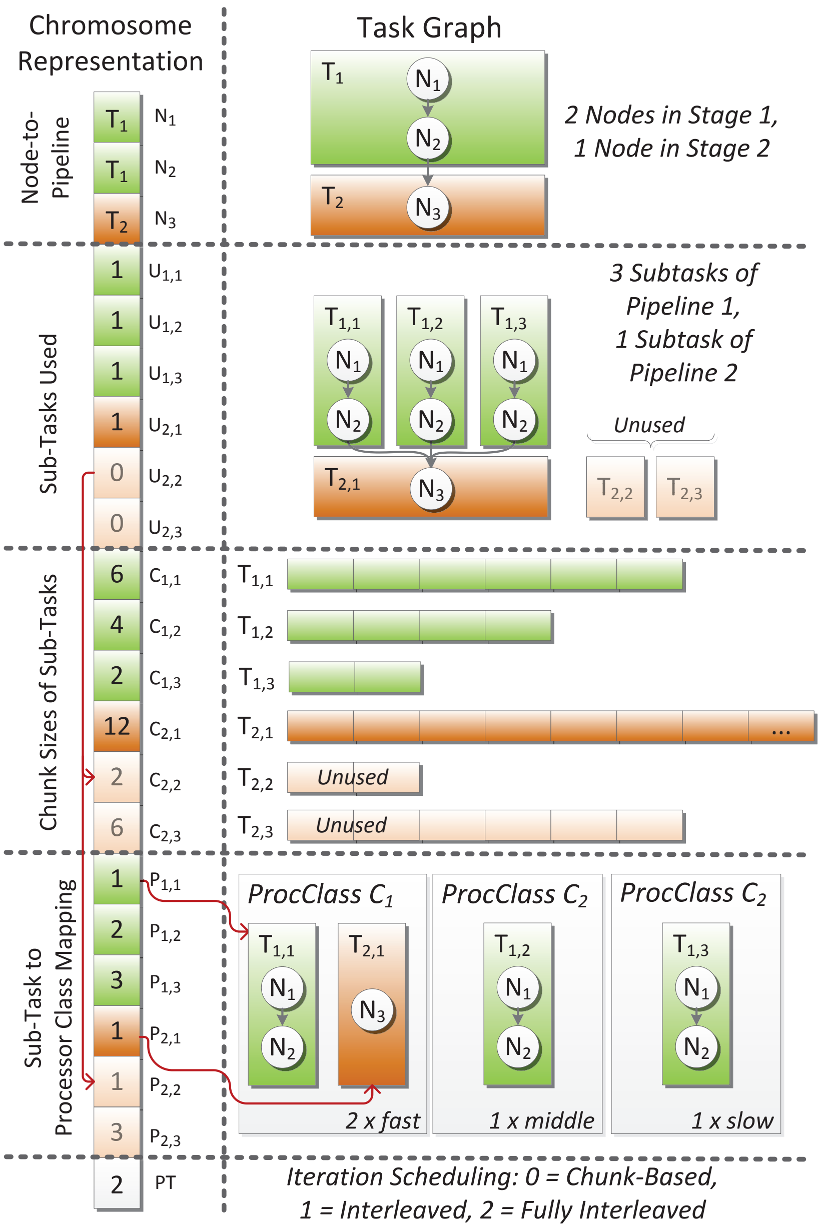


Figure 8.4: Impact of Chromosome Configuration on the Parallelized Hierarchical Node

stage is processed in parallel to faster supply pipeline stage T_2 with the required input data. The genes' values further depict that pipeline stage T_2 uses only one sub-task and is not split into concurrently executed sub-tasks.

So far, pipeline stage extraction and sub-task generation are encoded in the chromosome structure. This structure is – in general – sufficient if applications are to be parallelized for homogeneous architectures. For heterogeneous ones, however, the execution time of a task's iteration may change depending on the executing processing unit. Therefore, the third gene block allocates chunk sizes of the pipeline stages' iterations to the according sub-tasks. In this way, sub-tasks executing larger chunks can, e.g., be executed on faster processing units to balance the overall execution behavior. In the example of Figure 8.4, sub-task $T_{1,1}$ executes 6 iterations while sub-tasks $T_{1,2}$ and $T_{1,3}$ execute 4 and 2 iterations, respectively. Sub-task $T_{2,1}$ of pipeline stage T_2 executes all 12 iterations of the loop. The chunk sizes of $T_{2,2}$ and $T_{2,3}$ are ignored since the sub-tasks are not used according to the zero values in the last two positions of the *Sub-Tasks Used* genes. The chunk sizes are not directly taken from the genes' values because the probability that the sum of all chunk sizes per pipeline stage is equal to the number of loop iterations is quite low. Since all loop iterations of a pipeline stage must be executed exactly once by the generated sub-tasks, a solution with less or more executed iterations is invalid. To avoid this, the chunk sizes are translated into percentage-values so that the iterations are mapped to the sub-tasks according to these percentages.

The fourth block of gene values maps the extracted sub-tasks to the processor classes of the targeted heterogeneous MPSoC. In the given example, sub-task $T_{1,1}$ is mapped to processor class C_1 which contains two fast processors. This is a good choice for this scenario since $T_{1,1}$ executes more iterations than both other sub-tasks of pipeline stage T_1 like defined by the corresponding genes (Chunk Sizes of Sub-Tasks). $T_{1,2}$ and $T_{1,3}$ are mapped to processor class C_2 and C_3 with respect to their execution load.

The last position in the chromosome structure is reserved to select the iteration scheduling policy. Three different scheduling strategies are supported, namely *chunk-based*, *interleaved* and *fully-interleaved*. The first one maps all iterations continuously to one sub-task respecting its chunk size. Here, sub-task $T_{1,1}$ would be allocated loop iterations $\{0, .., 5\}$ while $T_{1,2}$ would be allocated iterations $\{6, .., 9\}$, respectively. This scheduling policy has a good data cache locality, but all succeeding iterations are executed by the same task so that tasks waiting for the generated data do not obtain the data of iterations concurrently. The *interleaved* policy executes smaller chunk sizes in an interleaved manner. Thus, task $T_{1,1}$ could, for example, execute iterations $\{0{-}2, 6{-}8\}$ while $T_{1,2}$ and $T_{1,3}$ could execute iterations $\{3{-}4, 9{-}10\}$ and $\{5, 11\}$, respectively. This scheduling policy is a good trade-off between cache locality and a faster response time for waiting tasks. The last scheduling policy, namely *fully-interleaved*, schedules the iterations in a way that all tasks start with the execution of a pipeline stage's iteration as soon as possible while respecting the configured chunk sizes. A possible schedule for this example with this scheduling methodology could be $\{0, 3, 5, 6, 9, 11\}$, $\{1, 4, 7, 10\}$, and $\{2, 8\}$ for tasks $T_{1,1}$, $T_{1,2}$, and $T_{1,3}$, respectively (cf. Figure 8.5). All three tasks execute one of the first three iterations concurrently. Afterwards, task $T_{1,3}$ starts with the next iteration after task $T_{1,1}$ has executed 3 iterations, like defined by the chunk size. In the future, the integration of additional scheduling strategies could be considered as well.

With the presented chromosome structure, it is possible to extract well-balanced pipeline parallelism optimized for heterogeneous MPSoCs. The timing behavior of the genes' values presented in Figure 8.4 is depicted in Figure 8.5. The node-to-task-mapping, the sub-task creation, the different chunk-sizes, the processor-class-

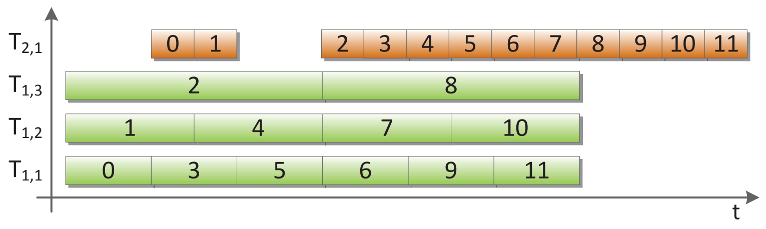


Figure 8.5: Timing of Example Depicted in Figure 8.4

mapping as well as the iteration scheduling are considered in the diagram. As soon as the first six iterations of pipeline stage T_1 have been executed, all sub-tasks are executing their work without any interruptions. This behavior would improve further if the loop would be executed for more than the shown twelve iterations. The solution depicted in Figure 8.4 has a good execution behavior regarding execution time. However, the new approach considers other objectives, such as energy consumption, as well. For this objective, other solutions with, e.g., a lower number of cores used may produce better results. Therefore, the new approach evaluates all considered objectives for all individuals so that a front of Pareto-optimal solutions is generated which also contains results with good trade-offs between the different objectives.

8.2.3 Objective Evaluation

This section defines the functions necessary to evaluate task-level and pipeline parallelization individuals. The evaluation functions presented in Chapter 6 were evaluated (by target platform simulation) to be accurate enough to estimate the parallel performance for homogeneous MPSoCs. Therefore, these models should be used as a basis for the new approaches.

8.2.3.1 Objective 1: Execution Time

The execution time of the longest execution path formed by the extracted tasks is used for the execution time of the parallelized node. This is valid since all tasks have to wait until the slowest one (or the longest execution path) has completed its work so that all tasks are joined before the parallel section is left.

Equations 6.1 - 6.3 (cf. page 114) define the employed model for task-level parallelism in a formal way. Fortunately, the adaptation of this model to heterogeneous architectures is straightforward. The only equation that has to be adapted is Equation 6.1 which estimates the costs of task T_i. In its original form, the formula does not distinguish between different execution times of the child nodes mapped to the task depending on the executing processing unit. Therefore, it is replaced by Equation 8.1.

$$ET(T_i) = TCO + \sum_{n \in Nodes(T_i)} ETN(n, S_{n,k}, c) \tag{8.1}$$

The execution times ETN of all nodes n mapped to task T_i depend on the chosen hierarchical solution candidate $S_{n,k}$ of node n and also on the selected processor class c. A similar adaptation had to be made to estimate the execution time of a pipeline parallelization candidate. Here, equation 6.9 defining $ET(T_i^j)$ for pipeline stage i in iteration j is changed in a similar way like denoted in Equation 8.2.

$$ET(T_i^j) = \sum_{n \in Nodes(T_i)} \frac{ETN(n,c)}{LI} \tag{8.2}$$

Equations 6.10 - 6.11 (cf. page 127) can be reused without any changes since the estimated path costs are based on $ET(T_i^j)$ which now distinguishes between different execution times depending on the mapped processor class. Thus, the path costs implicitly include this distinction without being modified.

8.2.3.2 Objective 2: Energy Consumption

The objective functions used to estimate the system's energy consumption for task-level and pipeline individuals have to be adapted in an analogous way to the first objective. The employed formulas used to evaluate task-level individuals for this objective on homogeneous architectures have already been presented in Equations 6.4 - 6.7 (cf. page 115). Only the estimation of the tasks' energy consumption has to be extended to take different processor classes into account. Equation 6.6 is replaced by the formula depicted in Equation 8.3.

$$E(T_i) = TCE + \sum_{n \in Nodes(T_i)} EEN(n, S_{n,k}, c) + ICE(T_i) + OCE(T_i) \tag{8.3}$$

In Equation 8.3, the energy consumption EEN consumed by all nodes n mapped to task T_i now depends on the chosen hierarchical parallel solution candidate $S_{n,k}$ and the executing processing unit c. Since the energy costs $E(T_i)$ are contained in the overall energy consumption defined in Equation 6.7, the different performance characteristics of the available processing units are implicitly part of the final objective value. A comparable adaptation had to be made to estimate the system's energy consumption of a pipeline parallelization candidate. Here, Equations 6.12 - 6.15 define the energy consumption's estimation for homogeneous MPSoCs (cf. page 128). All equations with an exception of Equation 6.14 determining the energy consumption of task T_i in iteration j could be reused without any changes. Equation 6.14 is replaced by Equation 8.4 so that this estimation also considers different energy costs for the execution on different processing units.

$$E(T_i^j) = ICE(T_i^j) + OCE(T_i^j) + \sum_{n \in Nodes(T_i)} \frac{EN(n,c)}{LI} \tag{8.4}$$

8.2.3.3 Objective 3: Communication Overhead

The third objective considered is the overhead introduced by communication which is independent of the executing processing units in the employed model. Therefore, this objective is determined in an identical way compared to the ones defined in Equations 6.8 and 6.16.

8.2.3.4 Portability of Models

Also in the heterogeneous case, constants like, e.g., the task creation overhead for execution time TCO or the communication cost multiplier $COSTS$ can be configured to be applicable to multiple target platforms. In addition, the objective values for the leaves of the AHTG are determined via target platform simulation so that they are also automatically adjusted for the considered target architecture.

8.2.4 Mutation & Cross-Over

The mutation and cross-over strategies that were previously presented in Chapter 6 are used as a basis for the new heterogeneous parallelization approach. The implemented standard mutation function employs a 1-position mutation strategy. One gene of the mutating individual's chromosome is randomly chosen and altered to a new value. The employed standard cross-over function (also called recombination) splits the chromosomes of two individuals at a random position and joins the left-hand side of the first one with the right-hand side of the second one and vice versa. Both functions lead to good solution candidates but have the drawback that they generate many invalid solution candidates ($> 98\%$). Solutions are invalid if, e.g., a deadlock is created by mutation or recombination so that one task is waiting for data of another one and vice versa. This can happen if a child node is moved from one task to another one since dependencies between the tasks may change, as well. The presented approach rejects such solutions and also ones with more concurrently executed tasks mapped to the same processor class as processing units are available in this class. As a result, additional scheduling overhead at runtime can be avoided.

This problem was also observed in Sections 6.2.4 and 6.3.4 so that smart mutation and cross-over strategies were developed. As soon as a valid solution becomes invalid after mutation or recombination, the algorithm determines the source of the problem and tries to fix it with a subsequently executed mutation. For example, to fix a deadlock, the target node which causes the deadlock is moved to a different task to solve the problem. Similarly, if too many tasks are allocated to a specific processor class after mutation, one of the other tasks is moved to a different processor class. These steps are repeated until the solution becomes valid again or a maximum number of fixing steps is reached.

By applying this strategy, the number of invalid solutions could be reduced from more than 98% to around 5% for the heterogeneous parallelization techniques, which significantly reduced the time which was necessary to extract efficient solution candidates. As stated earlier, it is clear to the author of this thesis that these modifications applied to the mutation and cross-over functions may influence the solutions generated by the GA-based approach. However, a negative influence on the solution quality could not be observed. Instead, the evaluation chapter will later

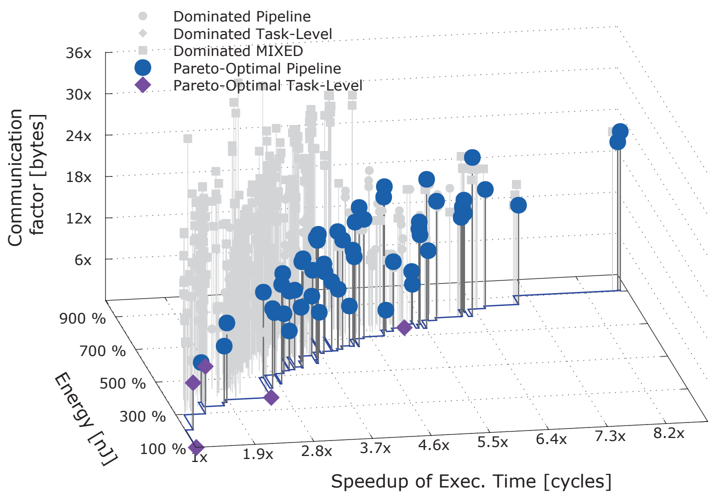


Figure 8.6: Final Parallel Solutions for the Edge Detect Benchmark.

show that the smart mutation and cross-over functions lead to efficient results in a short period of time.

8.3 Experimental Results

To evaluate the applicability of the newly presented multi-objective aware parallelization approaches for embedded heterogeneous MPSoCs, the same set of benchmarks used in the previous chapters was also employed here. As a target platform for the model-based evaluation, a same ISA-multi-core platform (like [Pet13]) configured with four ARM cores running at 100 MHz (1x), 250 MHz (1x) and 500 MHz (2x) to simulate a platform with large performance variances was chosen as an inspiring example. Unfortunately, all considered target platforms described in Section 3.3 lack an energy model for a heterogeneous system configuration. Therefore, the results in this section are only based on the evaluation models presented in Section 8.2.3. An evaluation of the employed models could be performed in future work as soon as a platform providing such a model exists.

Detailed results for four of the considered applications can be found in Figures 8.6 - 8.9. The presented approach supports three objectives, namely speedup of the execution time, energy consumption and inserted communication overhead. The considered objectives are arranged on the x-, y-, and z-axes, accordingly. Other objectives, like, e.g., the reduction of thermal issues or the size of allocated memory may easily be added by just providing a corresponding objective evaluation function. The sequential version of the application, executed on the slowest processing unit, is located at the bottom-left of each diagram and is used as base-line, here. No data has to be communicated so that the point is directly placed on the x-y-plane, due to its sequential execution. For improved readability, vertical bars are added to the diagrams to project the points into the x-y-plane. The third dimension (i.e., the amount of communicated bytes) can be compared by the height of these bars. A solid

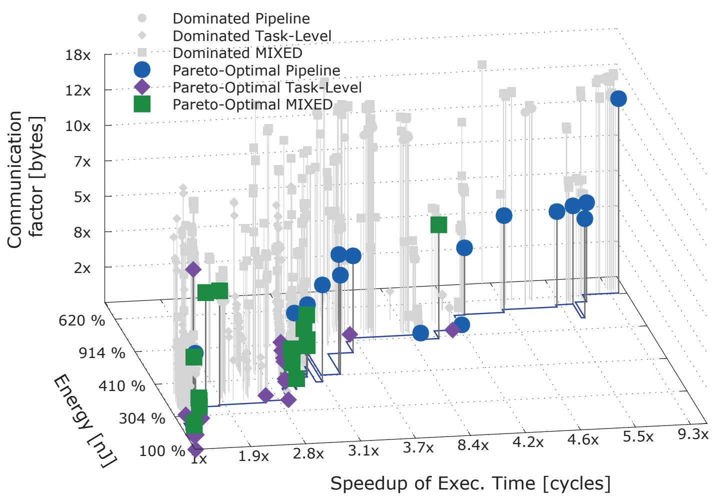


Figure 8.7: Final Parallel Solutions for the Spectral Benchmark.

line marks the front of Pareto-optimal solutions, also projected to the x-y-plane[3]. Each diagram contains both, Pareto-optimal and Pareto-dominated solutions. Of course, only the first ones are finally returned to the application designer as possible solution candidates. The diagrams contain three different shapes to mark solution candidates containing parallel sections generated by the task-level parallelization approach (cf. Section 8.2.1), the pipeline parallelization approach (cf. Section 8.2.2) and a mixture of both approaches. All objective values of the presented solutions are based on the high-level models presented in Section 8.2.3.

The visualized benchmarks have been selected since they show different behaviors with respect to the evaluated parallelization approaches and the maximum objective values. The *edge detect* application, for example, shown in Figure 8.6 profits most from the presented pipeline parallelization approach. Only a few solutions generated by the task-level parallelization approach are part of the Pareto-frontier since most parallelism is hidden in loops. In contrast, the solutions generated for the *spectral* benchmark shown in Figure 8.7 and the *jpeg2000* encoder shown in Figure 8.8 profit from both approaches and also from a mixture of them. The last visualized application is the *boundary value problem* in Figure 8.9. Here, fewer solutions are generated, but it was possible to extract a speedup of over 12× which is close to the theoretical speedup limit of the targeted embedded MPSoC. However, the solution increases the energy consumption to 940% compared to the solution which is executed sequentially on the slowest processing unit only. This highlights the trade-offs which are enabled by the presented parallelization framework due to the new multi-objective aware approach. If the application designer knows that a speedup of, e.g., 3.4× is sufficient for the considered application scenario of the *boundary value problem*, a solution which uses less and also slower processing units

[3]The projected Pareto-frontier is not in a straight echelon form due to the third objective. Even if a solution is worse in execution time and energy consumption, it may be added to the front of Pareto-optimal solutions if it requires less communication overhead.

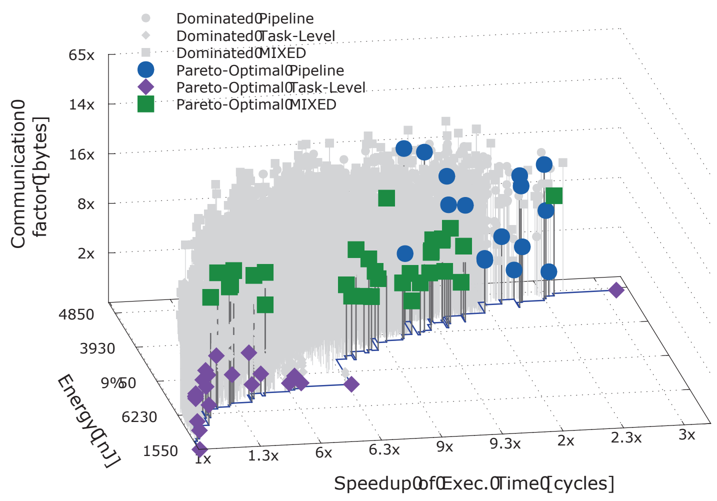


Figure 8.8: Final Parallel Solutions for the jpeg2000 Benchmark.

can be chosen so that the energy consumption is only increased to around 500% instead of 940%. If 2.2× is sufficient, a solution with an energy consumption of 320% could also be chosen to further decrease the system's overall energy consumption. Comparable trade-offs were also observed for the other evaluated benchmarks. The fastest solution, extracted for the *edge detect* benchmark shown in Figure 8.6, increases the system's energy consumption to 900% while gaining a speedup of 9×. If a speedup of 5× is sufficient, more energy efficient cores can be used for execution to reduce the energy consumption to 700%.

Most solutions generated by the presented pipeline parallelization approach extract the highest speedups at the cost of the system's energy consumption. Moreover, a large amount of data has to be communicated for this approach. The solutions generated by task-level parallelism are the ones producing only a small speedup but are much more energy efficient. The solutions which contain both, parallel sections based on task-level and pipeline parallelism, are mostly a good trade-off between higher speedups and lower energy consumption. This shows that both approaches are able to extract efficient parallelism from sequentially written embedded applications and that the approaches are applicable best for different objectives.

The observations made in this evaluation have revealed that the consideration of heterogeneity was an advantageous extension to the existing homogeneous multi-objective aware parallelization approaches. Even more useful trade-offs can be provided by the newly presented approaches compared to the homogeneous ones. As a result, the solution space of the homogeneous parallelization approaches could successfully be extended. This is also shown by the dense cloud of solution points shown in Figures 8.6 - 8.8. Significantly higher speedups and a wider range of optimizations for all considered objectives could be reached due to the heterogeneous performance properties of the available processing units. The results also motivate to invest additional time in the future to examine other objectives, as well.

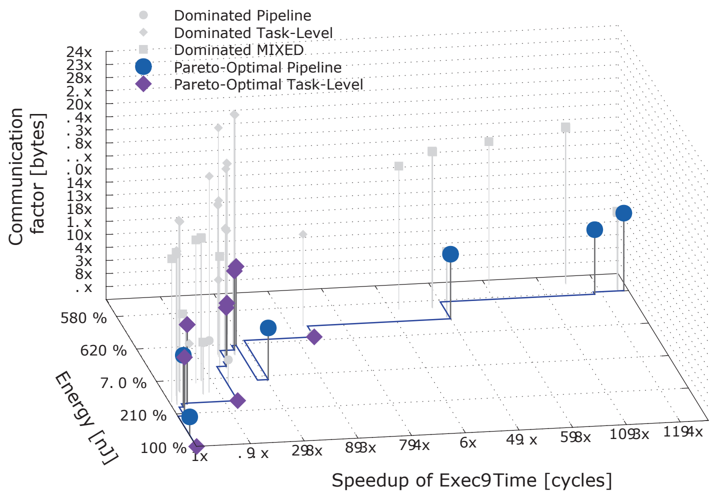


Figure 8.9: Final Parallel Solutions for the Boundary Value Benchmark.

8.3.1 Statistics of the GA-based Approaches

Table 8.1 summarizes the results for all evaluated benchmarks while providing additional statistics about timing and the employed Genetic Algorithm. The columns contain information about the required time to extract the final solution space in minutes and seconds (*Time*), the number of parallelized hierarchical nodes (#*N*), the number of created populations (#*Popul*), the number of created individuals (#*Ind*), the number of performed mutations (#*Mut*) and cross-over operations (#*Cross*) as well as the number of Pareto-optimal solutions (#*S*) returned to the application designer. All numbers are summed up over all performed parallelization steps. The population sizes as well as the number of populations are determined dynamically so that nodes with a larger number of child nodes require more time than nodes with a smaller number of child nodes. Thus, the number of created solution candidates automatically scales with the size of the solution space. It may seem that the number of solutions in the last column differs from the ones visible in Figures 8.6 - 8.9. In fact, some of the points are just too close to each other to be noticeable.

As can be seen, the number of finally returned Pareto-optimal solutions ranges between 30 solutions for the *compress* benchmark up to 333 solutions for the *jpeg2000* encoder. Nevertheless, the number of extracted solutions shows the huge optimization potential for trade-offs generated by the newly presented multi-objective aware parallelization approach. Another important aspect is the time the approach needs to parallelize an application. The time varies between 30 seconds for the *fir* benchmark up to 11 minutes for the *compress* benchmark measured on a system with four AMD-Opteron cores running at 2.4 GHz. To compensate for the higher complexity of the heterogeneous approach, mutation, recombination, and evaluation of the different individuals were parallelized to run on multiple cores. This was not implemented in the framework at the time the homogeneous approaches were eval-

Benchmark	Time[4]	#N	#Popul	#Ind	#Mut	#Cross	#S
adpcm enc.	01:31	36	1,520	153,453	30,729	104,962	63
bound. value	01:17	12	644	83,973	17,900	54,964	34
compress	10:59	289	8,936	706,266	141,170	464,112	30
edge detect	02:43	105	2,872	200,371	40,002	133,096	118
filterbank	03:24	7	412	50,779	12,229	44,164	47
fir 256 64	00:30	13	388	29,889	6,629	21,515	44
iir 4 64	03:02	13	852	105,224	22,631	79,206	63
jpeg2000	05:13	62	2,868	312,242	68,142	246,427	333
latnrm 32 64	01:11	17	636	53,642	11,462	39,831	54
mult 10 10	01:01	36	1,060	70,855	14,635	57,226	90
spectral	02:25	51	2,260	213,230	44,696	158,624	114
average	03:01	58	2,041	179,993	37,293	127,648	90

Table 8.1: Evaluation of Combined GA-based Parallelization Approaches for Heterogeneous MPSoCs

uated so that the execution times of the more complex heterogeneous approaches are sometimes even lower or in the same order of magnitude. For the *jpeg2000* encoder, over 300,000 solution candidates were created, evaluated and mutated or recombined. This means that creation and evaluation of one individual could be done in less than a millisecond due to the use of the proposed high-level models. Otherwise, it would not be possible to evaluate so many individuals which would result in a drastic reduction of the solution quality of a Genetic Algorithm.

8.4 Summary

This section presented the final parallelization approaches tailored towards resource restricted embedded devices of this thesis. Since heterogeneity has proven to be a key aspect for modern, efficient embedded systems in Chapter 7, the homogeneous multi-objective aware parallelization approaches presented in Chapter 6 were extended in this section to be able to distinguish between differing performance characteristics of the available processing units of heterogeneous architectures. The empirical evaluation on typical real-world embedded applications has shown that the new heterogeneous parallelization approaches – extracting task-level as well as pipeline parallelism – are able to provide a large number of parallel solution candidates. With these solution candidates, the application designer can optimize for different optimization objectives leading to either high speedups, low energy consumption, low communication overhead, or to solutions with good trade-offs between the different objectives.

Also, a combination of the presented task-level and pipeline parallelization ap-

[4]Time format MM:SS, measured on a system with four AMD Opteron cores running at 2.4GHz

proaches contributed additional solutions to the space of Pareto-optimal solution points. These solution candidates represent trade-offs between the advantages and limitations of both parallelization approaches adapted to heterogeneous embedded MPSoCs. It is easy to integrate additional objectives to the heterogeneous GA-based parallelization approaches since only a high-level objective evaluation function has to be provided. Other objectives, like, e.g., memory consumption, could easily be integrated by future research work as well. For completeness, it should be mentioned here, once again, that the evaluation of the high-level models' precision has to be examined in the future, since no energy model for one of the employed heterogeneous target platforms was available.

To summarize, the following results were achieved:

1. The consideration of differing performance characteristics of processing units available in a heterogeneous MPSoC combined with mapping decisions in the parallelization process could be exploited.

2. The presented framework is able to provide the application designer with a large number of solution candidates for trade-offs between different objectives.

3. The combination of task-level and pipeline parallelization approaches optimized for heterogeneous MPSoCs in one framework is advantageous since both approaches perfectly complement each other.

4. The extension to heterogeneity perfectly extends the solution space spanned by the homogeneous parallelization approaches of Chapter 6.

Chapter 9

Summary and Future Work

Contents

Recent years have shown a dramatic increase in the complexity of software written for embedded and cyber-physical systems. This resulted in demands for more computational performance which could not be satisfied by state-of-the-art embedded single-core architectures. As a consequence, Multiprocessor System-on-Chip (MPSoC) architectures gained more and more importance in the domain of embedded systems. Nowadays, high-performance embedded devices can hardly be designed without using multi-core processors. Compared to single-core platforms, MPSoCs contain multiple processing units clocked with lower CPU frequencies. By distributing the work of the considered application to multiple processing units, the same amount of work can often be processed with a lower energy consumption and less heat dissipation compared to single-core architectures. Unfortunately, these benefits do not come for free. Many new challenges have to be tackled if existing embedded applications are to be ported to such multiprocessor systems.

Most embedded applications are still written in sequential C code in such a way that they are not enabled to use multiple processors. Thus, to exploit the full potential of MPSoC platforms, applications have to be partitioned into several concurrently executed tasks to enable parallel execution on the available processing units. Since manual parallelization tends to be very error prone and time consuming, the application designer should be relieved from the burden of manually parallelizing an application.

Even though automatic parallelization is a research area for decades, resulting in a significant amount of available parallelization tools, only a minority of them can be reasonably applied to resource-restricted embedded devices. For such systems, aspects, like, e.g., energy consumption and heterogeneity have to be taken into account which were hardly ever considered by existing parallelization approaches so far. Therefore, this thesis presents a new parallelization framework containing several novel parallelization techniques, which are especially tailored towards resource-restricted embedded systems. The benefits of the developed approaches and their contribution to the current state of research are summarized in Section 9.1 before Section 9.2 finally concludes this thesis with an outlook to future research possibilities.

9.1 Research Contributions

The initial considerations that inspired this thesis (cf. Chapter 1) revealed many aspects which have to be taken into account if embedded applications should be mapped efficiently to embedded MPSoCs. From these characteristics, ambitious goals were defined. This chapter reviews these goals and checks if they were accomplished.

Task Balancing: Efficient task balancing has been one of the most important aspects which was determined to be essential in the beginning of this thesis. If tasks are not well balanced, a lot of performance may be wasted. This also has a negative impact on other important objectives, like, e.g., the energy consumption of the embedded device. Unfortunately, many existing parallelization approaches do not apply cost models or only use extremely rudimentary greedy algorithms to extract tasks from sequentially written applications. Therefore, several Integer Linear Programming and Genetic Algorithm-based parallelization approaches were presented in Chapters 5 - 8 which inherently contain cost models for the considered objectives. Based on these models, the approaches were able to extract well-balanced tasks and determine whether parallel execution really accelerates the applications. For most of the presented approaches, speedups which are close to the theoretical speedup limit of the targeted architectures could be reached. The *compress* benchmark, for example, could be accelerated by factors of nearly 2.0×, 2.6×, and 3.9× for two, three and four tasks running in parallel on the Arm11MPCore platform for the homogeneous ILP-based parallelization approaches (cf. Chapter 5). The performed evaluations highlight the applicability of the proposed approaches to embedded devices.

Complexity: Large complexity is always a problem for approaches dealing with automatic extraction of parallelism from sequentially written applications. This is one of the reasons why most existing approaches disregard the use of cost models or rely on simple heuristics. Since the approaches developed in this thesis employ cost models and sophisticated parallelization techniques based on Integer Linear Programming and Genetic Algorithms, the complexity of the vast solution space had to be pruned in a smart way. Therefore, Chapter 4 presents the Augmented Hierarchical Task Graph which was used as central intermediate representation by the developed parallelization framework including all approaches presented in Chapters 5 - 8. By dividing the application to be parallelized into different hierarchical levels which are processed in isolation by the proposed divide-and-conquer based approach, it was possible to drastically reduce the vast solution space. This was crucial for the complex parallelization approaches presented in this thesis. Even though the task-level parallelization approach, for example, presented in Section 5.2 is based on NP-complete Integer Linear Programming, solutions could be determined in 3 - 9 seconds on average for the tested benchmarks on multiple platform configurations.

Multiple Optimization Objectives: If sequentially written applications should be efficiently mapped onto resource-restricted embedded MPSoCs, the consideration of multiple optimization objectives is indispensable. Unfortunately, most previously presented parallelization approaches try to maximize the speedup without considering the impact on other objectives like, e.g., the system's energy consumption. Therefore, Chapters 6 and 8 studied such effects and presented several parallelization approaches which are able to extract different kinds of parallelism for homogeneous and also heterogeneous embedded MPSoCs. By putting some of the cores into idle mode and reducing the number of employed processing units, trade-offs between different objectives could be achieved. For a combination of the approaches presented in Chapter 6, for example, several solutions ranging from 1.7× with increased energy consumption of 200% to 2.7× with increased energy consumption of 320% were returned for the *filterbank* benchmark. If the application designer knows that a given speedup is sufficient to meet the specified deadlines, a solution consuming less energy can be beneficial. In this way, the application designer can choose a solution which perfectly fits to a specific application scenario.

Online vs. Offline Decisions: Additional overhead caused by runtime decisions should be avoided for embedded devices as much as possible, due to lower computational power of these devices compared to high-performance architectures. In contrast to, e.g., OpenMP's online task scheduling strategies, the approaches presented in this thesis extract optimized solutions by considering target architecture information. The number of extracted tasks can, for example, be limited to the amount of available processing units. In this way, it is possible to avoid additional scheduling overhead at runtime. The high speedups of the presented approaches have shown that this was a beneficial decision for the targeted embedded architectures.

Type of Parallelism: To be applicable to a large number of embedded applications from multiple application domains, it is often not sufficient to focus on the extraction of only one kind of parallelism. Therefore, each chapter presented task-level and pipeline parallelization approaches optimized for the intended use of homogeneous and heterogeneous as well as single and multi-objective aware parallelization approaches. The pipeline parallelization approach is also able to extract DoAll parallelism which is some sort of data-level parallelism. Moreover, the application designer does not have to stick to one kind of extracted parallelism. The approaches presented in Chapters 5 - 8 can also be executed in a combined manner, achieved by the plug-and-play model provided by the global parallelization technique presented in Chapter 4. The evaluation sections have shown that the different parallelization techniques perfectly complement each other to extract efficient parallelism optimized for resource-restricted embedded MPSoCs.

Heterogeneity: Heterogeneity is one of the key design patterns for current and future embedded MPSoCs. These systems combine different kinds of processors on one die to tackle problems concerning processing speed, energy consumption, heat dissipation, and other objectives. While task balancing was already a complex problem for homogeneous architectures, its complexity significantly increases for heterogeneous ones. The approaches presented in Chapters 7 and 8 have shown that it is crucial to take performance variances of the available processing units into account if applications should be parallelized efficiently for heterogeneous MPSoCs. Also here, speedups which are close to the theoretical maximum could be reached for the approaches presented in Chapter 7 (up to 10-12$\times$ for an architecture with a theoretical speedup limit of 13.5$\times$). In addition, the multi-objective aware heterogeneous parallelization approaches of Chapter 8 provide solutions with high speedups, low energy consumption, low communication overhead or useful trade-offs between these objectives.

As can be seen above, all goals defined in the introduction of this thesis were accomplished and are covered by the novel parallelization approaches. All approaches have been integrated into the developed parallelization framework and were presented at conferences and workshops with a strong background on embedded systems or parallelization problems. Since all points discussed above are fulfilled, the proposed approaches are able to extract well-optimized parallelism for resource-restricted embedded MPSoCs like intended as the main goal of this thesis. To the best of the author's knowledge, such a framework – at least covering all of the above-mentioned points – was not presented before which highlights the research contribution of this thesis.

It should also be mentioned here that the approaches presented in this thesis are optimized for specific intended use cases. All of them are useful for specific circumstances. If an application designer has to, e.g., extract parallelism for homogeneous architectures and the maximization of the speedup is the main goal, the approaches presented in Chapter 5 should be used since they are optimized for this intended use case. Of course, the heterogeneous multi-objective aware parallelization approaches presented in Chapter 8 could also be used in this situation (since a homogeneous architecture is a special case of a heterogeneous one) but the homogeneous ILP-based approaches are much faster and optimized for this use case. Therefore, all presented approaches are valuable without being superfluous due to more complex ones. With the help of the developed parallelization framework, all presented parallelization approaches could be integrated into one tool flow so that the most suitable approaches can easily be selected.

9.2 Future Work

In addition to the results achieved by the approaches presented in this thesis, there is always space for future research work which can extend or improve the quality of the presented approaches. This section closes this thesis with an outlook to possible future research work grouped into different categories.

High-Level Objective Models: The high-level objective models employed in this thesis are accurate enough to be used for estimating whether parallel execution may increase the overall performance or may reduce the system's energy consumption. These models can, of course, be extended to be more precise. In the current version, for example, there is no distinction between different communication strategies. Such aspects can be easily integrated into the presented high-level cost models. However, precision and runtime complexity have to be considered carefully to take care that the parallelization approaches can still be solved in a reasonable amount of time.

Execution Time Reduction: The execution time of the first presented approaches employing ILP for homogeneous embedded MPSoCs were fast in finding efficient solutions. However, the complexity of the proposed approaches increased from chapter to chapter so that it reached an upper limit of an acceptable runtime for the ILP-based pipeline parallelization approach for heterogeneous architectures. One possibility to accelerate the presented approaches significantly is to parallelize the parallelization approach itself. This can be easily achieved by parallelizing the divide-and-conquer-based parallelization approach presented in Chapter 4. All nodes which are part of the same hierarchical level can be processed concurrently which can significantly reduce the approaches' runtime. A nice side effect is that all approaches presented in Chapters 5 - 8 would profit from parallelization if the global parallelization algorithm would be parallelized at this point.

Multi-Objective aware Approaches: Up to now, three objectives are considered by the GA-based parallelization approaches, namely speedup, the system's energy consumption, and the inserted communication overhead. Additional objectives, like, e.g., the memory consumption or heat dissipation could also be interesting to analyze. Moreover, the approaches presented in this thesis only employed SPEA2 for individual selection and variation purposes. Since SPEA2 already returned good solution candidates in a short amount of time, other tools were not evaluated. This could also be done by future research work.

ILP-based approaches: Some of the presented ILP systems can be further improved since, e.g., several solutions with the same objective value may exist. For example, the objective value does not change if the nodes of two tasks T_1 and T_2 are swapped. Such solutions can, e.g., be avoided by the techniques presented in [LMM+97]. Even though this will not have an impact on the solution quality, it may reduce the time to solve the ILP systems.

Portability: The developed approaches were evaluated on three different target architectures. It might be interesting to test the developed approaches also for additional target architectures which are more diverse. Heterogeneous architectures, for example, containing processors with different instruction sets could not be tested in the context of this thesis due to portability problems of the employed operating system. If this limitation could be eliminated, the portability of the presented approaches to such platforms should also be evaluated.

Static Dependence Analysis: In the current version, the parallelization framework employs a profile-driven analysis to detect dependencies between statements of the application to be parallelized. The main reason was that this kind of analysis could be developed significantly faster than a static one. Furthermore, this thesis focuses on parallelism extraction techniques so that not too much effort could be spent on such analysis techniques. However, the profile-driven dependence analysis is implemented in a separate MACC tool so that it can easily be exchanged for a static analysis as soon as it is available.

Additional Programming Languages: Even though most embedded applications are written in sequential C code, support for additional programming languages may be beneficial. Especially for services which are less hardware specific, source code languages with higher abstraction levels, like, e.g., C++ or Java, are often used. Therefore, the presented parallelization framework could be extended to support such programming languages as well. This would require additional fundamental work on the parallelization framework for, e.g., extracting the AHTG. However, most concepts of the parallelization approaches presented in Chapters 5 - 8 may probably be re-used for C++ and Java without adaptations.

Appendix A

Appendix

Contents

A.1 Visualization Example of the Generated AHTG

The components and key properties of the Augmented Hierarchical Task Graph (AHTG) were presented in Chapter 4. Since AHTGs are, in general, very complex even for small applications, a simplified example was used, there. For the sake of completeness, this section shows additional screen shots of the AHTG generated by the proposed parallelization framework for the *spectral* [Lee13] benchmark. The framework is able to write out the initial as well as the parallelized AHTG in the so-called graphml format [Gra13]. This format can be parsed by many high-quality graph visualization applications. The screen shots shown in Figures A.1 - Figures A.3 were generated by the yED Graph Editor [yWo13]. At this point, the author would like to thank yWorks for providing the application free of cost.

The three figures show the graph at different zoom levels to present the general graph structure as well as an example depicting the different node types and dependencies between them. Figure A.1 shows the complete graph without going too much into detail. The different hierarchical levels cannot be seen here due to the tiny zoom level. The upper left corner of Figure A.1 is presented in more detail in Figure A.2. Here, different hierarchical nodes with first details can be seen. Hierarchical nodes are visualized by blue shapes while simple nodes are represented by yellow rectangles. Communication in- and out-nodes have a white background. Data- and control dependencies are marked by directed edges between the nodes. Finally, Figure A.3 further increases the zoom level of Figure A.2. There, node ids, the labels of the statements, iteration counts, communicated data, and additional information can be seen.

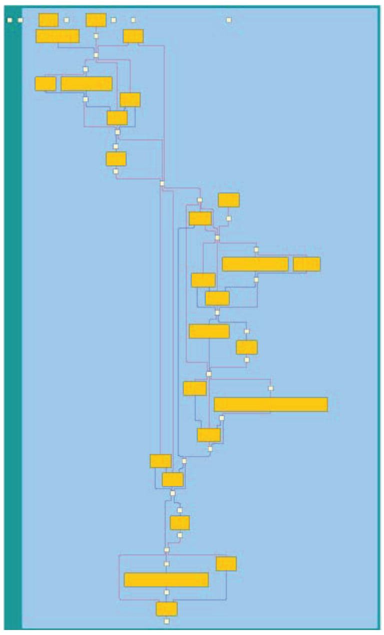


Figure A.1: Complete AHTG of the Spectral Benchmark's Main Function generated by the Parallelization Framework (visualized by the yED Graph Editor [yWo13]).

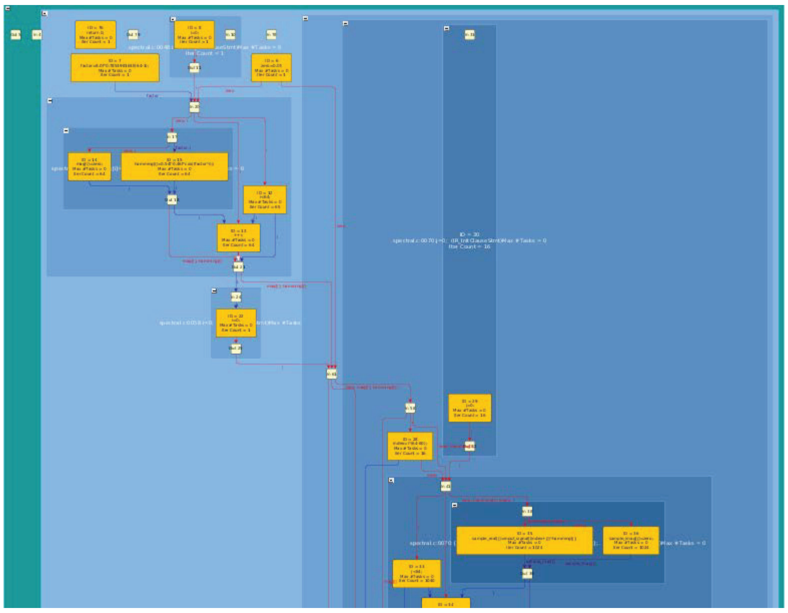


Figure A.2: More Detailed View of the AHTG of the Spectral Benchmark's Main Function (visualized with the yED Graph Editor [yWo13]).

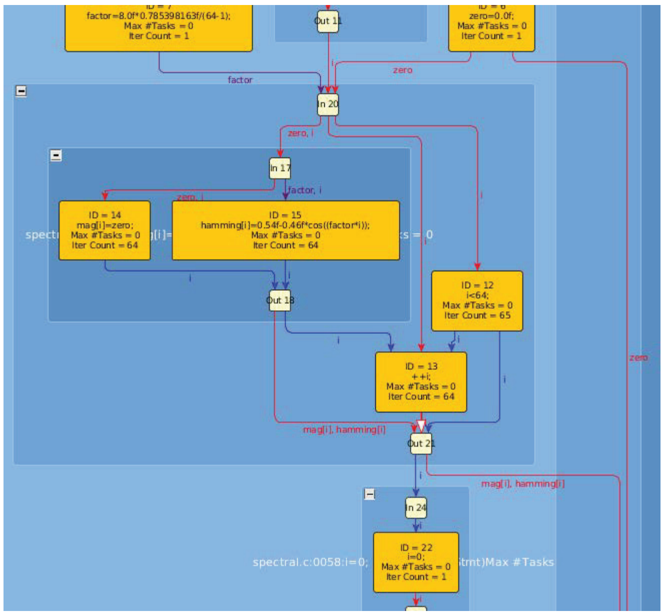


Figure A.3: More Detailed View of the AHTG of the Spectral Benchmark's Main Function (visualized with the yED Graph Editor [yWo13]).

A.2 Additional ILP Formulations

The Integer Linear Programming (ILP)-based systems presented in Chapters 5 and 7 contain operands which are not part of regular ILP formulations. This was done to improve readability of the proposed models. For completeness, this section presents and explains how these operands can be transformed to be compliant with the form of regular ILPs.

A.2.1 And-Operator in ILP

The $\wedge$ operator used in many equations can be substituted easily by a new variable and three inserted constraints as shown in Equation A.1.

$$\begin{gathered} z = (x \wedge y) \in \{0,1\} \\ z \geq x + y - 1, \qquad z \leq x, \qquad z \leq y \end{gathered} \tag{A.1}$$

A.2.2 Preconditions in ILP

Preconditions, like, e.g., $pred^{u,t} = 1$, are not part of regular ILP formulations. Nevertheless, they can be expressed by subtracting a constant whose value is greater than the sum of all other possible values of an ILP system if the precondition is not met. An example is given in Equation A.3 showing the expanded form of Equation A.2.

$$\begin{aligned} &\forall t, u \in Tasks : \forall n, o \in Nodes : t \neq u : n \neq o : \\ &pred^{t,u} \geq EDGE_{n,o} * (x_n^t \wedge x_u^o) \end{aligned} \tag{A.2}$$

The last line of Equation A.3 takes care that the equation is fulfilled automatically if task u is *not* a predecessor of task t. If the variable $pred^{u,t} = 0$, the constant $BIGCONST$ is subtracted from the right-hand side of the constraint so that it is fulfilled automatically for all variable assignments. Vice versa if $pred^{u,t} = 1$, the last line of the constraint nullifies itself.

$$\begin{aligned} &\forall t, u \in Tasks : t \neq u : \\ &accumcost^t \geq cost^t + accumcost^u + commcost^u \\ &-BIGCONST + BIGCONST * pred^{u,t} \end{aligned} \tag{A.3}$$

Bibliography

[Amd67] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, *in Proceedings of the Spring Joint Computer Conference (AFIPS)*, Atlantic City, New Jersey, 1967, pp. 483–485.

[ARM13a] ARM Ltd., *ARM1176 Processor*, June 2013, URL: `http://www.arm.com/products/processors/classic/arm11/arm1176.php` (visited on 08/01/2013).

[ARM13b] ARM Ltd., *ARM11MPCore Processor*, June 2013, URL: `http://www.arm.com/products/processors/classic/arm11/arm11-mpcore.php` (visited on 08/01/2013).

[ARM13c] ARM Ltd., *ARM7 Processor Family*, May 2013, URL: `http://www.arm.com/products/processors/classic/arm7/index.php` (visited on 08/01/2013).

[Ash10] D. Ashlock, *Evolutionary Computation for Modeling and Optimization*, Springer, 2010, ISBN: 9781441919694.

[ATN+11] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, *Journal of Concurrency and Computation: Practice & Experience (Euro-Par)* 23.2 (Feb. 2011), pp. 187–198, ISSN: 1532-0626.

[ACM+98] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture, *in Proceedings of the International Symposium on Computer Architecture (ISCA)*, Barcelona, Spain, 1998, pp. 227–237.

[BBW+09] R. Baert, E. Brockmeyer, S. Wuytack, and T. Ashby, Exploring parallelizations of applications for MPSoC platforms using MPA, *in Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, 2009, pp. 1148–1153.

[BPS+10] C. Baloukas, L. Papadopoulos, D. Soudris, S. Stuijk, O. Jovanovic, F. Schmoll, D. Cordes, R. Pyka, A. Mallik, S. Mamagkakis, F. Capman, S. Collet, N. Mitas, and D. Kritharidis, Mapping Embedded Applications on MPSoCs: The MNEMEE Approach, *in Proceedings of the International Symposium on VLSI (ISVLSI)*, 2010, pp. 512–517.

[BRMA+09] C. Baloukas, J. L. Risco-Martin, D. Atienza, C. Poucet, L. Papadopoulos, S. Mamagkakis, D. Soudris, J. Ignacio Hidalgo, F. Catthoor, and J. Lanchares, Optimization methodology of dynamic data structures based on genetic algorithms for multimedia embedded systems, *Journal of Systems and Software* 82.4 (Apr. 2009), pp. 590–602, ISSN: 0164-1212.

[BBB+05] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, MPARM: Exploring the Multi-Processor SoC Design Space with SystemC, *Journal of VLSI Signal Processing Systems* 41.2 (Sept. 2005), pp. 169–182, ISSN: 0922-5773.

[Ben11] N. Benoit, Compilation back-end pour les processeurs dédiés des systèmes multi-processeurs sur puce, PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2011.

[BKP13] M. Berkelaar, E. Kjell, and N. P., *lp_solve 5.5*, June 2013, URL: `http://lpsolve.sourceforge.net/5.5` (visited on 08/01/2013).

[BF02] A. Bhowmik and M. Franklin, A general compiler framework for speculative multithreading, *in Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA)*, Winnipeg, Manitoba, Canada, 2002, pp. 99–108.

[BLT+03] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, PISA: a platform and programming language independent interface for search algorithms, *in Proceedings of the International Conference on Evolutionary Multi-Criterion Optimization (EMO)*, Faro, Portugal, 2003, pp. 494–508.

[BEF+95] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford, Effective Automatic Parallelization with Polaris, *International Journal of Parallel Programming* (1995).

[BJK+95] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, Cilk: an efficient multithreaded runtime system, *ACM SIGPLAN Notices* 30.8 (Aug. 1995), pp. 207–216, ISSN: 0362-1340.

[BHR+08] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, A practical automatic polyhedral parallelizer and locality optimizer, *in Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, Tucson, AZ, USA, 2008, pp. 101–113.

[BRD00] A. Brokalakis, E. Rijpkema, and E. Deprettere, Compaan: deriving process networks from Matlab for embedded signal processing architectures, *in Proceedings of the International Conference on Hard-*

ware/Software Codesign and System Synthesis (CODES+ISSS), 2000, pp. 13–17.

[Bus09] Business Insider, *CHART OF THE DAY: Smartphone Sales To Beat PC Sales By 2011*, Aug. 2009, URL: `http://www.businessinsider.com/chart-of-the-day-smartphone-sales-to-beat-pc-sales-by-2011-2009-8` (visited on 08/01/2013).

[Bus11] Business Insider, *IT'S OFFICIAL: The Smartphone Market Is Now Bigger Than The PC Market*, Feb. 2011, URL: `http://www.businessinsider.com/smartphone-bigger-than-pc-market-2011-2` (visited on 08/01/2013).

[CJH+12a] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks, HELIX: automatic parallelization of irregular programs for chip multiprocessing, *in Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, San Jose, California, 2012, pp. 84–93.

[CJH+12b] S. Campanoni, T. Jones, G. Holloway, G.-Y. Wei, and D. Brooks, The HELIX project: overview and directions, *in Proceedings of the Design Automation Conference (DAC)*, San Francisco, California, 2012, pp. 277–282.

[CDF+11] E. Cannella, L. Di Gregorio, L. Fiorin, M. Lindwer, P. Meloni, O. Neugebauer, and A. Pimentel, Towards an ESL design framework for adaptive and fault-tolerant MPSoCs: MADNESS or not?, *in Proceedings of the Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, 2011, pp. 120–129.

[CSL11] J. Castrillon, W. Sheng, and R. Leupers, Trends in embedded software synthesis, *in Proceedings of the International Conference on Embedded Computer Systems (SAMOS)*, 2011, pp. 347–354.

[CCS+08] J. Ceng, J. Castrillon, W. Sheng, H. Scharwachter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda, MAPS: An integrated framework for MPSoC application parallelization, *in Proceedings of the Design Automation Conference (DAC)*, 2008, pp. 754–759.

[CGC+08] S. Chan, G. Gao, B. Chapman, T. Linthicum, and A. Dasgupta, Open64 compiler infrastructure for emerging multicore/manycore architecture, *in Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS)*, 2008, pp. 1–1.

[CGS+05] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, X10: an object-oriented approach to non-uniform cluster computing, *ACM SIGPLAN Notices* 40.10 (Oct. 2005), pp. 519–538, ISSN: 0362-1340.

[CM10] S. Cho and R. Melhem, On the Interplay of Parallelization, Program Performance, and Energy Consumption, *IEEE Transactions on Parallel and Distributed Systems* 21.3 (2010), pp. 342–353, ISSN: 1045-9219.

[CNO+88] R. Colwell, R. Nix, J. O'Donnell, D. Papworth, and P. Rodman, A VLIW architecture for a trace scheduling compiler, *IEEE Transactions on Computers* 37.8 (1988), pp. 967–979, ISSN: 0018-9340.

[CM12] D. Cordes and P. Marwedel, Multi-objective aware extraction of task-level parallelism using genetic algorithms, *in Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, Mar. 2012, pp. 394–399.

[Cor08] D. Cordes, Schleifenanalyse für einen WCET-optimierenden Compiler basierend auf Abstrakter Interpretation und Polylib (in german), Diploma Thesis, Technische Universtität Dortmund, 2008.

[CEM+12] D. Cordes, M. Engel, P. Marwedel, and O. Neugebauer, Automatic Extraction of Multi-Objective Aware Pipeline Parallelism Using Genetic Algorithms, *in Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Tampere, Finland, 2012, pp. 73–82.

[CEN+13a] D. Cordes, M. Engel, O. Neugebauer, and P. Marwedel, Automatic Extraction of Multi-Objective Aware Parallelism for Heterogeneous MPSoCs, *in Proceedings of the International Workshop on Multi-/Many-core Computing Systems (MuCoCoS)*, Edinburgh, Scotland, UK, 2013.

[CEN+13b] D. Cordes, M. Engel, O. Neugebauer, and P. Marwedel, Automatic Extraction of Pipeline Parallelism for Embedded Heterogeneous Multi-Core Platforms, *in Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Montreal, Canada, 2013.

[CEN+13c] D. Cordes, M. Engel, O. Neugebauer, and P. Marwedel, Automatic Extraction of Task-Level Parallelism for Heterogeneous MPSoCs, *in Proceedings of the International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*, Lyon, France, Sept. 2013.

[CHM+11] D. Cordes, A. Heinig, P. Marwedel, and A. Mallik, Automatic Extraction of Pipeline Parallelism for Embedded Software Using Linear Programming, *in Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*, Washington, DC, USA, 2011, pp. 699–706.

[CMM10] D. Cordes, P. Marwedel, and A. Mallik, Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming, *in Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Scottsdale, Arizona, USA, 2010, pp. 267–276.

[DM98] L. Dagum and R. Menon, OpenMP: an industry standard API for shared-memory programming, *IEEE Computational Science Engineering* 5.1 (1998), pp. 46–55, ISSN: 1070-9924.

[DBM+09] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, Cetus: A Source-to-Source Compiler Infrastructure for Multicores, *IEEE Transactions on Computers* 42.12 (2009), pp. 36–42, ISSN: 0018-9162.

[EGCS+03] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared-Memory Programming*, Wiley-Interscience, 2003, ISBN: 0471220485.

[FSS11] H. Falk, N. Schmitz, and F. Schmoll, WCET-aware Register Allocation based on Integer-Linear Programming, *in Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, Porto, Portugal, 2011, pp. 13–22.

[Fea96] P. Feautrier, Automatic Parallelization in the Polytope Model, *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, London, UK, UK, 1996, pp. 79–103.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren, The program dependence graph and its use in optimization, *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.3 (July 1987), pp. 319–349, ISSN: 0164-0925.

[Fis81] J. Fisher, Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Transactions on Computers* C-30.7 (1981), pp. 478–490, ISSN: 0018-9340.

[FO05] B. Franke and M. O'Boyle, A complete compiler approach to auto-parallelizing C programs for multi-DSP systems, *IEEE Transactions on Parallel and Distributed Systems* 16.3 (2005), pp. 234–245, ISSN: 1045-9219.

[GP94] M. Girkar and C. D. Polychronopoulos, The hierarchical task graph as a universal intermediate representation, *International Journal of Parallel Programming* 22.5 (Oct. 1994), pp. 519–551, ISSN: 0885-7458.

[GTA06] M. I. Gordon, W. Thies, and S. Amarasinghe, Exploiting coarse-grained task, data, and pipeline parallelism in stream programs, *in Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, California, USA, 2006, pp. 151–162.

[GTK+02] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, A stream compiler for communication-exposed architectures, *in Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, California, 2002, pp. 291–303.

[Gra13] GraphML Team, *The GraphML File Format*, Apr. 2013, URL: `http://graphml.graphdrawing.org` (visited on 08/16/2013).

[GL97] M. Griebl and C. Lengauer, The loop parallelizer LooPo - announcement, *Languages and Compilers for Parallel Computing*, 1997, pp. 603–604.

[HAM+95] M. H. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, Detecting coarse-grain parallelism using an interprocedural parallelizing compiler, *in Proceedings of the International Conference on Supercomputing (ICS)*, San Diego, California, USA, 1995.

[HAA+96] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, Maximizing Multiprocessor Performance with the SUIF Compiler, *IEEE Transactions on Computers* 29.12 (Dec. 1996), pp. 84–89, ISSN: 0018-9162.

[Har87] D. Harel, Statecharts: A visual formalism for complex systems, *Journal of Science of Computer Programming* 8.3 (June 1987), pp. 231–274, ISSN: 0167-6423.

[HSK+08] C. Haubelt, T. Schlichter, J. Keinert, and M. Meredith, SystemCoDesigner: Automatic design space exploration and rapid prototyping from behavioral models, *in Proceedings of the Design Automation Conference (DAC)*, 2008, pp. 580–585.

[Hei10] A. Heinig, *R2G: Supporting POSIX like semantics in a distributed RTEMS system*, Technical Report 836, TU Dortmund, Faculty of Computer Science 12, 2010.

[HMC+93] W.-M. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery, The Superblock: An Effective Technique for VLIW and Superscalar Compilation, *The Journal of Supercomputing - Special issue on instruction-level parallelism*, vol. 235, 1993, pp. 229–248.

[IBM13] IBM, *IBM - High-performance mathematical programming engine - IBM ILOG CPLEX - Software*, June 2013, URL: `http://www-01.ibm.com/software/integration/optimization/cplex` (visited on 08/01/2013).

[Inf13] Informatik Centrum Dortmund e.V., *ICD-C Compiler framework*, June 2013, URL: `http://www.icd.de/index.php/en/es/icd-c-compiler/icd-c` (visited on 08/01/2013).

[Int13] Intel Corporation, *Intel Parallel Studio*, Aug. 2013, URL: `http://software.intel.com/en-us/intel-parallel-studio/home` (visited on 08/01/2013).

[IMM+10] Y. Iosifidis, A. Mallik, S. Mamagkakis, E. De Greef, A. Bartzas, D. Soudris, and F. Catthoor, A framework for automatic parallelization, static and dynamic memory optimization in MPSoC platforms, *in Proceedings of the Design Automation Conference (DAC)*, 2010, pp. 549–554.

[IJT91] F. Irigoin, P. Jouvelot, and R. Triolet, Semantical interprocedural parallelization: an overview of the PIPS project, *in Proceedings of the International Conference on Supercomputing (ICS)*, Cologne, West Germany, 1991, pp. 244–251.

[ITF12] ITFacts.biz, *49.7% of Americans own smartphones*, Mar. 2012, URL: `http://www.itfacts.biz/49-7-of-americans-own-smartphones/13008` (visited on 08/01/2013).

[JEV04] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar, Min-cut program decomposition for thread-level speculation, *in Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, Washington DC, USA, 2004, pp. 59–70.

[JMB+12] O. Jovanovic, P. Marwedel, I. Bacivarov, and L. Thiele, MAMOT: Memory-Aware Mapping Optimization Tool for MPSoC, *in Proceedings of the Euromicro Conference on Digital System Design (DSD)*, Washington, DC, USA, 2012, pp. 743–750.

[KKS02] I. Kadayif, M. Kandemir, and U. Sezer, An integer linear programming based approach for parallelizing applications in On-chip multiprocessors, *in Proceedings of the Design Automation Conference (DAC)*, New Orleans, Louisiana, USA, 2002, pp. 703–706.

[Kah74] G. Kahn, The semantics of a simple language for parallel programming, *Information processing*, Stockholm, Sweden, 1974, pp. 471–475.

[Kat08] A. Katriniok, Speicherhierarchie Design-Space-Exploration für den MPARM System-on-Chip Simulator (in german), Diploma Thesis, Technische Universtität Dortmund, 2008.

[KSS+09] J. Keinert, M. Streubühr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, SystemCoDesigner - an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications, *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 14.1 (Jan. 2009), 1:1–1:23, ISSN: 1084-4309.

[KA02] K. Kennedy and J. R. Allen, *Optimizing compilers for modern architectures: a dependence-based approach*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, ISBN: 1-55860-286-0.

[KAC+96] R. Keryell, C. Ancourt, F. Coelho, B. Creusillet, F. Irigoin, and P. Jouvelot, *PIPS: A Framework for Building Interprocedural Compilers, Parallelizers and Optimizers*, Technical Report 289, CRI, École des mines de Paris, Apr. 1996.

[KAI11] D. Khaldi, C. Ancourt, and F. Irigoin, *Towards Automatic C Programs Optimization and Parallelization using the PIPS - PoCC Integration*, 2011, URL: `http://www.cri.ensmp.fr/classement/doc/A-448.pdf` (visited on 08/01/2013).

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, Optimization by simulated annealing, *Science, Number 4598* 220 (1983), pp. 671–680.

[KTR+04] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance, *ACM SIGARCH Computer Architecture News* 32.2 (Mar. 2004), pp. 64–, ISSN: 0163-5964.

[LMD94] B. Landwehr, P. Marwedel, and R. Dömer, OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming, *in Proceedings of the Conference on European Design Automation (EURO-DAC)*, Grenoble, France, 1994, pp. 90–95.

[LMM+97] B. Landwehr, P. Marwedel, I. Markhof, and R. Dömer, Exploiting Isomorphism for Speeding-Up Instance-Binding in an Integrated Scheduling, Allocation and Assignment Approach to Architectural Synthesis, *in Proceedings of the International Conference on Computer Hardware Description Languages and their Applications (CHDL)*, Toledo, Spain, 1997.

[Lee13] C. G. Lee, *UTDSP Benchmark Suite*, May 2013, URL: `http://www.eecg.toronto.edu` (visited on 08/01/2013).

[LM87] E. Lee and D. Messerschmitt, Synchronous data flow, *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245, ISSN: 0018-9219.

[LJE04] S.-I. Lee, T. A. Johnson, and R. Eigenmann, Cetus–an extensible compiler infrastructure for source-to-source transformation, *Languages and Compilers for Parallel Computing*, 2004, pp. 539–553.

[LS91] C. Leiserson and J. Saxe, Retiming synchronous circuitry, *Algorithmica* 6.1-6 (1991), pp. 5–35, ISSN: 0178-4617.

[LC10] R. Leupers and J. Castrillon, MPSoC programming using the MAPS compiler, *in Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2010, pp. 897–902.

[LPC12] F. Li, A. Pop, and A. Cohen, Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs, *IEEE Micro* 32.4 (2012), pp. 19–31, ISSN: 0272-1732.

[LCW+08] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, Merge: a programming model for heterogeneous multi-core systems, *in Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, USA, 2008, pp. 287–296.

[LSW+08] H. Liu, Z. Shao, M. Wang, and P. Chen, Overhead-Aware System-Level Joint Energy and Performance Optimization for Streaming Applications on Multiprocessor Systems-on-Chip, *in Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2008, pp. 92–101.

[LCF+09] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models, *in Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2009, pp. 136–146.

[LHK09] C.-K. Luk, S. Hong, and H. Kim, Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping, *in Proceedings of the International Symposium on Microarchitecture (MICRO)*, New York, New York, 2009, pp. 45–55.

[Mar11] P. Marwedel, *Embedded System Design (Second Edition)*, New Jork, USA: Springer-Verlag, 2011, ISBN: 13 978-94-007-0256-1.

[MTK+11] P. Marwedel, J. Teich, G. Kouveli, I. Bacivarov, L. Thiele, S. Ha, C. Lee, Q. Xu, and L. Huang, Mapping of applications to MPSoCs, *in Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Taipei, Taiwan, 2011, pp. 109–118.

[Mer07] R. Merritt, *Embedded software stuck at C*, Sept. 2007, URL: `http://www.eetimes.com/design/other/4023198/Embedded-software-stuck-at-C` (visited on 08/01/2013).

[Mit05] G. G. Mitchell, Validity Constraints and the TSP - GeneRepair of Genetic Algorithms, *in Proceedings of the Conference on Artificial Intelligence and Applications (AIA)*, Cambridge, UK, 2005, pp. 306–311.

[Mit98] M. Mitchell, *An Introduction to Genetic Algorithms*, Cambridge, MA, USA: MIT Press, 1998, ISBN: 0262631857.

[MSH00] S. Moon, B. So, and M. W. Hall, Evaluating automatic parallelization in SUIF, *IEEE Transactions on Parallel and Distributed Systems* 11.1 (2000), pp. 36–49, ISSN: 1045-9219.

[Muc97] S. S. Muchnick, *Advanced compiler design and implementation*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, ISBN: 1-55860-320-4.

[Neu79] K. Neumann, *Lineare Optimierung, Spieltheorie, Nichtlinerare Optimierung, Ganzzahlige Optimierung (in german)*, München, Wien: Carl Hanser Verlag, 1979, ISBN: 0-444-85330-8.

[NBF96] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming - a POSIX standard for better multiprocessing.* O'Reilly, 1996, ISBN: 978-1-56592-115-3.

[NTS+08] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere, Daedalus: toward composable multimedia MP-SoC design, *in Proceedings of the Design Automation Conference (DAC)*, Anaheim, California, 2008, pp. 574–579.

[ORS+05] G. Ottoni, R. Rangan, A. Stoler, and D. August, Automatic thread extraction with decoupled software pipelining, *in Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2005.

[Par13] Par4All, *Par4All - Single source, multiple targets*, 2013, URL: `http://www.par4all.org` (visited on 08/01/2013).

[Pet13] Peter Greenhalgh, ARM, *Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7*, 2013, URL: `http://www.arm.com/files/downloads/big.LITTLE_Final.pdf` (visited on 08/01/2013).

[Pet66] C. A. Petri, Communication with automata, PhD thesis, Universität Hamburg, 1966.

[PCR12] J. A. Pienaar, S. Chakradhar, and A. Raghunathan, Automatic generation of software pipelines for heterogeneous parallel systems, *in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Salt Lake City, Utah, 2012, 24:1–24:12.

[PRC11] J. A. Pienaar, A. Raghunathan, and S. Chakradhar, MDR: performance model driven runtime for heterogeneous parallel platforms, *in Proceedings of the International Conference on Supercomputing (ICS)*, Tucson, Arizona, USA, 2011, pp. 225–234.

[Pol91] C. D. Polychronopoulos, The hierarchical task graph and its use in auto-scheduling, *in Proceedings of the International Conference on Supercomputing (ICS)*, Cologne, West Germany, 1991, pp. 252–263.

[PC11] A. Pop and A. Cohen, A stream-computing extension to OpenMP, *in Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, Heraklion, Greece, 2011, pp. 5–14.

[PC13] A. Pop and A. Cohen, OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs, *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (Jan. 2013), 53:1–53:25, ISSN: 1544-3566.

[PE95] B. Pottenger and R. Eigenmann, Idiom recognition in the Polaris parallelizing compiler, *in Proceedings of the International Conference on Supercomputing (ICS)*, Barcelona, Spain, 1995, pp. 444–448.

[PBC+08] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time, *ACM SIGPLAN Notices* 43.6 (June 2008), pp. 90–100, ISSN: 0362-1340.

[PBB+10] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan, Combined Iterative and Model-driven Optimization in an Automatic Parallelization Framework, *in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Washington, DC, USA, 2010, pp. 1–11.

[PKM+10] R. Pyka, F. Klein, P. Marwedel, and S. Mamagkakis, Versatile system-level memory-aware platform description approach for embedded MP-SoCs, *in Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Stockholm, Sweden, 2010, pp. 9–16.

[QNY+10] M. Qiu, J.-W. Niu, L. Yang, X. Qin, S. Zhang, and B. Wang, Energy-Aware Loop Parallelism Maximization for Multi-core DSP Architectures, *in Proceedings of the International Conference on Green Computing and Communications (GreenCom)*, 2010, pp. 205–212.

[ROR+08] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August, Parallel-stage decoupled software pipelining, *in Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Boston, MA, USA, 2008, pp. 114–123.

[RVV+04] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, Decoupled Software Pipelining with the Synchronization Array, *in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Washington, DC, USA, 2004, pp. 177–188.

[RDK00] E. Rijpkema, E. F. Deprettere, and B. Kienhuis, Deriving Process Networks from Nested Loop Algorithms, *Parallel Processing Letters* 10 (2000), pp. 165–176.

[RS82] A. Rockstrom and R. Saracco, SDL–CCITT Specification and Description Language, *IEEE Transactions on Communications* 30.6 (1982), pp. 1310–1318, ISSN: 0090-6778.

[RTE13] RTEMS, *RTEMS Operating System | Real-Time and Real Free*, June 2013, URL: `http://www.rtems.com` (visited on 08/01/2013).

[Sar91a] V. Sarkar, Automatic partitioning of a program dependence graph into parallel tasks, *IBM Journal of Research and Development* 35.5-6 (Sept. 1991), pp. 779–804, ISSN: 0018-8646.

[Sar89] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, Cambridge, MA, USA: MIT Press, 1989, ISBN: 0262691302.

[Sar91b] V. Sarkar, The PTRAN Parallel Programming System, *Parallel Functional Programming Languages and Compilers* (1991), pp. 309–391.

[SH86] V. Sarkar and J. Hennessy, Partitioning parallel programs for macro-dataflow, *in Proceedings of the International Conference on LISP and Functional Programming (LFP)*, Cambridge, Massachusetts, USA, 1986, pp. 202–211.

[Sch75] H. Schwefel, Evolutionsstrategie und numerische Optimierung (in german), Technische Universität Berlin, 1975.

[SSO+13] W. Sheng, S. Schürmans, M. Odendahl, M. Bertsch, V. Volevach, R. Leupers, and G. Ascheid, A compiler infrastructure for embedded heterogeneous MPSoCs, *in Proceedings of the International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*, Shenzhen, Guangdong, China, 2013, pp. 1–10.

[SOHL+98] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd. Edition (Revised), Cambridge, MA, USA: MIT Press, 1998, ISBN: 0262692155.

[SGS10] J. Stone, D. Gohara, and G. Shi, OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems, *Computing in Science Engineering* 12.3 (2010), pp. 66–73, ISSN: 1521-9615.

[SGB10] S. Stuijk, M. Geilen, and T. Basten, A Predictable Multiprocessor Design Flow for Streaming Applications with Dynamic Behaviour, *in Proceedings of the Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*, Washington, DC, USA, 2010, pp. 548–555.

[SFF+05] M. Suzuki, N. Fujinami, T. Fukuoka, T. Watanabe, and I. Nakata, SIMD optimization in COINS compiler infrastructure, *Innovative Architecture for Future Generation High-Performance Processors and Systems*, 2005.

[Syn13a] Synopsys, *Synopsis CoMET-METeor*, May 2013, URL: `http://www.synopsys.com/Systems/VirtualPrototyping/Pages/CoMET-METeor.aspx` (visited on 08/01/2013).

[Syn13b] Synopsys, *Synopsis Virtualizer*, May 2013, URL: `http://www.synopsys.com/Systems/VirtualPrototyping/Pages/Virtualizer.aspx` (visited on 08/01/2013).

[Tar72] R. Tarjan, Depth-First Search and Linear Graph Algorithms, *SIAM Journal on Computing* 1.2 (1972), pp. 146–160.

[The13] The Free Software Foundation, *GCC, the GNU Compiler Collection*, Aug. 2013, URL: `http://gcc.gnu.org` (visited on 08/01/2013).

[TBH+07] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, Mapping Applications to Tiled Multiprocessor Embedded Systems, *in Proceedings of the International Conference on Application of Concurrency to System Design (ACSD)*, Washington, DC, USA, 2007, pp. 29–40.

[TCA07] W. Thies, V. Chandrasekhar, and S. Amarasinghe, A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs, *in Proceedings of the International Symposium on Microarchitecture (MICRO)*, Washington, DC, USA, 2007, pp. 356–369.

[TKA02] W. Thies, M. Karczmarek, and S. P. Amarasinghe, StreamIt: A Language for Streaming Applications, *in Proceedings of the International Conference on Compiler Construction (CC)*, London, UK, 2002, pp. 179–196.

[TIO13] TIOBE, *TIOBE Programming Community Index for May 2013*, May 2013, URL: `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html` (visited on 08/01/2013).

[TF10] G. Tournavitis and B. Franke, Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information, *in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Vienna, Austria, 2010, pp. 377–388.

[TF09] G. Tournavitis and B. Franke, Towards Automatic Profile-Driven Parallelization of Embedded Multimedia Applications, *in Proceedings of the Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, Paphos, Cyprus, 2009, pp. 53–64.

[TWF+09] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping, *in Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, 2009, pp. 177–187.

[TBR+06] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August, A framework for unrestricted whole-program optimization, *in Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Ontario, Canada, 2006, pp. 61–71.

[TK04] R. Turjan and B. Kienhuis, Translating affine nested-loop programs to process networks, *in Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2004, pp. 220–229.

[VNS07] S. Verdoolaege, H. Nikolov, and T. Stefanov, pn: a tool for improved derivation of process networks, *EURASIP Journal on Embedded Systems* 2007.1 (Jan. 2007), pp. 19–19, ISSN: 1687-3955.

[VE99] M. Voss and R. Eigenmann, Reducing parallel overheads through dynamic serialization, *in Proceedings of the International Symposium on Parallel Processing and the Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, 1999, pp. 88–92.

[WLL+11] Y. Wang, H. Liu, D. Liu, Z. Qin, Z. Shao, and E. H.-M. Sha, Overhead-aware energy optimization for real-time streaming applications on multiprocessor System-on-Chip, *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 16.2 (Apr. 2011), 14:1–14:32, ISSN: 1084-4309.

[Wan11] Z. Wang, Machine Learning Based Mapping of Data and Streaming Parallelism to Multi-cores, PhD thesis, University of Edinburgh, 2011.

[WO10] Z. Wang and M. F. O'Boyle, Partitioning streaming parallelism for multi-cores: a machine learning based approach, *in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Vienna, Austria, 2010, pp. 307–318.

[WM06] L. Wehmeyer and P. Marwedel, *Fast, Efficient and Predictable Memory Accesses*, The Netherlands: Springer, 2006, ISBN: 1-4020-4821-1.

[WFW+94] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, SUIF: an infrastructure for research on parallelizing and optimizing compilers, *ACM SIGPLAN Notices* 29.12 (Dec. 1994), pp. 31–37, ISSN: 0362-1340.

[Won12] W. Wong, *Why are You Still Using C?*, Apr. 2012, URL: `http://electronicdesign.com/blog/why-are-you-still-using-c` (visited on 08/01/2013).

[yWo13] yWorks, *yEd - Graph Editor*, May 2013, URL: `http://www.yworks.com/de/products_yed_about.html` (visited on 08/01/2013).

[ZIU+08] M. Zalfany Urfianto, T. Isshiki, A. Ullah Khan, D. Li, and H. Kunieda, A Multiprocessor SoC Architecture with Efficient Communication Infrastructure and Advanced Compiler Support for Easy Application Development, *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E91-A.4 (Apr. 2008), pp. 1185–1196, ISSN: 0916-8508.

[ZHC02] Y. Zhang, X. Hu, and D. Chen, Task scheduling and voltage selection for energy minimization, *in Proceedings of the Design Automation Conference (DAC)*, 2002, pp. 183–188.

[ZS02] C. Zilles and G. Sohi, Master/slave speculative parallelization, *in Proceedings of the International Symposium on Microarchitecture (MICRO)*, Istanbul, Turkey, 2002, pp. 85–96.

[ZLT01] E. Zitzler, M. Laumanns, and L. Thiele, *SPEA2: Improving the Strength Pareto Evolutionary Algorithm*, Technical Report, Eidgenössische Technische Hochschule Zürich (ETH), Institut für Technische Informatik und Kommunikationsnetze (TIK), 2001.

List of Figures

List of Algorithms

List of Tables

Index